Symbolic Algebra and Mathematics with Xcas

Renée De Graeve, Bernard Parisse[*],
Jay Belanger[†]
Sections written by Luka Marohnić[‡]

[*]Université de Grenoble, initial translation of parts of the French user manual
[†]Full translation and improvements
[‡]Optimization, signal processing. The graph theory is in a separate manual.

renee.degraeve@wanadoo.fr
bernard.parisse@ujf-grenoble.fr

Contents

1 Introduction

1.1 Notations used in this manual

User input in XCAS typically takes one of the following three forms:

Commands that you enter on the command line. For example, to compute $\sin\frac{\pi}{4}$, you can type

```
> sin(pi/4)
```

The output will be typeset in blue and the messages in green.

Keyboard shortcuts. These are indicated by separating the prefix key and the standard key with a plus +. For example, to exit an XCAS session, you can type the control key along with the q key, which will be denoted by Ctrl+Q.

Menu commands. When denoting menu items (which are typeset in sans serif font), the submenus are connected by ▶. For example, from within XCAS you can choose the File menu, then choose the Open submenu, and then choose the File item. This will be indicated by File ▶ Open ▶ File.

When describing command syntax, specific values that you enter for arguments are in `typewriter` font, while argument placeholders that should be replaced by actual values are in *italics*. Optional arguments will be enclosed by angular brackets. For example, you can find the derivative of an expression with the `diff` command (see Section 13.2.1, p. 277), which takes the form `diff(`*expr*$\langle$`,`*x*$\rangle$`)` where *expr* is an expression and x is (optionally) a variable or a list of variables.

In XCAS there exist many synonymous commands (i.e. having different names but the same effect). When a group of synonyms is described, the syntax details are provided only for the first command name and it is implied that the instructions apply to the synonyms as well.

In the index(see p. 837), the entries which are part of XCAS language are written in `typewriter` font (command names in black and programming keywords in blue) or *italics* (option names and specific values like color names).

1.2 Interfaces for the `giac` library

The `giac` library is a C++ mathematics library. It comes with two interfaces use directly; a graphical interface and a command-line interface. All interfaces can do symbolic and numeric calculations, use `giac`'s programming language, and have a built in help function.

The graphical interface is called XCAS, and is the most full-featured interface. XCAS has additional help features to make it easy to use, plus it has a built-in spreadsheet, it can do dynamic geometry and it can do turtle graphics. The output given by this interface is typeset; for example:

```
> sqrt(2)
```

$$\sqrt{2}$$

The command-line interface can be run inside a terminal, and in a graphical environment can also draw graphs. The output given by this interface is in text form; for example:

```
> sqrt(2)
```

```
sqrt(2)
```

There is also a web version, which can be run through a javascript-enabled browser (it works best with Firefox), either over the Internet or from local files. Other programs (for example, TEXMACS) have interfaces for the command-line version. Some of these interfaces, such as the two mentioned here, typeset their output.

1.2.1 Xcas interface

How you start Xcas in a graphical environment depends on which operating system you are using.

- If you are using Unix or Linux, you can usually find an entry for the program in a menu provided your desktop environment. Otherwise, you can start it from a terminal by typing

  ```
  xcas &
  ```

  If for some reason Xcas becomes unresponsive, you can open a terminal and type

  ```
  killall xcas
  ```

  This will kill any running Xcas processes. Xcas keeps an automatic backup files, so when you restart Xcas, you will be asked if you want to resume where you left off.

- If you are running Windows, use the explorer to go to the directory where Xcas is installed. In that directory is a file called `xcas.bat`. You can click on that file to start Xcas.

- If you are running Mac OS, use the Finder to go to the `xcas_image.dmg` file and double-click it. Then double-click the Xcas disk icon. Finally, you can double-click the Xcas program to launch Xcas.

When you start Xcas, a window will open with menu entries across the top, below that will be a bar giving information about the current Xcas configuration, and below that will be an entry line use to enter commands. This interface will be described in more detail later, but the menu item Help ▶ Interface will bring up an introduction.

1.2.2 Command-line interface

In Unix and MacOS you can run `giac` from a terminal with the command `icas` (the command `giac` also works). There are two ways to use the command-line interface.

1. If you just want to evaluate one expression, you can give `icas` the expression (in quotes) as a command line argument. For example, to factor the polynomial $x^2 - 1$, you can type

   ```
   icas 'factor(x^2-1)'
   ```

   in a command prompt. The result will be

   ```
   (x-1)*(x+1)
   ```

   and you will be returned to the operating system command line.

2. If you want to evaluate several commands, you can enter an interactive `giac` session by entering the command `icas` (or `giac`) by itself at a command prompt. You will then be given a prompt specifically for `giac` commands, which will look like

   ```
   0>>
   ```

   You can enter a `giac` command at this prompt and get the result.

```
0>> factor(x^2-1)
(x-1)*(x+1)
1>>
```

After the result, you will be given another prompt for `giac` commands. You can exit this interactive session by typing Ctrl+D.

Alternatively, you can run `icas` in batch mode; that is, you can have `icas` run `giac` commands stored in a file. This can be done in Windows as well as Unix and Mac OS. To do this, simply enter

```
icas filename
```

in a command prompt, where `filename` is the name of the file containing the `giac` commands.

1.2.3 Firefox interface

You can run `giac` without installing it by using a javascript-enabled web browser. Using Firefox for this is highly recommended; Firefox runs `giac` several times faster than Chrome, for example, and Firefox also supports MathML natively.

To run `giac` in Firefox, open the url https://www-fourier.ujf-grenoble.fr/~parisse/giac/xcasen.html. At the top of this page will be a button which will open a quick tutorial; the tutorial also tells you how to install the necessary files to run `giac` through Firefox without being connected to the Internet.

1.2.4 TeXmacs interface

TeXmacs (http://www.texmacs.org) is a sophisticated word processor with special mathematical features. As well as being designed to nicely typeset mathematics, it can be used as a frontend for various mathematics programs, including `giac`.

Once you have started TeXmacs, you can interactively run `giac` within it by using the menu command Insert ▶ Session ▶ Giac. Once started, you can enter `giac` commands as you would in the command-line interface. You can later re-enter a `giac` entry line by choosing it with your arrow keys or clicking on it with a mouse. The TeXmacs interface also has a menu containing `giac` commands.

Note that a `giac` session in TeXmacs may be started directly from a terminal using the script `xgiac` available in the `src` subdirectory of the `giac` source.

Within TeXmacs, you can combine `giac` commands and their output with ordinary text. To enter normal text within a `giac` session, use the menu item Focus ▶ Insert Text Field Above.

You can also use mathematical input when entering commands, which allows for fast typing of complex formulas. For details, see the `giac` plugin documentation in TeXmacs. Also, any command output in a TeXmacs session (or a part of it) may be copied back to an input field, thanks to the conversion routines between `giac` syntax and TeXmacs Scheme.

Graphics created by `giac` are converted to PDF and embedded into TeXmacs sessions. Note that this requires the Ghostscript package to be installed on your system.

1.2.5 Checking the version of `giac` that you are using

The `version` (or `giac`) command returns the version of `giac` that is running. It does not have any arguments, but it does require parentheses.

```
> version()
```

"giac 1.6.0, (c) B. Parisse and R. De Graeve, Institut Fourier, Universite de Grenoble I"

2 Xcas interface

2.1 Entry levels

The Xcas interface can run several independent calculation sessions, each session will be contained in a separate tab. Before you understand the Xcas interface, it would help to be familiar with the components of a session.

Each session can have any number of input levels. Each input level will have a number to the left of it; the number is used to identify the level. Each level can have one of the following:

Command line entry — This is the default; you can open a new command line with Alt+N.

You can enter a `giac` command (or a series of commands separated by semicolons) on a command line and send it to be evaluated by hitting enter. The result will then be displayed, and another command line will appear. You can also scroll through the command history with Ctrl+Up and Ctrl+Down.

If the output is a number or an expression, then it will appear in blue text in a small area below the input region; this area will be an expression editor (see Section 2.9, p. 28). There will be a scrollbar and a small button M to the right of this area, which is a menu providing various options.

If the output is a graphic, then it will appear in a graphing area below the input region. To the right of the graphic will be a control panel which use to manipulate the graphic (see Section 19.1.1, p. 454).

Expression editor — See Section 2.9, p. 28. You can open an expression editor with Alt+E.

2D geometry screen — See Section 19.1.1, p. 454. You can open a 2D geometry screen with Alt+G. This level will have a screen, as well as a control panel, menus and a command line to control the screen.

3D geometry screen — See Section 19.1.1, p. 454. You can open a 3D geometry screen with Alt+H. This level will have a screen, as well as a control panel, menus and a command line to control the screen.

Turtle graphics screen — You can open a turtle graphics screen with Alt+D. This level will have a screen, as well as a program editor and command line.

Spreadsheet — See Section 2.10, p. 31. You can open a spreadsheet with Alt+T. A spreadsheet can open a graphic screen.

Program editor — See Section 25.1.1, p. 647. You can open a program editor with Alt+P.

Comment line — See Section 2.8.2, p. 27. You can open a comment line with Alt+C.

Levels can be moved up and down in a session, or even moved to a different session.

The level containing the cursor is the *current level*. The current level can be evaluated or re-evaluated by typing enter.

You can select a level (for later operations) by clicking on the number in the white box to the left of the level. Once selected, the box containing the number turns black. You can select a range of levels

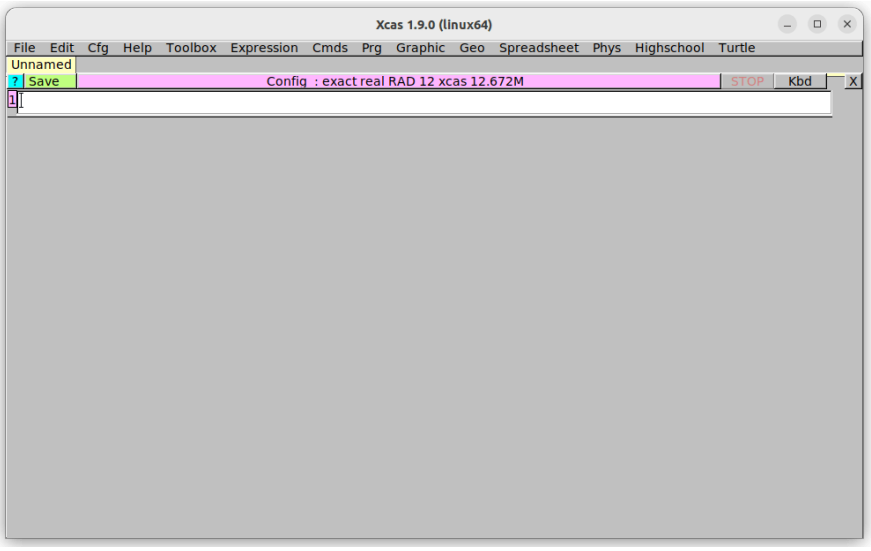


Figure 2.1: The initial Xcas window

by clicking on the number for the beginning level, and then holding the shift key while you click on the number for the ending level.

You can copy the instructions in a range of levels by selecting the range, and then clicking the middle mouse button on the number of the target level.

2.2 Starting window

When you first start Xcas, you get a largely blank window, as shown in Figure 2.1. The first row will consist of the main menus; you can save and load Xcas sessions, configure Xcas and its interface and run various commands with entries from these menus.

The second row will contains tabs; one tab for each session that you are running in Xcas. Each tab will have the name of its session, or `Unnamed` if the session has no name. The first time you start Xcas, there will be only one session, which will be unnamed.

The third row will contain various buttons.

- The button ? opens the help index (the same as the Help ▶ Index menu entry; see Section 2.3.3, p. 7). If there is a command on the command line, the help index will open at this command.
- The button Save saves the session to a file. The first time you click on it you will be prompted for a file name ending in `.xws` in which to save the session. The button will be pink if the session is not saved or if it has changed since the last change, it will be green once the session is saved. The name in the title will be the name of the file used to save the session.
- The wide button, which in the picture above says Config: exact real RAD 12 xcas 12.6728M, is a status line indicating the current Xcas configuration (see Section 2.5, p. 13). If the session is unsaved, it will begin with `Config:`; if the session is saved in a file *filename.xws*, this button will begin with `Config` *filename.xws*`:`. Other information on this status line:
  1. `exact` or `approx`
     This tells you whether Xcas will give you exact values, such as $\sqrt{2}$ when possible, or gives you decimal approximations, such as 1.4142135. (See Section 2.5.4, p. 14.)
  2. `real`, `cplx` or `CPLX`.

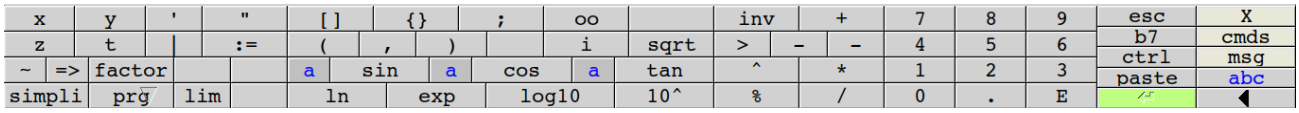


Figure 2.2: On-screen scientific keyboard in Xcas

When this shows `real` (for example), then Xcas will by default only find real solutions of equations. When this shows `cplx`, then Xcas will find complex solutions of equations. When this shows `CPLX`, then Xcas will regard variables as complex; for example, it won't simplify `re(z)` (the real part of the variable z) to `z`. (See sections 2.5.5 and 2.5.6.)

3. `RAD` or `DEG`.

   This tells you whether angles, as in trigonometric arguments, are measured in radians or degrees. (See Section 2.5.3, p. 14.)

4. An integer.

   This tells you how many significant digits will be used in floating point calculations. (See Section 2.5.1, p. 13.)

5. Xcas, `python ^=**`, `python ^=xor`, Maple, MuPAD, or TI89.

   This tells you what syntax Xcas will use. Xcas can be set to emulate the languages of Python, Maple, MuPAD, or the TI89 series of calculators. (See Section 2.5.2, p. 14.)

6. The last item tells you how much memory Xcas is using.

Clicking on this status line button opens a window where you can configure the settings shown on this line as well as some other settings; you can also open the window with the menu item Cfg ▶ CAS Configuration (see Section 2.5.7, p. 15).

- The button STOP (in red) is used to halt a computation which is running on too long.

- The button Kbd toggles an on-screen scientific keyboard at the bottom of the window, which is shown in Figure 2.2. Along the right hand side of the keyboard are some keys that can be used to change the keyboard.
  - The X key hides the keyboard, just like pressing the Kbd button again.
  - The cmds key toggles a menu bar at the bottom of the screen which can be used as an alternate menu or persistent submenu. This bar will contain buttons home, «, some menu titles, », var, cust and X.

    The « and » buttons scroll through menu items. Clicking on one of the menu buttons will perform the appropriate action or replace the menu items by its submenu items. When submenu items appear, there will also be a BACK button to return to the previous menu. Clicking on the home button returns the menu buttons to the main menu.

    After the menu buttons is a var button. This replaces the menu buttons by buttons representing the variables that you have defined. After that is a cust button, which displays commands that you store in a list variable `CST` (see section 3.3.10).

    The last button, X, closes the menu bar.
  - The msg key brings up a message window at the bottom of the window which will give you helpful messages; for example, if you save a graphic, it will tell you the name of the file it is saved in and how to include it in a LaTeX file.

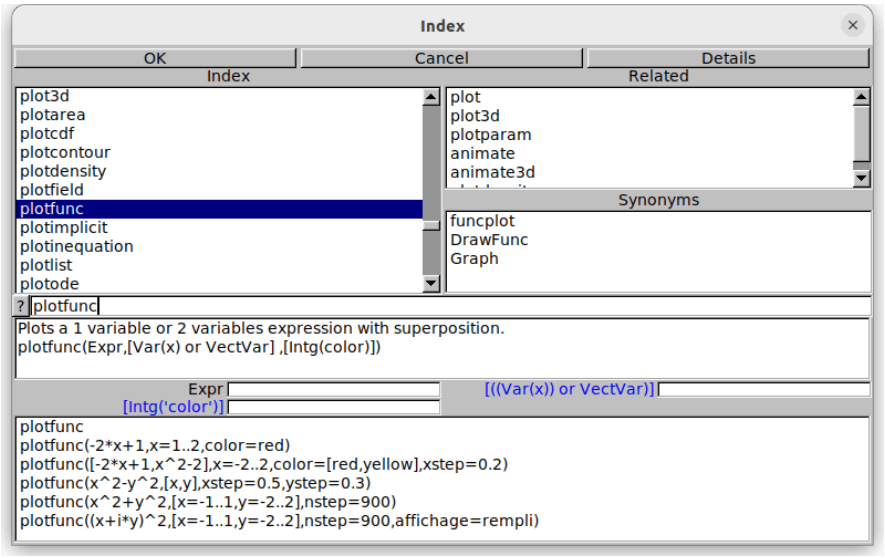


Figure 2.3: Help index in Xcas

  - The abc key toggles the keyboard between the scientific keyboard and an alphabetic keyboard.

- The button X closes the current session.

2.3 Getting help

Xcas is an extensive program, but using it is simplified with several different ways of getting help. The help menu (see section 2.4.4) has several submenus for various forms of help, some of which are mentioned below.

2.3.1 Tooltips

If you hover the mouse cursor over certain parts of the Xcas window, a temporary window will appear with information about the part. For example, if you move the mouse cursor over the status line, you will get a message saying Current CAS status. Click to modify.

If you type a function name in the Xcas command line, a similar temporary window will appear with information about the function.

2.3.2 HTML help

If you press the F12 key, you will get a window which you can use to search the html version of the manual. You can also open this window with the menu entry Help ▶ Find word in HTML help.

The HTML help window has a search area; if you type a string in that area you will be given a list of help topics that contain that string. If you choose a topic and click View, your web browser will show the appropriate page of the manual.

2.3.3 Help index

If you click on the ? button on the status line you will get the help index. You can also get the help index with the menu item Help ▶ Index.

The help index, shown in Figure 2.3, is a list of the `giac` function and variable names. You can scroll through the help index items and click on the word that you want. There is also a line in the help

index window that you can use to search the index; you can enter some text and be taken to the part of the index with the words that begin with that text. The ? button next to this search line will open the HTML help window.

If you select a function or variable name, a list of related words (names of functions or variables) and a list of synonymous words will appear in regions to the right. Below the search line, there will be an area which will have a brief description of the chosen term as well as how to call it. If the term is a command name, the calling sequence will be given as the command name with the arguments within parentheses separated by commas. Any optional arguments will be shown within brackets. In the window shown in Figure 2.3, the first argument to `plotfunc` is an expression, representing the function to be graphed. There is an optional second argument, which is either a variable name (which defaults to `x`) or a vector of variable names for multivariable functions. Finally, there is an optional third argument which can be used to specify a color for the graph.

Below the brief description will be some entry fields that use to enter the arguments. If you fill them out and press the enter key, the command with the arguments filled out will be put on the command line.

Below the entry fields for the arguments will be a list of examples of the command being used. If you click on one of these examples, it will be put on the command line.

A more thorough description of the function and its arguments is available with the Details button at the top of the help index, which will open the relevant part of the manual in your browser. Alternatively, if you click on the ? button next to the search line, you will be taken to the HTML help window.

You can also open the help index in the following ways:

- You can press the tab key while typing in the Xcas command box. If you have entered part of a command name, you will be at the part of the index with words that begin with the text that you entered.
- You can select a command from one of the menus. If `Auto index help` is chosen (see Section 2.5.9, p. 18), then the help index will open with the command chosen.

2.3.4 Getting help in a commandline

You can get help from Xcas by using the `findhelp` function. If you enter `findhelp(`*function*`)` (or equivalently `?` *function*) at the command input, where *function* is the name of a `giac` function, then some notes on *function* will appear in the answer portion and the appropriate page of the manual will appear in your web browser.

2.4 Menus

The menus provide different ways to work with Xcas and its sessions, as well as ways of inserting functions and constants into the current session. Selecting a menu item corresponding to a function or constant brings up the help index (see section 2.3.3) with the chosen function or constant selected.

2.4.1 File menu

The File menu contains commands that are used to save sessions, save parts of sessions, and load previously saved sessions. This menu contains the following entries:

New Session — This creates and opens a new session. The new session will be in a new tab, which will be labeled Unnamed until you save it (using the menu item File ▶ Save or the keystroke Alt+S).

Open — This allows you to open a previously saved session. There will be a submenu with a list of saved session files in the primary directory (see Section 4.2.1, p. 48) that you can open, as well as

a File item which will open a directory browser use to find a session file. This directory browser can also be opened with Alt+O.

Import — This allows you to open a session that was created with the Maple CAS, a TI89 calculator or a Voyage 200 calculator. You can execute this session with the Edit ▶ Execute Session menu entry, but it may be better to execute the commands one at a time to see if any modifications need to be done.

Clone — This creates a copy of the current session in a Firefox interface; either using the server at http://www-fourier.ujf-grenoble.fr/~parisse/xcasen.html (Online) or a local copy (Offline).

Insert — This allows you to insert a previously saved session, a link to a Firefox session, or a previously saved figure, spreadsheet or program.

Save (Alt+S) — This saves the current session.

Save as — This saves the current session under a name that you choose.

Save all — This saves all currently opened sessions.

Export as — This allows you to save the current session in different formats; either in χCAS(which is `giac` ported to run on various calculators) format, standard Xcas format, Xcas with Python syntax format, Maple format, MuPAD format or TI89 format.

Kill — This kills the current session.

Print — This allows you to create an image of the session in various ways.

The Preview menu item saves an image of the current session in a file that you name. The `To printer` item sends an image of the current session to the printer. The Preview selected levels item saves the images of the commands and outputs of the selected levels, each in a separate file.

LaTeX — This has submenu items that render the session in LaTeX and give you the result in various ways. The `LaTeX preview` menu item displays a compiled LaTeX version of the current session. The LaTeX print item saves a copy of the session in LaTeX form, along with the compiled version in various formats. The `LaTeX print selection` does the same as LaTeX print, but only for the selected levels.

Screen capture — This creates a screenshot that can be saved in various formats.

Quit and update Xcas — This quits Xcas after checking for a newer version.

Quit (Ctrl+Q) — This quits Xcas.

2.4.2 Edit menu

The Edit menu contains commands that are used to execute and undo parts of the current session. This menu contains the following entries:

Execute worksheet (Ctrl+F9) — This recalculates each level in the session.

Execute worksheet with pauses — This recalculates each level in the session, pausing between calculations.

Execute below — This recalculates the current level and each level below it.

Remove answers below — This removes the answers to the current level and the levels below it.

Undo — This undoes the latest edit done to the levels, including a deletion of a level. It can be repeated to undo more than one edit.

Redo — This redoes the undone editing.

Paste — This pastes the contents of the system clipboard to the cursor position.

Del selected levels — This deletes any entry levels that you have selected.

selection → LaTeX (Ctrl+T) — This puts a LaTeX version of the selection (level, part of a level, or answer selected by clicking and dragging the mouse) on the system clipboard.

New entry (Alt+N) — This inserts a new entry level above the current one.

New parameter (Ctrl+P) — This brings up a window in which you can enter a name and conditions for a new parameter.

Insert newline — This inserts a newline below the cursor. Note that simply typing return will evaluate the current entry rather than inserting a newline.

Merge selected levels — This merges the selected levels into a single level.

2.4.3 Cfg menu

The Cfg menu contains commands that are used to set the behaviour of Xcas. This menu contains the following entries:

CAS configuration — This opens a window that allows you to configure how Xcas performs calculations (see Section 2.5.7, p. 15). This is the same window you get when you click on the status line.

Graph configuration — This opens a window that allows you to configure the default settings for a graph (see Section 2.5.8, p. 17). This includes such things as the initial ranges of the variables. Each graph also has a cfg button to configure the settings on a per graph basis.

General configuration — This opens a window that allows you to configure various non-computational aspects of Xcas, such as the fonts, the default paper size, and the like (see Section 2.5.9, p. 18).

Mode (syntax) — This changes the default syntax (see Section 2.5.2, p. 14). By default, Xcas uses its own syntax, but you can change it to Python syntax, Maple syntax, MuPAD syntax or TI89 syntax.

Show — This displays parts of Xcas.

- **DispG** — This shows the graphics display screen; which has all graphical commands from the session together on one screen.
- **keyboard** — This shows the on-screen keyboard; the same as clicking on the Kbd button on the status line (see Section 2.2, p. 5, item 2.2).
- **bandeau** — This shows the menu buttons at the bottom of the window; the same as clicking on cmds on the on-screen keyboard (see Section 2.2, p. 5, item 2.2).
- **msg** — This shows the messages window; the same as clicking on msg on the on-screen keyboard (see Section 2.2, p. 5, item 2.2).

Hide — This hides the same items that you can show with Show.

Index language — This allows you to choose a language in which to display the help index.

Colors — This allows you to choose colors for various parts of the display.

Session font — This allows you to choose a font for the sessions.

All fonts — This allows you to choose fonts for the session, the main menu and the keyboard.

browser — This allows you to choose a browser that Xcas will use when needed. If this is blank, then Xcas will use its own internal browser.

Save configuration — This saves the configurations that you chose with the Cfg menu or chose by clicking on the status line.

2.4.4 Help menu

The Help menu contains commands that let you get information about Xcas from various sources. This menu contains the following entries:

Index — This brings up the help index (see Section 2.3.3, p. 7).

Find word in HTML help (F12) — This brings up a page which helps you search for keywords in the html documentation that came with Xcas (see Section 2.3.2, p. 7). The help will be displayed in your browser.

Interface — This brings up a tutorial for the Xcas interface. The tutorial will be displayed in your browser.

Reference card, fiches — This brings up a pdf reference card for Xcas. The card will be displayed in your browser.

Manuals — This allows you to choose from a variety of manuals for Xcas, which will appear in your browser.

CAS reference — This brings up the manual for Xcas.

Algorithmes (HTML) — This brings up a manual for the algorithms used by Xcas.

Algorithmes (PDF) — This brings up a pdf version of the manual for the algorithms used by Xcas.

Geometry — This brings up a manual for 2D geometry in Xcas.

Programmation — This brings up a manual for programming in Xcas.

Simulation — This brings up a manual for statistics and using the Xcas spreadsheet.

Turtle — This brings up a manual for using the Turtle drawing screen in Xcas.

Exercices — This brings up a page of exercises that you can do with Xcas.

Amusement — This brings up a page of mathematical amusements that you can work through with Xcas.

PARI-GP — This brings up documentation for the GP/PARI functions.

Internet — The `Internet` menu contains menu items that take you to various web pages related to Xcas. Among them are the following entries:

Forum — This takes you to the Xcas forum.

Update help — This installs updated help files (retrieved from the Xcas website).

There are also several menu items that take you to Xcas related pages written in French; namely:

Aide-memoire lycee — This takes you to a paper discussing Xcas and high school.

Documents pedagogiques lycee — This takes you to a page on the Xcas website with a list of useful links.

Documents algorithmique — This takes you to a page on the Xcas website with a list of links.

Site Lycee de G. Connan — This takes you to a page about a free book written by Guillaume Connan teaching algorithms to high school students.

Site Lycee de L. Briel — This takes you to a website about Xcas for high school students.

Calcul formel au lycee, par D. Chevallair — This takes you to a pdf file discussing the use of Xcas in high school.

Site de F. Han — This takes you to a website by Frederic Han about Xcas and a QT frontent for `giac`.

Ressources Capes — This takes you to a website with various external sources.

Ressources Agregation externe — This takes you to a collection of external resources.

Ressources Agregation interne — This takes you to a page on the Xcas website.

Start with CAS — This menu has the following entries:

Tutorial — This opens up the tutorial.

Solutions — This opens up the solutions to the exercises in the tutorial.

Tutoriel algo — This opens up a tutorial on algorithms and programming with Xcas.

Rebuild help cache — This rebuilds the help index.

About — This displays a message window with information about Xcas.

Examples — This allows you to choose from a variety of example worksheets, which will then be copied to your current directory and opened.

2.4.5 Toolbox menu

The Toolbox menu contains commands that are used to insert operators into the session. This menu includes the following entries:

New entry (Alt+N) — This inserts a new level.

New comment (Alt+C) — This inserts a new comment level.

The other entries let you insert mathematical operations into the current level. If Auto index help is chosen (see Section 2.5.9, p. 18), then the help index will open help index (see Section 2.3.3, p. 7) with the chosen command selected.

2.4.6 Expression menu

The Expression menu contains commands that are used to transform expressions. The first entry is Expression ▶ New expression (which is equivalent to Alt+E), which inserts a new level and brings up the on-screen keyboard (see Section 2.2, p. 5, item 2.2). The rest of the entries can be used to insert various transformations.

2.4.7 Cmds menu

The Cmds menu contains various `giac` functions and constants separated into categories. If Auto index help is chosen (see Section 2.5.9, p. 18), then when you select a function or constant, the help index (see Section 2.3.3, p. 7) opens with the function or constant selected, which can be used to insert the entry on the command line. Otherwise, the constant or function will be inserted on the command line.

2.4.8 Prg menu

The Prg menu contains commands that are used to write `giac` programs. The first entry, Prg ▶ New program (equivalent to Alt+P), inserts a program level and brings up the program editor (see Section 25.1.1, p. 647).The other entries are useful commands for writing `giac` programs.

2.4.9 Graphic menu

The Graphic menu contains commands that are used to create graphs. The first entry, Graphic ▶ Attributs (equivalent to Alt+K), brings up a window containing different attributes of the graph (such as line width, color, etc.). The other entries are commands for creating and manipulating graphs.

2.4.10 Geo menu

The Geo menu contains commands that are used to work with two- and 3D geometric figures. The first two entries, Geo ▶ New figure 2d (equivalent to Alt+G) and Geo ▶ New figure 3d (equivalent to Alt+H) create levels for two- and 3D figures, respectively. (See Section 19.1.1, p. 454.) The other menu items are for working with the figures.

2.4.11 Spreadsheet menu

The Spreadsheet menu contains commands that are used to work with spreadsheets. (See See Section 2.10, p. 31.) The first menu item, Spreadsheet ▶ New spreadsheet (equivalent to Alt+T), brings up a window where you can set the size and other attributes of a spreadsheet, after which one will be created. The submenus contain commands for working with spreadsheets. Notice that the spreadsheet itself will have menus that are the same as these submenus.

2.4.12 Phys menu

The Phys menu contains submenus with various categories of constants, as well as functions for converting units.

2.4.13 Highschool menu

The Highschool menu contains computer algebra commands that are useful at different levels of highschool. There is also a Program submenu with some program control functions.

2.4.14 Turtle menu

The Turtle menu contains the commands that are used to create and control a Turtle screen. The first menu item, Turtle ▶ New turtle, creates a Turtle drawing screen. The other menu items contain commands for working with the screen.

2.5 Configuring Xcas

2.5.1 Number of significant digits

By default, Xcas uses and displays 12 significant digits, but you can set the number of digits to other positive integers. If you set the number of significant digits to a number less than 14, then Xcas will use the computer's floating point hardware, and so calculations will be done to more significant digits than you asked for, but only the number of digits that you asked for will be displayed. If you set the number of significant digits to 14 or higher, then both the computations and the display will use that number of digits.

You can set the number of significant digits for XCAS by using the CAS configuration screen (see Section 2.5.7, p. 15). The number of significant digits is stored in the variable `DIGITS` or `Digits`, so you can also set it by giving the variable `DIGITS` a new value, as in `DIGITS:=20`. The value will be stored in the configuration file (see Section 2.5.10, p. 18), and so can also be set there.

2.5.2 Language mode

XCAS has its own language which it uses by default, but you can also use either PYTHON (with the option having the ^ character represent either exponentiation or the *exclusive or* operator) or the language used by MAPLE, MUPAD, or the TI89 calculator.

You can set which language XCAS uses in the CAS configuration screen (see Section 2.5.7, p. 15). You can also set the language with the `xcas_mode` command.

- The `xcas_mode` command takes an integer: 0, 1, 2, 3, 256 or 512.
  - `xcas_mode(0)` to use the XCAS language.
  - `xcas_mode(1)` to use the MAPLE language.
  - `xcas_mode(2)` to use the MUPAD language.
  - `xcas_mode(3)` to use the TI89 language.
  - `xcas_mode(256)` to use the PYTHON language with ^ representing exponentiation.
  - `xcas_mode(512)` to use the PYTHON language with ^ representing *exclusive or*.

The language you choose will be stored in the configuration file (see Section 2.5.10, p. 18), and so can also be set there.

2.5.3 Units for angles

By default, XCAS assumes that any angles you use (for example, as the argument to a trigonometric function) are being measured in radians. If you want, you can have XCAS use degrees.

You can set which angle measure XCAS uses in the CAS configuration screen (see Section 2.5.7, p. 15). Your choice will be stored in the variable `angle_radian`; this will be `1` if you measure your angles in radians and `0` if you measure your angles in degrees. You can also change which angle measure you use by setting the variable `angle_radian` to the appropriate value. The angle measure you want to use will be stored in the configuration file (see Section 2.5.10, p. 18), and so can also be set there.

2.5.4 Exact or approximate values

Some numbers, such as π and $\sqrt{2}$, cannot be written down exactly as decimal numbers. When computing with such numbers, by default XCAS leaves them in exact, symbolic form. If you want, you can have XCAS automatically give you decimal approximations for these numbers.

You can set whether or not XCAS gives you exact or approximate values by using the CAS configuration screen (see Section 2.5.7, p. 15). Your choice will be stored in the variable `approx_mode`, where a value of 0 means that XCAS will give you exact answers when possible and a value of 1 means that XCAS will give you decimal approximations. Your choice will be stored in the configuration file (see section 2.5.10), and so can also be set there.

2.5.5 Complex numbers

When factoring polynomials (see Section 9.1.10, p. 172), by default XCAS won't introduce complex numbers if they aren't already being used. For example,

```
> factor(x^2+2)
```

simply returns

$$x^2+2$$

but if an expression already involves complex numbers then XCAS uses them:

```
> factor(i*x^2+2*i)
```

$$\left(x-\mathrm{i}\sqrt{2}\right)\left(\mathrm{i}x-\sqrt{2}\right)$$

XCAS can also find complex roots when complex numbers are not present; for example, the command `cfactor` (see Section 9.1.10, p. 172) or `cFactor` command factors over the complex numbers. For example:

```
> cfactor(x^2+2)
```

$$\left(x+\mathrm{i}\sqrt{2}\right)\left(x-\mathrm{i}\sqrt{2}\right)$$

If you want XCAS to use complex numbers by default, you can turn on complex mode. In complex mode, the command line

```
> factor(x^2+2)
```

returns

$$\left(x+\mathrm{i}\sqrt{2}\right)\left(x-\mathrm{i}\sqrt{2}\right)$$

You can turn on complex mode from the CAS configuration screen (see Section 2.5.7, p. 15). This mode is determined by the value of the variable `complex_mode`; if this is `1` then complex mode is on, if this is `0` then complex mode is off. This option will be stored in the configuration file (see Section 2.5.10, p. 18), and so can also be set there.

2.5.6 Complex variables

By default, new variables are assumed to be real; functions which work with the real and imaginary parts of variables will assume that a variable is real. For example, `re` returns the real part of its argument and `im` returns the imaginary part (see Section 7.4.2, p. 141), and so

```
> re(z)
```

returns z and

```
> im(z)
```

returns 0.

If you want variables to be complex by default, you can have XCAS use complex variable mode. You can set this from the CAS configuration screen (see Section 2.5.7, p. 15). Your choice will be stored in the variable `complex_variables`, where a value of 0 means that XCAS will assume that variables are real and and a value of 1 means that XCAS will assume that variables are complex. Your choice will be stored in the configuration file (see Section 2.5.10, p. 18), and so can also be set there.

2.5.7 Configuring the computations

You can configure how XCAS computes by using the menu item Cfg ▶ CAS configuration or by clicking on the status line. This will open a window with the following options:

Prog style (default: XCAS) — This has a menu from which you can choose a different language to program in; you can choose from XCAS, `Python ^==**` (PYTHON syntax, except that `^` will be the exponentiation operator as in XCAS rather than the *exclusive or* operator as in PYTHON), `Python ^==xor` (PYTHON syntax, where `^` is the *exclusive or* operator), MAPLE, MUPAD and TI89/92.

eval (default: 25) — Here you can input a positive integer specifying the maximum number of recursions allowed when evaluating expressions.

prog (default: 1) — Here you can input a positive integer specifying the maximum number of recursions allowed when executing programs.

recurs (default: 100) — Here you can input a positive integer specifying the maximum number of recursive calls.

debug (default: 0) — Here you can input a 0 or 1. If this is 1, then Xcas will display intermediate information on the algorithms used by Xcas. If this is 0, then no such information is displayed.

maxiter (default: 20) — Here you can input an integer specifying the maximum number of iterations to be used in Newton's method.

Float format (default: `standard`) — This has a menu from which you can choose how to display decimal numbers. Your choices will be:

standard In standard notation, a number will be written out completely without using exponentials; for example, `15000.12` will be displayed as `15000.12`.

scientific In scientific notation, a number will be written as a number between 1 and 10 times a power of ten; for example, `15000.12` will be displayed as `1.500012000000e+04` (where the number after `e` indicates the power of 10).

engineer In engineering notation, a number will be written as a number between 1 and 1000 times a power of ten, where the power of 10 is a multiple of three. For example, `15000.12` will be displayed as `15.00012e3`.

Digits (default: 12) — Here you can input a positive integer which will indicate the number of significant digits that Xcas will use.

epsilon (default: 10^{-12}) — Here you can input a floating point number which will be the value of epsilon used by `epsilon2zero`, which is a function that replaces numbers with absolute value less than epsilon by 0 (see Section 9.4.1, p. 185).

proba (default: 10^{-15}) — Here you can input a floating point number. If this number is greater than zero, then in some cases Xcas can use probabilistic algorithms and give a result with probability of being false less than this value. (One such example of a probabilistic algorithm that Xcas can use is the algorithm to compute the determinant of a large matrix with integer coefficients.)

approx (default: unchecked) — If checked, then exact numbers such as $\sqrt{2}$ will be given a floating point approximation, otherwise exact values will be used when possible. (See Section 2.5.4, p. 14.)

autosimplify (default: 1) — Here you can input 0, 1 or 2. A value of 0 means no automatic simplification will be done, a value of 1 means grouped simplification will be automatic. A value of 2 means that all simplification will be automatic.

threads (default: 1) — Here you can enter a positive integer to indicate the number of threads (for a possible future threaded version).

Integer basis (default: 10) — This has a menu from which you can choose an integer base to work in; your choices will be 8, 10 and 16.

radian (default: checked) — If checked, then angles will be measured in radians, otherwise they will be measured in degrees.

Complex (default: unchecked) — If checked, then XCAS will work in complex mode, meaning, for example, that polynomials will be factored with complex numbers if necessary.

Cmplx_var (default: unchecked) — If checked, then variables will by default be assumed to be complex. For example, the expression `re(z)` won't be simplified, it will return `re(z)`. If unchecked, then variables by default will be assumed to be real, and so `re(z)` will be simplified to `z`.

increasing power (default: unchecked) — If checked, then polynomials will be written out in increasing powers of the variable, otherwise they will be written in decreasing powers.

All_trig_sol (default: unchecked) — If checked, then XCAS will give the complete solutions of trigonometric equations. For example, the solution of $\cos x = 0$ will be given as $[(2n\_0\pi + \pi)/2]$, where n_0 can be any integer. If unchecked, then only the primary solutions of trigonometric equations will be given. For example, the solutions of $\cos(x) = 0$ will be the pair $[-\pi/2, \pi/2]$.

Sqrt (default: checked) — If checked, then the `factor` command will factor second degree polynomials, even when the roots are not in the field determined by the coefficients. For example:

```
> factor(x^2-3)
```

$$\left(x - \sqrt{3}\right)\left(x + \sqrt{3}\right)$$

If unchecked, then the same command line returns $x^2 - 3$.

This page also has buttons for applying the settings, saving the settings for future sessions, canceling any new settings, and restoring the default settings.

2.5.8 Configuring the graphics

You can configure each graphics screen by clicking on the `cfg` button on the graphics screen's control panel to the right of the graph. You can also change the default graphical configuration using the the menu item Cfg ▶ Graph configuration. You will then be given a window in which you can change the following options:

- `X-` and `X+`: these determine the x values for which calculations will be done.
- `Y-` and `Y+`: these determine the y values for which calculations will be done.
- `Z-` and `Z+`: these determine the z values for which calculations will be done.
- `t-` and `t+`: these determine the t values for which calculations will be done (when plotting parametric curves, for example).
- `WX-` and `WX+`: these determine the range of x values for the viewing window.
- `WY-` and `WY+`: these determine the range of y values for the viewing window.
- `TX` and `TY`: these determine the tick ranges on the x- and y-axes.
- `class_min`: this determines the minimum size of a statistics class.
- `class_size`: this determines the default size of a statistics class.
- `autoscale`: when checked, the graphic will be autoscaled.
- `ortho`: when checked, all axes of the graphic will be scaled equally.

- `>W` and `W>`: these are convenient shortcuts to copy the `X-`, `X+`, `Y-` and `Y+` values to `WX-`, `WX+`, `WY-` and `WY+`, or the other way around.

Note that the viewing window is not the same as the calculation window; if the calculation window is larger than the visible window, then you can scroll to bring other parts of the calculation into view.

This page also has buttons for applying the settings, saving the settings for future sessions, or canceling any new settings.

2.5.9 More configuration

You can configure other aspects of Xcas (besides the computational aspects and graphics) using the the menu item Cfg ▶ General configuration. You will then be given a window in which you can change the following options:

Font — This lets you choose a session font, the same as choosing the menu item Cfg ▶ Session font.

Level — This determines what type of level should be open when you start a new session.

browser — This determines what browser Xcas will use when it requires one, for example when displaying help. If this is empty, Xcas will use its built-in browser.

Auto HTML help — If checked, then whenever you choose a function from a menu, a help page for that function will appear in your browser. Regardless of whether this box is checked or not, the help page will also appear in your browser if you enter `?` *function* from a command box.

Auto index help — If checked, then whenever you choose a command from a menu, the help index page for that function will appear. This is the same page you get when you choose the command from the help index. (See Section 2.3.3, p. 7.)

Print format — This determines the paper size for printing and saving files. There is also a button use to have the printing done in landscape mode; if this button is not checked, the printing will be done in portrait.

Disable Tool tips — If checked, Xcas will stop displaying tool tips (see Section 2.3.1, p. 7).

rows, columns — These determine the default number of rows and columns for the matrix editor and spreadsheet (see Section 2.10, p. 31).

PS view — This determines what program is used to preview Postscript files.

Step by step — If checked, then Xcas will not save context information.

Proxy — This sets a proxy server for updates.

2.5.10 Configuration file

When you save changes to your configuration, they are stored in a configuration file, which will be `.xcasrc` in your home directory in Unix and `xcas.rc` in Windows. This file will have four functions – `widget_size`, `cas_setup`, `xcas_mode` and `xyztrange` – which determine the configuration and which are evaluated when Xcas starts.

The `widget_size` command sets properties of the opening Xcas window.

- `widget_size` takes between 1 and 12 arguments. The arguments (in order) are:

  Font size. The first argument is a positive integer specifying the font size. Optionally, this can be a bracketed list whose first number indicates the font and the second the font size.

Horizontal and vertical offset. The second and third arguments are horizontal and vertical distances in pixels from the upper left hand corner of the screen. They specify where the upper left corner of the Xcas window is when it opens.

Window size. The fourth and fifth arguments specify the width and height in pixels of the Xcas window when it opens.

Keyboard (see Section 2.2, p. 5, item 2.2). The sixth argument is either 0 or 1; a 1 indicates that the on-screen keyboard will be open when Xcas starts, a 0 indicates that the keyboard will be hidden.

Open browser. The seventh argument is either 0 or 1; a 1 indicates that the browser will be automatically opened to display help for the selected command in the menu or index, a 0 indicates that the browser will not be automatically opened.

Message window (see Section 2.2, p. 5, item 2.2). The eighth argument is either 0 or 1; a 1 indicates that Xcas will open with the message window, a 0 indicates that Xcas will open without the message window.

<empty>. The ninth argument is currently not used.

Browser name. The tenth argument is a string with the name of the browser to use to read the help pages. A value of `"builtin"` means that Xcas will use a small browser built into Xcas.

Starting level (see Section 2.1, p. 4). The eleventh argument indicates what level Xcas will start at; a 0 means command line, a 1 means program editor, a 2 means spreadsheet, and a 3 means a 2D geometry screen.

Postscript previewer. The twelfth argument is a string with the name of a program for postscript previews; for example, `"gv"`.

The `cas_setup` command determines how computations will be performed.

- `cas_setup` takes nine arguments. The arguments (in order) are:

  Approximate mode (see Section 2.5.4, p. 14). A 1 means Xcas works in approximate mode, a 0 means exact mode.

  Complex variables (see Section 2.5.5, p. 14). A 1 means Xcas works with complex variables, a 0 means real variables.

  Complex mode (see Section 2.5.5, p. 14). A 1 means Xcas works with in complex mode, a 0 means real mode.

  Radian (see Section 2.5.3, p. 14). A 1 means work in radians, a 0 means work in degrees.

  Display format (see Section 2.5.7, p. 15, item 2.5.7). A 0 means use the standard format to display numbers, a 1 means use scientific format, a 2 means use engineering format, and a 3 means use floating hexadecimal format (which is standardized with a non-zero first digit).

  Epsilon (see Section 2.5.7, p. 15, item 2.5.7). This is the value of `epsilon` used by Xcas.

  Digits. This is the number of digits to use to display a float.

  Tasks. This will be used in the future for parallelism.

  Increasing power. This is 0 to display polynomials in increasing power, and 1 to display polynomials in decreasing powers.

The `xcas_mode` command sets the computer language used by Xcas (see Section 2.5.2, p. 14).
The `xyztrange` command sets or returns the values of the graphics configuration.

- To set the values, `xyztrange` takes 15 arguments:

  - *x–* and *x+*, the beginning and the end of the x interval for which calculations will be done.
  - *y–* and *y+*, the beginning and the end of the y interval for which calculations will be done.
  - *z–* and *z+*, the beginning and the end of the z interval for which calculations will be done.
  - *t–* and *t+*, the beginning and the end of the t interval for which calculations will be done, when plotting parametric curves, for example.
  - *wx–* and *wx+*, the beginning and the end of the x values for the viewing window.
  - *wy–* and *wy+*, the beginning and the end of the y values for the viewing window.
  - *show_axes*, to determine whether axes are shown or hidden (`1` to show, `0` to hide).
  - *class_min*, the minimum size of a statistics class.
  - *class_size*, the default size of a statistics class.

- `xyztrange(`*x–*, *x+*, *y–*, *y+*, *z–*, *z+*, *t–*, *t+*, *wx–*, *wx+*, *wy–*, *wy+*, *show_axes*, *class_min*, *class_size*`)` sets the parameters to the given values.

  Note that the viewing window is not the same as the calculation window; if the calculation window is larger than the visible window, then you can scroll to bring other parts of the calculation window into view.

- To return the values, `xyztrange` takes no arguments.

- `xyztrange()` returns a matrix where each row consists of a short description of the first twelve arguments along with their values.

2.6 Printing and saving

2.6.1 Saving a session

Each tab above the status line represents a session, the tab for the active session will be yellow. The label of each tab will be the name of the file that the session is saved in; if the session hasn't been saved the tab will read Unnamed.

You can save your current session by clicking on the Save button on the status line. If the session contains unsaved changes the Save button will be red; the button will be green when nothing needs to be saved. The first time that you save a session you will be prompted for a file name; you should choose a name that ends in `.xws`. Subsequent times that you save a session it will be saved in the same file; to save a session in a different file you can use the menu item File ▶ Save as.

If you have a session saved in a file and you want to load it in a tab, use the menu item File ▶ Open. From there you can choose a specific file from a list or open a directory browser that use to choose a file. The directory browser can also be opened with Alt+O.

2.6.2 Saving a spreadsheet

If you have a spreadsheet in one of the levels, you can save it separately from the rest of the session.

When a spreadsheet is inserted it will have menus next to the level number. The Table menu has items that let you save the spreadsheet in different formats, as well as insert previously saved spreadsheets.

You can save a spreadsheet with the Table ▶ Save sheet as text menu item. If you select that, you will be prompted for a file name; you should choose a file name that ends in `.tab`. Once you save a spreadsheet, there will be a button to the right of the menus which use to save any changes you make. If you want to save the spreadsheet under a different name, use the Table ▶ Save as alternate filename menu entry.

You can save a spreadsheet in other formats. The Table ▶ Save as CSV menu item will save a spreadsheet in a comma-separated values file, and the Table ▶ Save as mathml menu item will save the spreadsheet in as a MathML file.

use the Table menu to insert previously saved spreadsheets; the menu item Table ▶ Insert will bring up a directory browser that use to select a file to enter.

2.6.3 Saving a program

You can open up a program editor (see Section 25.1.1, p. 647) with the menu item Prg ▶ New program or with Alt+P. If you select this item, you will be prompted for information to fill out a template for a program and then be left in the program editor.

At the top of the program editor are menus and buttons, at the far right will be a Save button that you can press to save the program. The first time you save a program, you will be prompted for a file name; you should choose a name ending in `.cxx`. Once a program is saved, the file name will appear to the right of the Save button. If you want to save the program under a different name, use the Prog ▶ Save as item from the program editor menu.

To insert a previously saved program, use the Prog ▶ Load item from the program editor menu.

2.6.4 Printing a session

You can print a session with the File ▶ Print ▶ To printer menu item.

If you prefer to save the printed form as a file, use the File ▶ Print ▶ Preview menu item. You will prompted for a file name to save the printed form in; the file will be a PostScript file, so the name should end in `.ps`. If you only want to save certain levels in printable form, use the File ▶ Print ▶ Preview selected levels menu item; this file will be encapsulated PostScript, so the name should end in `.eps`.

2.7 Translating to other computer languages

Xcas can export a session, or parts of a session, to typesetting languages such as LaTeX and MathML.

2.7.1 Translating an expression to LaTeX

The `latex` command translates expressions to LaTeX.

- `latex` takes *expr*, an expression.
- `latex(`*expr*`)` returns the result of evaluating *expr* written in the LaTeX typesetting language. Note that the returned value is a string which typesets properly when inserted in a (displayed) math environment.
- When copying the `latex` command output manually from a `giac` interface to a LaTeX document, be sure to remove the double quotes.

Example

```
> latex(1+1/2)
```

```
\frac{3}{2}
```

2.7.2 Translating the entire session to LaTeX

To export entire session to a LaTeX file, use the menu item File ▶ LaTeX ▶ LaTeX preview.

2.7.3 Translating graphical output to LaTeX

You can see all of your graphic output at once on the DispG screen, which you can bring up with the command DispG(). (This screen can be cleared with the command line command erase().) On the DispG screen there will be a Print menu; the Print ▶ LaTeX print will give you several files DispG.tex, DispG.dvi, DispG.ps and DispG.png with the graphics in different formats. To save it without using the DispG() command use the graph2tex command.

The graph2tex command saves all current graphic output to a LaTeX file.

- graph2tex takes *filename*.tex, the name of a file.
- graph2tex("*filename*.tex") saves all graphic output in LaTeX form to the file *filename*.tex.

Example

The command line

```
> graph2tex("myfile.tex")
```

creates a LaTeX file myfile.tex with the graphs. To save a 3D graph, use the command graph3d2tex.

To save a single graph as a LaTeX file, use the M menu to the right of the graph. Selecting M ▶ Export Print ▶ Print (with LaTeX) will save the current graph. You can also save a single graph by selecting that level, then use the menu item File ▶ LaTeX ▶ LaTeX print selection. This method will save the graph in several formats; *sessionname*.tex, *sessionname*.dvi, *sessionname*.ps and *sessionname*.png. If the session has not been saved and named, the files will begin with session*n* for some integer *n*.

2.7.4 Translating an expression to MathML

The mathml command translates expressions to MathML.

- mathml takes *expr*, an expression.
- mathml(*expr*) returns the result of evaluating *expr* written in MathML.

Remark. mathml is a legacy export command; a more complete translation to MathML is provided by the export_mathml command (see Section 2.7.7, p. 23). The latter is useful if you need quality rendering of giac/Xcas output for your web pages.

Example

```
> mathml(1/4+1/4)
```

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 plus MathML 2.0//EN"
"http://www.w3.org/TR/MathML2/dtd/xhtml-math11-f.dtd" [
<!ENTITY mathml "http://www.w3.org/1998/Math/MathML">
]>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<math mode="display" xmlns="http://www.w3.org/1998/Math/MathML">

<mfrac><mrow><mn>1</mn></mrow><mrow><mn>2</mn></mrow></mfrac>

</math><br/>
```

```
</body> </html>
```

This is the number 1/2 in MATHML form, along with enough information to make it a complete HTML document.

2.7.5 Translating a spreadsheet to MathML

You can translate an entire spreadsheet to MATHML with the spreadsheet menu command Table ▶ Save as MathML.

2.7.6 Indent an XML string

The `xml_print` command formats an XML string.

- `xml_print` takes *str*, a string, assumed to contain XML.
- `xml_print(`*str*`)` returns a string with the XML code indented for better readability. The default indentation is two spaces.

Example

```
> xml_print("<?xml version='1.0'?><r><c1>bla</c1><c2></c2><c3/></r>")
```

```
<?xml version='1.0'?>
<r>
  <c1>bla</c1>
  <c2></c2>
  <c3/>
</r>
```

2.7.7 Export to presentation or content MathML

You can translate the result of an expression into various types of MATHML with the `export_mathml` command.

- `export_mathml` takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally, *format*, which can be `content` or `display`, specifying the output format.
- `export_mathml(`*expr*$\langle$, *format*$\rangle$`)` returns the result of evaluating *expr* written in MATHML, with a single `math` block which will be a `semantics` block. Choices for the second argument are:
  - None: the `semantics` block will contain both presentation and content MATHML.
  - `content`: the `semantics` block will contain only the content MATHML.
  - `display`: the `semantics` block will contain only the presentation MATHML.

Examples

```
> xml_print(export_mathml(a+2*b))
```

```
<math mode='display' xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow xref='id5'>
      <mi xref='id1'>a</mi>
      <mo>+</mo>
      <mrow xref='id4'>
        <mn xref='id2'>2</mn>
        <mo>&it;</mo>
        <mi xref='id3'>b</mi>
      </mrow>
    </mrow>
    <annotation-xml encoding='MathML-Content'>
      <apply id='id5'>
        <plus/>
        <ci id='id1'>a</ci>
        <apply id='id4'>
          <times/>
          <cn id='id2' type='integer'>2</cn>
          <ci id='id3'>b</ci>
        </apply>
      </apply>
    </annotation-xml>
    <annotation encoding='giac'>a+2*b</annotation>
  </semantics>
</math>
```

```
> xml_print(export_mathml(a+2*b,content))
```

```
<math mode='display' xmlns='http://www.w3.org/1998/Math/MathML'>
  <apply id='id5'>
    <plus/>
    <ci id='id1'>a</ci>
    <apply id='id4'>
      <times/>
      <cn id='id2' type='integer'>2</cn>
      <ci id='id3'>b</ci>
    </apply>
  </apply>
</math>
```

```
> xml_print(export_mathml(a+2*b,display))
```

```
<math mode='display' xmlns='http://www.w3.org/1998/Math/MathML'>
  <mrow>
    <mi>a</mi>
    <mo>+</mo>
    <mrow>
      <mn>2</mn>
      <mo>&it;</mo>
      <mi>b</mi>
    </mrow>
  </mrow>
```

```
</math>
```

```
> s:=export_mathml(1/(x^2+1),display):;
xml_print(s)
```

```
<math mode='display' xmlns='http://www.w3.org/1998/Math/MathML'>
  <mfrac>
    <mn>1</mn>
    <mrow>
      <msup>
        <mi>x</mi>
        <mn>2</mn>
      </msup>
      <mo>+</mo>
      <mn>1</mn>
    </mrow>
  </mfrac>
</math>
```

2.7.8 Configuring markup export

Export to LaTeX, MathML and TeXmacs Scheme has some configuration options that can be set by using the `markup_cfg` command. This applies to `latex` and `export_mathml` commands as well as to TeXmacs sessions (note that `mathml` command does not support this configuration).

- `markup_cfg` takes a sequence of one or more arguments, each of which being one of:
  - `simplify=[`*simp*`]`, where *simp* is a sequence containing one or more of the following: `product`, `apply`, or `pow`. If *simp* has only one element, square brackets may be omitted.
  - `mathml=[`*comp*`]`, where `comp` is a sequence of one or more of the following: `display` or `content`. If *comp* has only one element, square brackets may be omitted.
- `markup_cfg` returns 1 if the configuration is successful, else it returns 0.
- The `simplify` argument refers to enabling/disabling automatic simplification of mathematical notation. Precisely:
  - Multiplication is made implicit where possible, hence the export commands favor e.g. $2x$ instead of $2 \cdot x$. If *simp* contains `product`, then implicit multiplication is allowed, else it is not allowed (in that case the $\cdot$ symbol is always inserted).
  - Parentheses are omitted from function application when possible, i.e. when the argument is sufficiently simple. Therefore the export commands favor e.g. $\sin x$ instead of $\sin(x)$. If *simp* contains `apply`, then parentheses may be omitted when appropriate, otherwise they are always present.
  - Elementary function powers are simplified according to the traditional way of writing where the exponent moves between the function name and its argument; export commands therefore favor e.g. $\sin^2 x$ instead of $\sin(x)^2$. If *simp* contains `pow`, then exponents may be moved, else this feature is disabled (by default it is eanbled).
- The `mathml` argument specifies which MathML components should be included in the output by default. If the sequence *comp* contains `display`, the presentation markup block will be present, and if *comp* contains `content`, then the content block will be present in the output. If one of these is not specified, the corresponding block will not appear in the MathML output. Note that you have to specify at least one component (by default, both are enabled).

Examples

With default configuration:

```
> latex(x*sin(x)^2)
```

```
x \sin ^{2}x
```

which is $x\sin^2 x$ when typeset.

To enable only implicit multiplication, enter:

```
> markup_cfg(simplify=product)
```

```
1
```

Then:

```
> latex(x*sin(x)^2)
```

```
x \sin \left(x\right)^{2}
```

which is $x\sin(x)^2$ when typeset.

The command

```
> markup_cfg(simplify=[product,apply],mathml=display)
```

enables only implicit multiplication and function application; additionally, it prevents content MathML from appearing in the output of `export_mathml`.

2.7.9 Translating a Maple file to Xcas

The `maple2xcas` command translates a file of Maple commands to the Xcas language.

- `maple2xcas` takes two arguments:
  - *maplefile*, the name of the Maple input file.
  - *xcasfile*, the file where you want to save the Xcas commands.
- `maple2xcas("`*maplefile*`","`*xcasfile*`")` results in an Xcas file named *xcasfile* with the Maple commands from *maplefile* translated to the Xcas language.

2.8 Entry in Xcas

If you enter a command into Xcas and press Enter, the result will appear in the output box below the input, colored in blue. Any messages generated during the computation will be shown in the box between the input and output, colored in green (the box is otherwise hidden). For example:

```
> integrate(a/(x+a)^2,x=0..inf)
```

```
No checks were made for singular points of antiderivative -1/(x+a) for definite integration in [0,+infinity]
```

```
1
```

Note that line breaks in a command entry can be entered by pressing Shift+Enter.

2.8.1 Suppressing output

The `nodisp` command is used to evaluate an expression and suppress the output.

- `nodisp` takes *expr*, an expression.
- `nodisp(`*expr*`)` evaluates *expr* but displays `Done` in place of the result.

An alternate way of suppressing the output is to end the input with the `:;` symbol.

Example

```
> nodisp(a:=2+2)
```

Done

and a will be set to 4.

```
> b:=3+3:;
```

Done

and b will be set to 6.

2.8.2 Entering comments

You can annotate an Xcas session by adding comments. You can enter a comment on the current line at any time by typing Alt+C. The line will appear in green text and conclude when you type Enter. (To force a new line, press Shift+Enter.) Comments are not evaluated and so have no output. If you have started entering a command when you begin a comment, the command line with the start of the command will be pushed down so that you can finish it when you complete the comment.

You can open the browser using a comment line by entering the web address beginning with the @ sign. For example, if you enter the comment line

```
The Xcas homepage is at
@www-fourier.ujf-grenoble.fr/~parisse/giac.html
```

then the browser will open to the Xcas home page.

To add a comment to a program, rather than a session, use the comment command.

- comment takes *str*, a string.
- comment(*str*) makes *str* a comment.

Alternatively, any part of a program between // and the end of the line is a comment. So both

```
bs():={
  comment("Hello");
  return "Hi there!";
}
```

and

```
bs():={ // Hello
  return "Hi there!";
}
```

are programs with the comment "Hello".

2.8.3 Previous outputs

The ans command returns the outputs of the previous commands.

- ans takes one optional argument:

  Optionally, n, an integer (the 0-based index of the command, by default $n = -1$).
- ans(⟨n⟩) returns the corresponding result; in particular, ans(-1) returns the previous result.

Example

If the first command that you enter is:

```
> 2+5
```

7

then later references to `ans(0)` will evaluate to 7.

Note that the argument to `ans` does not correspond to the line number in XCAS. For one thing, the line numbers begin at 1. What's more, if you go back and re-evaluate a previous line, then that will become part of the commands that `ans` keeps track of.

If you give `ans` a negative number, then it counts backwards from the current input. To get the latest output, for example, you can use `ans(-1)`. With no argument, `ans()` will also return the latest output.

Similarly, the `quest` command returns the previous inputs. Since these will often be simplified to be the same as the output, `quest(`n`)` sometimes has the same value as `ans(`n`)`.

You can also use Ctrl plus the arrow keys to scroll through previous inputs. With the cursor on the command line, Ctrl+Up will go backwards in the list of previous commands and display them on the current line, and Ctrl+Down will go forwards.

2.9 Editing expressions

You can enter expressions on the command line, but XCAS also has a built-in expression editor that use to enter expressions in two dimensions, the way they normally look when typeset. When you have an expression in the editor, you can also manipulate subexpressions apart from the entire expression.

2.9.1 Entering expressions in the editor: an example

The expression

$$\frac{x+2}{x^2-4}$$

can be entered on the command line as:

```
> (x+2)/(x^2-4)
```

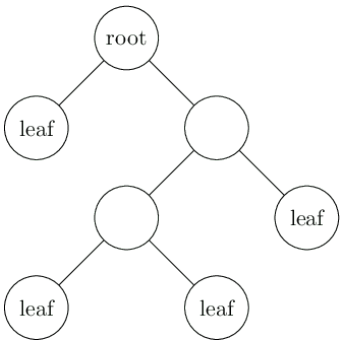
You also can use the expression editor to enter it visually, as $x+2$ on top of x^2-4. To do this, you can start the expression editor with the Alt+E keystroke (or the Expression ▶ New Expression menu command). There will be a small button M on the right side of the expression line, which is a menu with some commands use on the expressions. There will also be a `0` selected on the expression line and an on-screen keyboard at the bottom (see Section 2.2, p. 5, item 2.2). If you type `x + 2`, it will overwrite the `0`. To make this the top of the fraction, you can select it with the mouse (you can also make selections with the keyboard, as will be discussed later) and then type `/`. This will leave the `x + 2` on the top of a horizontal fraction bar and the cursor on the bottom. To enter x^2-4 on the bottom, begin by typing `x`. Selecting this `x` and typing `^2` will put on the superscript. Finally, selecting the x^2 and typing `- 4` will finish the bottom. If you then hit Enter, the expression will be evaluated and will appear on the output line.

2.9.2 Subexpressions

XCAS can operate on expressions in the expression editor or subexpressions of the expression. To understand subexpressions and how to select its parts, it helps to know that XCAS stores expressions as *trees*.

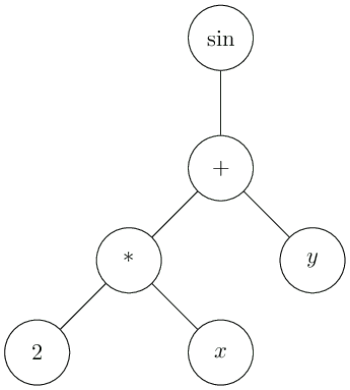
A tree, in this sense, consists of objects called nodes. A node can be connected to lower nodes, called the children of the node. Each node (except one) will be connected to exactly one node above it, called

the parent node. One special node, called the root node, won't have a parent node. Two nodes with the same parent nodes are called siblings. Finally, if a node does not have any children, it is called a leaf. This terminology comes from a visual representation of a tree,



which looks like an upside-down tree; the root is at the top and the leaves are at the bottom.

Given an expression, the nodes of the corresponding tree are the functions, operators, variables and constants. The children of a function node are its arguments, the children of an operator node are its operands, and the constants and variables will be the leaves. For example, the tree for $\sin(2x + y)$ will look like



A subexpression of an expression will be a selected node together with the nodes below it. For example, both $2x$ and $2x + y$ are subexpressions of $\sin(2x + y)$, but $x + y$ is not.

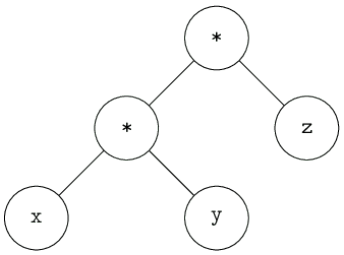
A subexpression of the contents of the expression editor can be selected with the mouse; the selection will appear white on a black background. A subexpression can also be chosen with the keyboard using the arrow keys. Given a selection:

- The up arrow will go to the parent node.
- The down arrow will go to the leftmost child node.
- The right and left arrows will go to the right and left sibling nodes.
- The control key with the right and left arrows will switch the selection with the corresponding sibling.
- If a constant or variable is selected, the backspace key will delete it. For other selections, backspace will delete the function or operator, and another backspace will delete the arguments or operands.

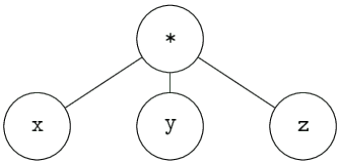
Use the arrow keys to navigate the tree structure of an expression, which is not always evident by looking at the expression itself. For example, suppose you enter `x*y*z` in the editor. The two multiplications will be a different levels; the tree will look like

Key	Action on selection	Key	Action on selection
Ctrl+D	differentiate	Ctrl+P	partial fraction
Ctrl+F	factor	Ctrl+R	integrate
Ctrl+L	limit	Ctrl+S	simplify
Ctrl+N	normalize	Ctrl+T	copy LaTeX version to clipboard

Table 2.1: Keystroke operations on subexpressions



If you select the entire expression with the up arrow and then go to the M menu to the right of the line and choose eval, then the expression will look the same but, as you can check by navigating it with the arrow keys, the tree will look like



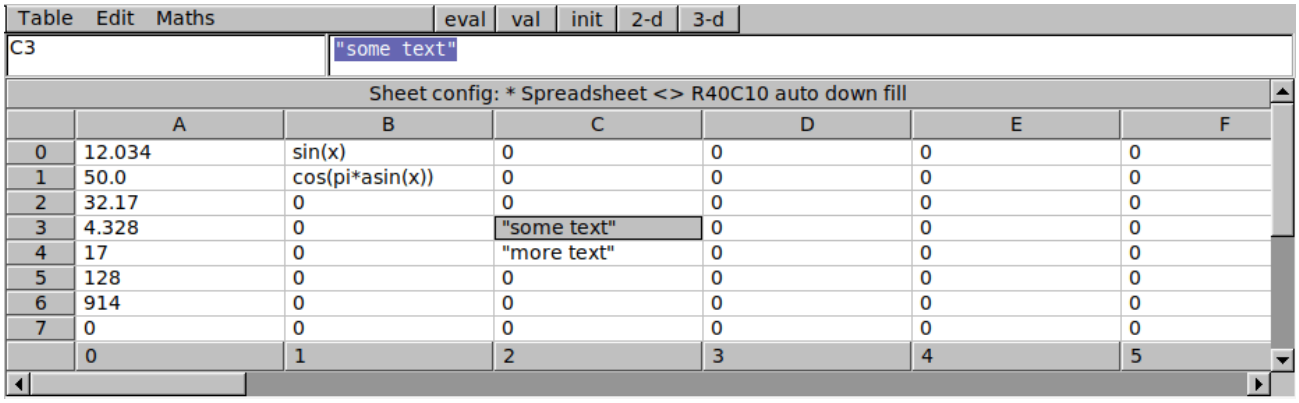
2.9.3 Manipulating subexpressions

If a subexpression is selected in the expression editor, then any menu command will be applied to that subexpression.

For example, enter the expression `(x+1)*(x+2)*(x-1)` in the expression editor. Note that use the abilities of the editor to make this easier. First, enter `x+1`. Select this with the up arrow, then type `*` followed by `x+2`. Select the `x+2` with the up arrow and then type `*` followed by `x-1`. Using the up arrow again will select the `x-1`. Select the entire expression with the up arrow, and then select `eval` from the M menu. This will put all factors at the same level. Suppose you want the factors `(x+1)*(x+2)` to be expanded. You could select `(x+1)*(x+2)` with the mouse and do one of the following:

- Select the Expression ▶ Misc ▶ normal menu item. You will then have `normal((x+1)*(x+2))*(x-1)` in the editor. If you hit enter, the result $(x^2+3x+2)(x-1)$ will appear in the output window.
- Select the Expression ▶ Misc ▶ normal menu item, so again you have `normal((x+1)*(x+2))*(x-1)` in the editor. Now if you select `eval` from the M menu, then the expression in the editor will become the result $(x^2+3x+2)(x-1)$, which you can continue editing.
- Choose normal from the M menu. This will apply normal to the selection, and again you will have the result $(x^2+3x+2)(x-1)$ in the editor.

There are also keystroke commands that use to operate on subexpressions that you have selected. There are the usual Ctrl+Z and Ctrl+Y for undoing and redoing. Some of the others are given in Table 2.1, p. 30.



	A	B	C	D	E	F
0	12.034	sin(x)	0	0	0	0
1	50.0	cos(pi*asin(x))	0	0	0	0
2	32.17	0	0	0	0	0
3	4.328	0	"some text"	0	0	0
4	17	0	"more text"	0	0	0
5	128	0	0	0	0	0
6	914	0	0	0	0	0
7	0	0	0	0	0	0
	0	1	2	3	4	5

Figure 2.4: A spreadsheet in Xcas

2.10 Spreadsheets

2.10.1 Opening a spreadsheet

You can open a spreadsheet with the Spreadsheet ▶ New Spreadsheet menu item or with the key Alt+T.
When you open a new spreadsheet, a configuration screen with the following options appears:

Variable — Here you can specify a variable in which the spreadsheet will be saved as a matrix.

Rows, Columns — Here you can specify the number of rows and columns in the spreadsheet.

Eval — If checked, then the spreadsheet will be re-evaluated every time you make a change to it. (You can always re-evaluate the spreadsheet by clicking the eval button on its menu bar.)

Distribute — If checked, then entering a matrix will distribute the contents across an appropriate array of cells, otherwise the matrix will be put in one cell.

Landscape — If checked, the graphical representation of the spreadsheet will be displayed below the spreadsheet, otherwise it will be displayed to the right of the spreadsheet.

Move right — If checked, the cursor will move to the cell to the right of the current cell when data is entered, otherwise it will be moved to the cell below the current cell.

Spreadsheet — If checked, the spreadsheet will be formatted as a spreadsheet, otherwise it will be formatted as a matrix.

Graph — If checked, the graphical representation of the spreadsheet will be displayed.

Undo history — Here you can specify how many undo's can be performed at a time.

The configuration screen can be reopened with the Edit ▶ Configuration ▶ Cfg window menu attached to the spreadsheet.

2.10.2 Spreadsheet window

When you open a spreadsheet, the input line will become the spreadsheet, like the one shown in Figure 2.4. The top will be a menu bar with Table, Edit and Maths menus as well as eval, val, init, 2D and 3D buttons. To the right will be the name of the file the spreadsheet will be saved into. Below the menu bar will be two boxes; a box which displays the active cell (and can be used to choose a cell) and a command line to enter information into the cell. Below that will be a status line, which can be clicked to return to the configuration screen.

3 CAS building blocks

3.1 Constants

3.1.1 Numbers

Xcas works with both real and complex numbers. The real numbers can be integers, rational numbers, floating point numbers or symbolic constants.

You can enter an integer by simply typing the digits.

```
> 1234321
```

$$1234321$$

Alternatively, you can enter an integer in binary (base 2) by prefixing the digits (0 through 1) with `0b`, in octal (base 8) by prefixing the digits (0 through 7) with `0` or `0o`, and in hexadecimal (base 16) by prefixing the digits (0 through 9 and a through f) with `0x`. (See Section 5.4.1, p. 65.)

```
> 0xab12
```

$$43794$$

You can enter a rational number as the ratio of two integers.

```
> 123/45
```

$$\frac{41}{15}$$

The result will be put in lowest terms. If the top is a multiple of the bottom, the result will be an integer.

```
> 123/3
```

$$41$$

A floating point number is regarded as an approximation to a real number. You can enter a floating point number by writing it out with a decimal point.

```
> 123.45
```

$$123.45$$

You can also enter a floating point number by entering a sequence of digits, with an optional decimal point, followed by `e` and then an integer, where the `e` represents "times 10 to the following power."

```
> 1234e3
```

$$1234000.0$$

Floating point numbers with a large number of digits will be printed with `e` notation; you can control how other floats are displayed (see Section 2.5.7, p. 15, item 2.5.7). An integer or rational number can be converted to a floating point number with `evalf` (see Section 7.3.1, p. 128).

A complex number is a number of the form a+b`i`, where a and b are real numbers. The numbers a and b will be the same type of real number; one type will be converted to the other type if necessary

Symbol	Value
`e` (or `%e`)	the number e^1
`pi` (or `%pi`)	the number π
`infinity`	unsigned ∞
`+infinity` (or `inf`)	$+\infty$
`-infinity` (or `-inf`)	$-\infty$
`i` (or `%i`)	the complex number $0 + 1 \cdot \mathrm{i}$
`euler_gamma`	Euler's constant $\gamma := \lim_{n\to\infty} \left(\sum_{k=1}^{n} \frac{1}{k} - \ln(n)\right)$

Table 3.1: Symbolic constants in Xcas

(an integer can be converted to a rational number or a floating point number, and a rational number can be converted to a floating point number).

```
> 3+1.1i
```

$$3 + 1.1\mathrm{i}$$

3.1.2 Symbolic constants

Standard constants in Xcas are built-in symbols, given in Table 3.1, p. 33.

Since these numbers cannot be written exactly as standard decimal numbers, they are necessarily left unevaluated in exact results (see Section 2.5.4, p. 14).

```
> 2*pi
```

$$2\pi$$

```
> 2.0*pi
```

$$6.28318530718$$

You can also use `evalf` (see Section 7.3.1, p. 128), for example, to approximate one of the real-valued constants to as many decimal places as you want.

```
> evalf(pi,50)
```

$$3.1415926535897932384626433832795028841971693993751$$

3.1.3 Testing for undefined and infinity symbols

The `isinf` command checks whether its argument is an infinity (unsigned, positive or negative) or not.

The `isnan` command checks whether its argument is undefined or not.

Examples

```
> isnan(0/0)
```

true

```
> isnan(0/1)
```

false

```
> isinf(0/1)
```

false

```
> isinf(1/0)
```

true

3.2 Sequences, sets and lists

3.2.1 Sequences

A sequence is represented by a sequence of elements separated by commas, without delimiters or with either parentheses (`(` and `)`) or `seq[` and `]` as delimiters, as in:

```
> 1,2,3,4
```

or:

```
> (1,2,3,4)
```

or:

```
> seq[1,2,3,4]
```

$1, 2, 3, 4$

Note that the order of the elements of a sequence is significant. For example, if `B:=(5,6,3,4)` and `C:=(3,4,5,6)`, then `B==C` returns `false`.

(A value can be assigned to a variable with the `:=` operator; see Section 3.3.1, p. 36. Also, `==` is the test for equality; see Section 5.1.2, p. 50.)

Note also that the expressions `seq[...]` and `seq(...)` are not the same (see Section 6.1.2, p. 67 for information on `seq(...)`). For example, `seq([0,2])=(0,0)` and `seq([0,1,1,5])=[0,0,0,0,0]` but `seq[0,2]=(0,2)` and `seq[0,1,1,5]=(0,1,1,5)`

See Section 6.1, p. 67 for operations on sequences.

3.2.2 Sets

To define a set of elements, put the elements separated by commas, with delimiters `%{` and `%}` or `set[` and `]`.

```
> set[1,2,3,4]
```

or:

```
> %{1,2,3,4%}
```

$[\![1, 2, 3, 4]\!]$

In the XCAS output, the set delimiters are displayed as $[\![$ and $]\!]$ in order not to confuse sets with lists (see Section 3.2.3, p. 35). For example, $[\![1, 2, 3]\!]$ is the set `%{1,2,3%}`, unlike $[1, 2, 3]$ (normal brackets) which is the list `[1,2,3]`.

```
> A:=%{1,2,3,4%}
```

or:

```
> A:=set[1,2,3,4]
```

$[\![1, 2, 3, 4]\!]$

```
> B:=%{5,5,6,3,4%}
```

or:

```
> B:=set[5,5,6,3,4]
```

$$[\![5,6,3,4]\!]$$

Remark. The order in a set is not significant and the elements in a set are all distinct. If you input `B:=%{5,5,6,3,4%}` and `C:=%{3,4,5,3,6%}`, then `B==C` will return `true`.

See Section 6.4, p. 96 for operations on sets.

3.2.3 Lists

A list is delimited by `[` and `]`, its elements must be separated by commas. For example, `[1,2,5]` is a list of three integers. Lists are also called vectors in Xcas.

Lists can contain lists (for example, a matrix is a list of lists of the same size, see Section 14.2, p. 325). Lists may be used to represent vectors (lists of coordinates), matrices, or univariate polynomials (lists of coefficients by decreasing order, see Section 11.1.1, p. 211).

Lists are different from sequences, because sequences are flat: an element of a sequence cannot be a sequence. Lists are different from sets, because for a list, the order is important and the same element can be repeated in a list (unlike in a set where each element is unique). See Section 6.3, p. 78 for operations on lists.

In Xcas output:

- matrix and list delimiters are displayed as square brackets ([and]),
- polynomial delimiters are displayed as ⟦, ⟧
- set delimiters are displayed as ⟦, ⟧.

3.2.4 Accessing elements

The elements of sequences and lists are indexed starting from 0 in Xcas syntax mode and from 1 in all other syntax modes (see Section 2.5.2, p. 14). To access an element of a list or a sequence, follow the list with the index between square brackets.

Examples

```
> L:=[2,5,1,4]
```

$$[2,5,1,4]$$

```
> L[1]
```

$$5$$

To access the last element of a list or sequence, you can put `-1` between square brackets.

```
> L[-1]
```

$$4$$

If you want the indices to start from 1 in Xcas syntax mode, you can enter the index between double brackets.

```
> L[[1]]
```

$$2$$

3.3 Variables

3.3.1 Variable names

A variable or function name is a sequence of letters, numbers and underscores that begins with a letter. If you define your own variable or function, you cannot use the names of built-in variables or functions or other keywords reserved by XCAS.

3.3.2 Assigning values to variables

You can assign a value to a variable with the `:=` operator. For example, to give the variable `a` the value of `4`, enter:

```
> a:=4
```

Alternatively, use the `=>` operator; when you use this operator, the value comes before the variable:

```
> 4=>a
```

The function `sto` (or `Store`) can also be used; again, the value comes before the variable (the value is stored into the variable):

```
> sto(4,a)
```

After any one of these commands, whenever you use the variable `a` in an expression, it will be replaced by `4`.

use sequences or lists to make multiple assignments at the same time. For example,

```
> (a,b,c):=(1,2,3)
```

will assign `a` the value `1`, `b` the value `2` and `c` the value `3`. Note that this can be used to swap the values of two variables; with `a` and `b` as above, the command

```
> (a,b):=(b,a)
```

will set `a` equal to `b`'s original value, namely `2`, and will set `b` equal to `a`'s original value, namely `1`.

Another way to assign values to variables, useful in MAPLE mode, is with the `assign` command. If you enter

```
> assign(a,3)
```

or

```
> assign(a=3)
```

then `a` will have the value `3`. You can assign multiple values at once; if you enter

```
> assign([a=1,b=2])
```

then `a` will have the value `1` and `b` will have the value `2`. This command can be useful in MAPLE mode, where solutions of equations are returned as equations. For example, if you enter (in MAPLE mode):

```
> sol:=solve([x+y=1,y=2],[x,y])
```

(see Section 9.3.6, p. 184), you will get

$$[x = -1, y = 2]$$

If you then enter

```
> assign(sol)
```

the variable `x` will have value `-1` and `y` will have the value `2`. This same effect can be achieved in standard XCAS mode, where

```
> sol:=solve([x+y=1,y=2],[x,y])
```

will return

$$[[-1, 2]]$$

In this case, the command

```
> [x,y]:=sol[0]
```

will assign `x` the value `-1` and `y` the value `2`.

3.3.3 Assignment by reference

A list is simply a sequence of values separated by commas and delimited by `[` and `]` (see Section 6.1, p. 67). Suppose you give the variable `a` the value `[1,1,3,4,5]`:

```
> a:=[1,1,3,4,5]
```

If you later assign to `a` the value `[1,2,3,4,5]`, then a new list is created. It may be better to just change the second value in the original list by reference. This can be done with the `=<` command. Recalling that lists are indexed beginning at 0, the command

```
> a[1]=<2
```

will simply change the value of the second element of the list instead of creating a new list, and is a more efficient way to change the value of `a` to `[1,2,3,4,5]`.

3.3.4 Copying lists

If you enter

```
> list1:=[1,2,3]
```

and then

```
> list2:=list1
```

then `list1` and `list2` will be equal to the same list, not simply two lists with the same elements. In particular, if you change (by reference) the value of an element of `list1`, then the change will also be reflected in `list2`. For example, if you enter

```
> list1[1]=<5
```

then both `list1` and `list2` will be equal to `[1,5,3]`.

The `copy` command creates a copy of a list (or vector or matrix) which is equal to the original list, but distinct from it. For example, if you enter

```
> list1:=[1,2,3]
```

and then

```
> list2:=copy(list1)
```

then `list1` and `list2` will both be `[1,2,3]`, but now if you enter

```
> list1[1]=<5
```

then `list1` will be equal to `[1,5,3]` but `list2` will still be `[1,2,3]`.

3.3.5 Incrementing variables

You can increase the value of a variable `a` by `4`, for example, with

```
> a:=a+4
```

If beforehand `a` were equal to `4`, it would now be equal to `8`. A shorthand way of doing this is with the `+=` operator;

```
> a+=4
```

will also increase the value of `a` by `4`.

Similar shorthands exist for subtraction, multiplication and division. If `a` is equal to `8` and you enter

```
> a-=2
```

then `a` will be equal to `6`. If you follow this with

```
> a*=3
```

then `a` will be equal to `18`, and finally

```
> a/=9
```

will end with `a` equal to `2`.

3.3.6 Storing and recalling variables and their values

The `archive` command stores the values of variables for later use in a file of your choosing.

- `archive` takes two arguments:
  - *filename*, a filename in which to store values.
  - *vars*, a variable or list of variables.
- `archive("`*filename*`",`*vars*`)` saves the values of *vars* (or the values of the variables in the list) in file *filename*.

For example, if the variable `a` has the value `2` and the variable `bee` has the value `"letter"` (a string), then entering

```
> archive("foo",[a,bee])
```

will create a file named "`foo`" which contains the values `2` and `"letter"` in a format meant to be efficiently read by XCAS.

The `unarchive` command will read the values from a file created with `archive`.

- `unarchive` takes *filename*, the filename.
- `unarchive("`*filename*`")` returns the value or list of values stored in *filename*.

Example

With the file `foo` as above:

```
> unarchive("foo")
```

[2, "letter"]

You can enter

```
> [a,bee]:=unarchive("foo")
```

in order to reassign these values to `a` and `bee`.

3.3.7 Copying variables

The `CopyVar` command copies the contents of one variable into another, without evaluating the contents.

- `CopyVar` takes two arguments:
  - *fromvar*, the name of a variable to copy from.
  - *tovar*, the name of a variable to copy to.
- `CopyVar(`*fromvar*, *tovar*`)` copies the unevaluated contents of *fromvar* into `tovar`.

Example

```
> a:=c:;
  c:=5:;
  CopyVar(a,b);
```

$$c$$

then:

```
> b
```

$$5$$

Changing the value if `c` will also change the output of `b`, since `b` contains `c`.

```
> c:=10:;
  b
```

$$10$$

3.3.8 Assumptions on variables

The `assume` (or `supposons`) command lets you tell Xcas some properties of a variable without giving the variable a specific value.

- `assume` (or `supposons`) takes one mandatory argument and one optional argument:
  - *assumptions*, statements about a variable (such as equalities and inequalities, possibly combined with `and` and `or`, and domains).
  - Optionally, `additionally`, which indicates that the assumptions are to be added to previous assumptions, as opposed to replace them.
- `assume(`*assumptions* ⟨`,additionally`⟩`)` places the assumptions on the variable. With no second argument, it will remove any previous assumptions.

The `additionally` command can be used to place additional assumptions on a variable.

- `additionally` takes *assumptions* as above.
- `additionally(`*assumptions*`)` adds the assumptions to a variable without removing assumptions.

The `about` command will display the current assumptions about a variable. (See Section 28.1.8, p. 819 and Section 28.2.6, p. 825 for other uses of `about`.)

- `about` takes *var*, the name of a variable.
- `about(`*var*`)` returns the current assumptions on the variable.

The `purge` command will remove all values and assumptions about a variable.

- `purge` takes *var*, a variable name or a sequence of variable names.
- `purge(`*var*`)` removes any assumptions you have made about the variable *var* (or about all the variables in the sequence).

Examples

If you enter

```
> assume(variable>0)
```

then XCAS will assume that `variable` is a positive real number, and so

```
> abs(variable)
```

will be evaluated to

variable

You can put one or more conditions in the `assume` command by combining them with `and` and `or`. For example, if you want the variable `a` to be in $[2, 4) \cup (6, \infty)$, you can enter

```
> assume((a>=2 and a<4) or a>6)
```

If a variable has attached assumptions, then making another assumption with `assume` will remove the original assumptions. To add extra assumptions, you can either use the `additionally` command or give `assume` a second argument of `additionally`. If you assume that $b > 0$ with

```
> assume(b>0)
```

and you want to add the condition that $b < 1$, you can either enter

```
> assume(b<1,additionally)
```

or

```
> additionally(b<1)
```

As well as equalities and inequalities, you can make assumptions about the domain of a variable. If you want `n` to represent an integer, for example, you can enter

```
> assume(n,integer)
```

If you want `n` to be a positive integer, you can add the condition

```
> additionally(n>0)
```

You can also assume a variable is in one of the domains `real`, `integer`, `complex` or `rational` (see Section 25.2.5, p. 651).

You can check the assumptions on a variable with the `about` command. For the above positive integer `n`,

```
> about(n)
```

[integer, [[0, +∞]], [0]]

The first element tells you that `n` is an integer, the second element tells you that `n` is between 0 and $+\infty$, and the third element tells you that the value 0 is excluded.

If you assume that a variable is equal to a specific value, such as

```
> assume(c=2)
```

then by default the variable `c` will remain unevaluated in later levels. If you want an expression involving `c` to be evaluated, you would need to put the expression inside the `evalf` command (see Section 7.3.1, p. 128). After the above assumption on `c`, if you enter

```
> evalf(c^2+3)
```

then you will get

7.0

Right below the `assume(c=2)` command line there will be a slider; namely arrows pointing left and right with the value `2` between them. These can be used to change the values of `c`. If you click on the right arrow, the `assume(c=2)` command will transform to

```
> assume(c=[2.2,-10.0,10.0,0.0])
```

and the value between the arrows will be `2.2`. Also, any later levels where the variable `c` is evaluated will be re-evaluated with the value of `c` now `2.2`. The output to

```
> evalf(c^2+3)
```

will become

7.84

The `-10.0` and `10.0` in the `assume` line represent the smallest and largest values that `c` can become using the sliders. You can set them yourself in the `assume` command, as well as the increment that the value will change; if you want `c` to start with the value `5` and vary between `2` and `8` in increments of `0.05`, then you can enter

```
> assume(c=[5,2,8,0.05])
```

Recall the `purge` command removes assumptions about a variable.

```
> purge(a)
```

then `a` will no longer have any assumptions made about it.

```
> purge(a,b)
```

then `a` and `b` will no longer have any assumptions made about them.

3.3.9 Unassigning variables

XCAS has commands that help you keep track of what variables you are using and resetting them if desired. The `VARS` command will list all the variables that you are using, the `purge`, `DelVar` and `del` commands will delete selected variables, and the `rm_a_z` and `rm_all_vars` commands will remove classes of variables. All variables can be removed by using `restart`.

- `VARS` takes no arguments.
- `VARS()` returns a list of the variables that you have assigned values or made assumptions on.

The `purge` command will clear the values and assumptions you make on variables (see Section 3.3.8, p. 39). For TI compatibility there is also `DelVar`, and for PYTHON compatibility there is `del`.

- The `purge` command takes *var*, the name of a variable.
- `purge(`*var*`)` clears the variable *var* of all values and assumptions.
- The `DelVar` (and `del`) commands take one argument: *var*, the name of a variable.
- `Delvar` *var* (or `del` *var*) removes the values attached to *var*. (Note that they do not take their argument in parentheses.

The `rm_all_vars` and `restart` commands clear the values and assumptions you have made on all variables use.

- `rm_all_vars` takes no arguments.
- `rm_all_vars()` removes all the values that you have attached to variables.
- `restart` takes no arguments.
- `restart` removes all the values that you have attached to variables. (Note that it does not use parentheses.)

The rm_a_z command clears the values and assumptions on all variables with single lowercase letter names.

- rm_a_z takes no arguments.
- rm_a_z() purges all variables whose names are one letter and lowercase.

Examples

Enter:

```
> a:=1; anothervar:=2
```

then:

```
> VARS()
```

$$[a, \text{anothervar}]$$

To clear the variable a:

```
> purge(a)
```

or (for TI compatibility):

```
> DelVar a
```

or (for PYTHON compatibility):

```
> del a
```

If you have variables names A,B,a,b,myvar, then after:

```
> rm_a_z()
```

you will only have the variables named A,B,myvar.

3.3.10 CST variable

The menu available with the cust button in the bandeau on the onscreen keyboard (see Section 2.2, p. 5, item 2.2) is defined with the CST variable. It is a list where each list item determines a menu item; a list item is either a builtin command name or a list itself consisting of a string to be displayed in the menu and the input to be entered when the item is selected.

For example, to create a custom defined menu with the builtin function diff, a user defined function foo, and a menu item to insert the number 22/7, enter:

```
> CST:=[diff,["foo",foo],["My pi approx",22/7]]
```

Note that if the input to be entered is a variable and the variable has a value when CST is defined, then CST will contain the value of the variable. For example,

```
> app:=22/7:; CST:=[diff,["foo",foo],["My pi approx",app]]
```

will be equivalent to the previous definition of CST. However, if the variable does not have a value when CST is defined, for example:

```
> CST:=[diff,["foo",foo],["My pi approx",app]]:; app:=22/7
```

then CST will behave as the previous values to begin with, but in this case if the variable app is changed, the the result of pressing the My pi approx button will change also.

Since CST is a list, a function can be added to the cust menu by using the concat command (see Section 6.3.12, p. 84):

```
> CST:=concat(CST,evalc)
```

will add the evalc command to the cust menu.

3.4 Functions

3.4.1 Defining functions

Similar to how you can assign a value to a variable (see Section 3.3.2, p. 36), use the `:=` and `=>` operators to define a function; both

```
> f(x):=x^2
```

and

```
> x^2=>f(x)
```

give the name `f` to the function which takes a value and returns the square of the value. In either case, if you then enter:

```
> f(3)
```

you will get:

9

You can define an anonymous function, namely a function without a name, with the `->` operator; for example, the squaring function can be defined by

```
> x->x^2
```

use this form of the function to assign it a name; both

```
> f:=x->x^2
```

and

```
> x->x^2=>f
```

are alternate ways to define `f` as the squaring function.

You can similarly define functions of more than one variable. For example you could enter

```
> hypot(a,b):=sqrt(a^2+b^2)
```

or

```
> hypot:=(a,b)->sqrt(a^2+b^2)
```

to define a function which takes the lengths of the two legs of a right triangle and returns the hypotenuse.

4 Files and directories

4.1 Files

4.1.1 Writing variable values to a file

The `write` command saves variable values to a file, to be read later.

- `write` takes two arguments:
  - *filename*, a string.
  - *vars*, a sequence of variables.
- `write(`*filename*, *vars*`)` writes the variables in *vars* assigned to their values in a file named *filename*.

Example

```
> a:=3.14
  b:=7
  write("foo",a,b)
```

creates a file named "foo" containing:

```
a:=(3.14);
b:=7;
```

If you wanted to store the first million digits of π to a file, you could set it equal to a variable and store it in a file:

```
> pidec:=evalf(pi,10^6):;
  write("pi1million",pidec)
```

The file is written so that it can be loaded with the `read` command (Section 4.1.3, p. 45), which simply takes a file name as a string. This allows you to restore the values of variables saved this way, for example in a different session or if you have purged the variables.

Example

If, in a different session, you want to use the values of `a` and `b` above, enter:

```
> read("foo")
```

This will reassign the values `3.14` and `7` to `a` and `b`. Be careful, this will silently overwrite any values that `a` and `b` might have had.

4.1.2 Writing output to a file

Xcas has a basic file I/O support in form of three commands: `fopen`, `fprint` and `fclose`. Note that reading from files is not supported.

The `fopen` command creates and opens a file to write into.

- `fopen` takes *filename*, a string.
- `fopen(`*filename*`)` creates a file named *filename* (and erases it if it already exists).

To use this, you need to associate it with a variable *var*`:=fopen(`*filename*`)` which use to refer to the file when printing to it.

The `fprint` command writes to a file.

- `fprint` takes two mandatory arguments and one optional argument:
  - *var*, a variable name associated with a file through `fopen`.
  - Optionally, `Unquoted`, the symbol.
  - *info*, a list of what you want to write to the file.
- `fprint(`*var*⟨`,Unquote`⟩`,`*info*`)` writes *info* into the file given by *var*. By default, strings in *info* are written with their quotation marks, with the option `Unquoted`, `fprint` will print them with the quotation marks.

The `fclose` command closes a previously opened file.

- `fclose` takes *var*, a variable assigned to a file with `fopen`.
- `fclose(`*var*`)` closes the file given by *var* to further writing.

Example

To write contents to a file, you first need to open the file and associate it with a variable.

```
> f:=fopen("bar")
```

This creates a file named "bar" (and so erase it if it already exists). To write to the file:

```
> x:=9:; fprint(f,"x+1 is ",x+1)
```

This puts

```
"x+1 is "10
```

in the file. Note that the quotation marks are not inserted if you use the `Unquoted` argument:

```
> x:=9:; fprint(f,Unquoted,"x+1 is ",x+1)
```

This puts

```
x+1 is 10
```

in the file. Finally, after you have finished writing what you want into the file, you close the file with the `fclose` command:

```
> fclose(f)
```

This returns 1 on success and `0` on failure.

4.1.3 Reading files

Information for Xcas can be stored in a file; this information can be read with the `read` command, depending on the type of information.

The `read` command reads a file containing Xcas information, such as a program that you saved (see Section 2.6.3, p. 21) or simply commands that you typed into a file with a text editor. The file should have the suffix `.cxx`.

- `read` takes *filename*, the name of a file (a string) containing a saved program (see Section 2.6.3, p. 21) or other commands.
- `read(`*filename*`)` reads the content of the file.

Examples

If you have a file named `myfunction.cxx`,

```
> read("myfunction.cxx")
```

will read in the file, as long as the directory is in the current working directory. If the file is in a different directory, you can still read it by giving the path to the file:

```
> read("/path/to/file/myfunction.cxx")
```

In Windows, you should use `\\` instead of `/` as the directory separator. In Linux, absolute paths usually begin with `/home/`.

4.1.4 Reading CSV data

CSV data is textual data formatted into m lines separated by a character `linesep`, each of which contains n elements separated by a character `sep`. Usually, `linesep` is the newline character and `sep` is the comma or tab character.

The `csv2gen` command converts CSV data from files and strings to Xcas matrices.

- `csv2gen` takes one mandatory argument and up to five optional arguments:
  - *data*, a string containing either CSV data or the path to a CSV file.
  - Optionally, *sep*, a string specifying the character `sep`. Note that Xcas will use only the first character in *sep*. By default, *sep* = `";"`. (Note that the tab character is entered as `\t`.)
  - Optionally, *linesep*, a string specifying the character `linesep`. Note that Xcas will use only the first character in *linesep*. By default, *linesep* = `"\n"` (line feed).
  - Optionally, *decsep*, a string specifying the character used as the decimal point. Note that Xcas will use only the first character in *decsep*. By default, *decsep* = `","`.
  - Optionally, *eof*, a string or character specifying the end of CSV data. Note that Xcas will use only the first character in *eof* if it is a string. By default, *eof* = 0 (the EOF character).
  - Optionally, `string`, the symbol specifying that *data* is CSV data and not a file name. This argument may always be appended to the input sequence regardless of omitting some or all of the optional arguments above. By default, *data* is interpreted as the file name.
- `csv2gen(`*data*, ⟨, *sep* ⟨, *linesep* ⟨, *decsep* ⟨, *eof*⟩⟩⟩⟩ ⟨, `string` ⟩`)` returns a matrix with m rows and n columns containing the CSV data. Numbers and other `giac` expressions are automatically converted from strings.
- if *decsep* is given, then it is replaced by `"."` in the imported matrix prior to string→number conversion. If your data is using `"."` as the decimal point already (as is the Xcas standard) and you have specified *sep* = `","`, then you do not have to set *decsep* explicitly since there is no commas left after importing and attempting to replace them will have no effect.
- If *eof* is given, then importing will stop as soon as the specified character is read (it is subsequently discarded and therefore does not appear in the result).
- If there is some data missing in some line, i.e. when there are less than n elements in a line, then Xcas will append zeros to the corresponding row of the output matrix.
- `csv2gen` skips empty lines; they will not be included in the result.
- Using `csv2gen` is a simple way to import data to Xcas from text files. Equivalently, you can create a spreadsheet entry and use Table ▶ Insert CSV from its menu bar (see Section 2.10, p. 31).

Examples

To convert MATLAB array syntax to a `giac` matrix:

```
> csv2gen("1 2 3; 4 5 6"," ",";",string)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Assuming that the file `hooke.csv` (containing Hooke's Law demo data) is downloaded from here to the `Downloads` folder, you can load it by typing something like

```
> hooke:=csv2gen("/home/luka/Downloads/hooke.csv",",")
```

"Index"	"Mass (kg)"	"Spring 1 (m)"	"Spring 2 (m)"
1	0.0	0.05	0.05
2	0.49	0.066	0.066
3	0.98	0.087	0.08
4	1.47	0.116	0.108
5	1.96	0.142	0.138
6	2.45	0.166	0.158
7	2.94	0.193	0.174
8	3.43	0.204	0.192
9	3.92	0.226	0.205
10	4.41	0.238	0.232

The command

```
> tail(hooke)
```

is a convenient way to obtain the table with the header row removed (see Section 6.1.5, p. 71).

Application: loading real-world data to Xcas

Assume that you require global annual mean temperature anomaly data for the last two of centuries (it has been recorded since 1880). The corresponding CSV file can be found here. The file can be imported in XCAS by entering:

```
> data:=csv2gen("/home/luka/Downloads/annual_csv.csv",",");;
  header:=data[0]
```

["Source", "Year", "Mean"]

There are three columns in the obtained table: Source, Year, and Mean. The last column contains the mean anomalities in degrees Celsius. To collect different data sources, enter:

```
> sources:=set[op(tail(col(data,0)))]
```

⟦"GCAG", "GISTEMP"⟧

There are two sources of data: GCAG and GISTEMP, and the corresponding entries are interleaved. To sort data by source, enter:

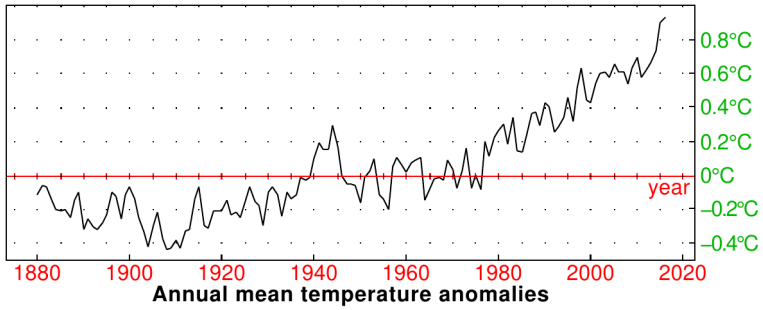
```
> t:=table();;
  for src in sources do
    t[src]=<sort(tran([col(select(r->r[0]==src,data),1..2)]));
  od;;
```

Indeed, `select` selects the `data` rows in which the first element is the source `src`; `col` returns a sequence containing the second and third column, which is converted into a two-row matrix by using the `[]`

delimiters; `tran` transposes the matrix, returning the desired list of pairs; finally, `sort` sorts the list according to the lexicographic order (effectively along the first column, i.e. the time axis).

For instance, to plot the GCAG data, enter:

```
> gcag:=t["GCAG"]:;
  labels=["year",""];
  legend=["","°C"];
  title="Annual mean temperature anomalies";
  listplot(gcag)
```



4.2 Directories

4.2.1 Working directories

Xcas has a working directory where it stores files that it creates; typically this is the user's home directory. The `pwd` command will tell you what the current working directory is, and and the `cd` command lets you change it.

- `pwd` takes no arguments.
- `pwd()` returns the name of the current working directory.
- `cd` command takes *dirname*, the name of a directory (a string).
- `cd(`*dirname*`)` changes the working directory to *dirname*.

Examples

```
> pwd()
```

The output might be something like:

"/home/username"

If you enter:

```
> cd("foo")
```

or (on a Unix system):

```
> cd("/home/username/foo")
```

then the working directory will change to the directory `foo`, if it exists. Afterwards, any files that you save from Xcas will be in that directory.

To load or read a file, it will need to be in the working directory. Note that if you have the same file name in different directories, then loading the file name will load the file in the current directory.

4.2.2 Internal directories

You can create a directory that is not actually on your hard drive but is treated like one by Xcas with the command `NewFold`.

- `NewFold` takes one argument: *MyIntDir*, a variable name (see Section 3.3.1, p. 36).
- `NewFold(`*MyIntDir*`)` creates a new internal directory named *MyIntDir*. (Note that quotation marks are not used.)

Internal directories will be listed with the `VARS()` command (see Section 3.3.9, p. 41). To actually use this directory, you will have to use the `SetFold` command.

- The `SetFold` command takes *MyIntDir*, the variable name of an internal directory created with `NewFold`.
- `SetFold(`*MyIntDir*`)` makes *MyIntDir* the working directory (see Section 4.2.1, p. 48).

You can print out the internal directory that you are in with the `GetFold` command.

- `GetFold` takes no arguments.
- `GetFold()` returns the name of the current internal directory.

The `DelFold` command will delete an internal directory.

- `DelFold` takes *MyIntDir*, the variable name of an internal directory.
- `DelFold(`*MyIntDir*`)` will delete the directory if it is empty.

5 Booleans and strings

5.1 Booleans

5.1.1 Boolean values

The symbols `true` and `false` are *booleans*, and are meant to indicate a statement is true or false.
These constants have synonyms:

- `true` is the same as `TRUE`.
- `false` is the same as `FALSE`.
- `true` and `false` are both integers with values 1 and 0, respectively, but have a different subtype than ordinary integers to indicate that they are boolean values.

A function which returns a boolean is called a *test* (or a *condition* or a *boolean function*).

5.1.2 Comparison operators

The usual comparison operators between numbers are examples of tests. In Xcas, they are the following infixed operators:

- a`==`b tests the equality between a and b and returns 1 if a is equal to b and `0` otherwise.
- a`!=`b returns 1 if a and b are different and `0` otherwise.
- a`>=`b returns 1 if a is greater than or equal to b and `0` otherwise.
- a`>`b returns 1 if a is strictly greater than b and `0` otherwise.
- a`<=`b returns 1 if a is less than or equal to b and `0` otherwise.
- a`<`b returns 1 if a is strictly less than b and `0` otherwise.

Remark. Note that a`=`b is *not* a boolean. This form is used to state that the expression *is* an equality, perhaps with the intent to solve it. To test for equality, you need to use a`==`b, which *is* a boolean.

5.1.3 Defining functions with boolean tests

use boolean tests to define functions not given by a single simple formula. Notably, use the `ifte` command or the `?:` operator to define piecewise-defined functions.

- `ifte` takes three arguments:
  - *condition*, a boolean condition.
  - *restrue*, the result to return if *condition* is true.
  - *resfalse*, the result to return if *condition* is false.
- `ifte(`*condition*, *restrue*, *resfalse*`)` returns *restrue* if *condition* holds and *resfalse* otherwise.

Example

You can define your own absolute value function with:

```
> myabs(x):=ifte(x>=0,x,-x)
```

Hence:

```
> myabs(-4)
```

4

However, `myabs` will return an error if it cannot evaluate the condition.

```
> myabs(x)
```

Ifte: Unable to check test Error: Bad Argument Value

The `?:` construct behaves similarly to `ifte`, but is structured differently and does not return an error if the condition cannot be evaluated.

- The `?:` construct takes three arguments:
  - *condition*, a boolean condition.
  - *restrue*, the result to return if *condition* is true.
  - *resfalse*, the result to return if *condition* is false.
- *condition*`?`*restrue*`:`*resfalse* returns *restrue* if *condition* holds and *resfalse* otherwise.

Example

You can define your absolute value function with

```
> myabs(x):=(x>=0)?x:-x
```

If you enter

```
> myabs(-4)
```

you will again get

4

but now if the conditional cannot be evaluated, you won't get an error.

```
> myabs(x)
```

$$x \geq 0 \ ? \ x \ : \ -x$$

The `when` and `IFTE` commands are prefixed synonyms for the `?:` construct.

- `when` (and `IFTE`) take three arguments:
  - *condition*, a boolean condition.
  - *restrue*, the result to return if *condition* is true.
  - *resfalse*, the result to return if *condition* is false.
- `when(`*condition*, *restrue*, *resfalse*`)` and `IFTE(`*condition*, *restrue*, *resfalse*`)` both return *restrue* if *condition* holds and *resfalse* otherwise.

`(`*condition*`) ?` *restrue* `:` *resfalse*

`when(`*condition*, *restrue*, *resfalse*`)`

and

IFTE(*condition*, *restrue*, *resfalse*)

all represent the same expression.

If you want to define a function with several pieces, it may be simpler to use the `piecewise` function. The latter can also be used for finding piecewise representations of expressions.

- `piecewise` can take an even number of arguments, followed by a single optional argument:
  - $cond_1$, $return_1$, $cond_2$, $return_2$, ..., $cond_n$, $return_n$, an arbitrary number of pairs of conditions and corresponding return values.
  - Optionally, *expr*, an expression.

  Alternatively, `piecewise` can take one argument, an expression *expr*, optionally followed by a second argument x, an identifier.
- With conditional arguments, `piecewise(`$cond_1, return_1, \ldots, cond_n, return_n \langle, expr\rangle$`)` returns an expression which evaluates to $return_k$ if $cond_k$ is the first true condition, or to *expr* if none of the conditions are true. If *expr* is not given, then the default value is `undef`.
- With a single argument, `piecewise(`*expr*`)` returns a piecewise representation of *expr*. This is useful if *expr* involves Heaviside/signum/absolute value functions, since `piecewise` attempts to remove them; if it fails, then *expr* is returned. The optional second argument may be given, specifying the relevant variable (otherwise it is guessed).

Examples

To define

$$f(x) = \begin{cases} -2 & \text{if } x < -2 \\ 3x+4 & \text{if } -2 \le x < -1 \\ 1 & \text{if } -1 \le x < 0 \\ x+1 & \text{if } x \ge 0 \end{cases}$$

enter:

```
> f(x):=piecewise(x<-2,-2,x<-1,3*x+4,x<0,1,x+1)
```

To verify, plot $f(x)$ for e.g. $-3 \le x \le 1$.

To convert the expression $x\,\theta\,(x+1) - x^2\,\theta\,(x-1) + x^3\text{sign}\,(x)$ to a piecewise expression, enter:

```
> f:=x*Heaviside(x+1)-x^2*Heaviside(x-1)+x^3*sign(x):;
```

and then

```
> piecewise(f)
```

or:

```
> piecewise(f,x)
```

$$\begin{cases} -x^3, & -1 > x \\ -x^3 + x, & 0 > x \\ x^3 + x, & 1 > x \\ x^3 - x^2 + x, & \text{otherwise} \end{cases}$$

`piecewise` can be used for "flattening" expressions containing piecewise defined subexpressions. For example:

```
> piecewise(1+piecewise(x<0,-x,x<1,x,1)*when(x<1/2,4x^2,2-2x),x)
```

$$\begin{cases} -4x^3+1, & 0>x \\ 4x^3+1, & \frac{1}{2}>x \\ -2x^2+2x+1, & 1>x \\ -2x+3, & \text{otherwise} \end{cases}$$

5.1.4 Logical operators

Booleans can be combined to form new booleans. For example with `and`, the statement "*bool1* `and` *bool2*" is true if both *bool1* and *bool2* are true, otherwise the statement is false.

Xcas has the standard boolean operators, as follows (a and b are two booleans):

- `or` or || — These are infixed operators. `(`a `or` b`)` (or `(`a || b`)`) returns 0 (or `false`) if a and b are both equal to 0 (or `false`) and returns 1 (or `true`) otherwise.
- `xor` — This is an infixed operator. It is the "exclusive or" operator, meaning "one or the other but not both". `(`a `xor` b`)` returns 1 if a is equal to 1 and b is equal to 0 or if a is equal to 0 and b is equal to 1, and returns 0 if a and b are both equal to 0 or if a and b are both equal to 1.
- `and` or `&&` — These are infixed operators. `(`a `and` b`)` (or `(`a `&&` b`)`) returns 1 (or `true`) if a and b are both equal to 1 (or `true`) and returns `0` (or `false`) otherwise.
- `not` — This is a prefixed operator. `not(`a`)` returns 1 (or `true`) if a is equal to 0 (or `false`), and `0` (or `false`) if a is equal to 1 (or `true`).

Examples

```
> 1>=0 or 1<0
```

$$1$$

```
> 1>=0 xor 1>0
```

$$0$$

```
> 1>=0 and 1>0
```

$$1$$

```
> not(0==0)
```

$$0$$

5.1.5 Transforming a boolean expression to a list

The `exp2list` command can transform certain booleans into a list.

- `exp2list` takes one argument: *eqseq*, a sequence of equalities (or inequalities) connected with `or`s, such as `(`$x=a_1$`) or ... or (`$x=a_n$`)`.
- `exp2list(`*eqseq*`)` returns the list $[a_1,\ldots,a_n]$ of right-hand sides of the (in)equalities.

The `exp2list` command is useful in TI mode for easier processing of the answer to a `solve` command.

Examples

```
> exp2list((x=2) or (x=0))
```

$$[2, 0]$$

```
> exp2list((x>0) or (x<2))
```

$$[0, 2]$$

In TI mode:

```
> exp2list(solve((x-1)*(x-2)))
```

$$[1, 2]$$

5.1.6 Transforming a list into a boolean expression

The `list2exp` command is the inverse of `exp2list`; it takes lists and tranforms them into boolean expressions. It can do this in two ways.

- `list2exp` can take two arguments:
  - L, a list of values of the form $[a_1, \ldots, a_n]$.
  - x, a variable name.
- `list2exp(`L, x`)` returns the boolean expression `(`$x = a_1$`) or ... or (`$x = a_n$`)`.
- Alternatively, `list2exp` can take two arguments:
  - L, a list where each element of L it itself a list of n values of the form $[a_1, \ldots, a_n]$.
  - *vars*, a list $[x_1, \ldots, x_n]$ of n variable names.
- `list2exp(`L, *vars*`)` returns a boolean expression of the form $(x_1 = a_1)$ `and ... and` $(x_n = a_n)$ for each list of n values in the first argument, combined with `or`s.

Examples

```
> list2exp([0,1,2],a)
```

$$a = 0 \vee a = 1 \vee a = 2$$

```
> list2exp(solve(x^2-1=0,x),x)
```

$$x = -1 \vee x = 1$$

```
> list2exp([[3,9],[-1,1]],[x,y])
```

$$x = 3 \wedge y = 9 \vee x = -1 \wedge y = 1$$

5.1.7 Evaluating booleans

The MAPLE command `evalb` evaluates a boolean expression (see Section 5.1, p. 50). Since XCAS evaluates booleans automatically, it includes a `evalb` command only here for compatibility and is equivalent to `eval` (see Section 9.1.1, p. 169).

- `evalb` takes *bool*, a boolean expression.
- `evalb(`*bool*`)` returns 1 if *bool* is true and returns 0 otherwise.

Examples

```
> evalb(sqrt(2)>1.41)
```

or:

```
> sqrt(2)>1.41
```

1

```
> evalb(sqrt(2)>1.42)
```

or:

```
> sqrt(2)>1.42
```

0

5.2 Strings

5.2.1 Characters and strings

Strings are delimited with quotation marks (`"`). A character is a string of length one.

Do not confuse `"` with `'` which is used to prevent evaluation of an expression (see Section 9.1.4, p. 170). For example, `"a"` returns a string with one character but `'a'` or `quote(a)` returns the variable `a` unevaluated.

When a string is entered on a command line, it is evaluated to itself, hence the output is the same string. You can use `+` to concatenate two strings or a string and another object (where the other object will be converted to a string, see Section 5.2.13, p. 61).

Examples

```
> "Hello"
```

"Hello"

```
> "Hello"+", how are you?"
```

"Hello, how are you?"

```
> "Hello"+ 123
```

"Hello123"

You can refer to a particular character of a string using index notation, like for lists (see Section 6.3, p. 78). Indices begin at 0 in Xcas mode, 1 in other modes.

Example

```
> "Hello"[1]
```

"e"

5.2.2 Newline character

A newline can be inserted into a string with `\n`.

Example

```
> "Hello\nHow are you?"
```

"Hello
How are you?"

5.2.3 Length of a string

The `size` or `length` command finds the length of a string (as well as the length of lists in general, see Section 6.1.3, p. 70).

- `size` takes *str*, a string.
- `size(`*str*`)` returns the length of the string.

Example

```
> size("hello")
```

5

5.2.4 Extracting portions of a string

The `left` and `right` commands find the left and right parts of a string. (See Section 8.2.3, p. 149, Section 6.5.1, p. 98, Section 6.6.2, p. 100, Section 6.3.6, p. 80, Section 9.3.4, p. 183, Section 9.3.5, p. 183, Section 28.2.8, p. 826 and Section 28.2.9, p. 827 for other uses of `left` and `right`.)

- `left` takes two arguments:
  - *str*, a string.
  - n, a non-negative integer.
- `left(`*str*$, n$`)` returns the first n characters of the string *str*.
- `right` takes two arguments:
  - *str*, a string.
  - n, a non-negative integer.
- `right(`*str*$, n$`)` returns the last n characters of the string *str*.

The `head` command finds the first character of a string.

- `head` takes *str*, a string.
- `head(`*str*`)` returns the first character of the string *str*.

The `mid` command finds a selected part from the middle of a string.

- `mid` takes three arguments:
  - *str*, a string.
  - p, an integer for the starting index of the result.
  - q, an integer q for the length of the string.

- `mid(`*str*, *p*, *q*`)` returns the part of the string *str* starting with the character at index *p* with length *q*. (Remember that the first index is 0 in XCAS mode.)

The `tail` command removes the first character of a string.

- `tail` takes *str*, a string.
- `tail(`*str*`)` returns the string *str* without its first character.

Examples

```
> left("hello",3)
```

"hel"

```
> right("hello",4)
```

"ello"

```
> head("Hello")
```

"H"

```
> mid("Hello",1,3)
```

"ell"

```
> tail("Hello")
```

"ello"

5.2.5 Finding and removing leading/trailing whitespace

The `trim` command can find and/or remove leading and trailing whitespace in a string (see Section 21.2.2, p. 569, Section 28.1.9, p. 820 and Section 28.2.2, p. 823 for other usages of `trim`).

- `trim` takes one mandatory arguments and one or two optional arguments:
  - *str*, a string.
  - Optionally, `left` or `right`, the symbol specifying an one-sided truncation.
  - Optionally, `index`, the symbol.
- `trim(`*str*$\langle$`,left|right`$\rangle\langle$`,index`$\rangle$`)` finds the index l of the first non-whitespace character in *str* and the index u of the first trailing whitespace character in *str*. If `index` is given, then the return value is either l if `left` is given, u if `right` is given, or $(l, u - l)$ otherwise (the last sequence contains the start and length of the truncated string). If `index` is omitted, then the return value is the portion between the lth character (inclusive) and uth character (exclusive); if `left` is given, then $u =$ `length(`*str*`)`, and if `right` is given, then $l = 0$.
- Whitespace characters in ASCII are: 9 (horizontal tab), 10 (line feed), 11 (vertical tab), 12 (form feed), 13 (carriage return) and 32 (space).

Instead of `trim`, you can also use the `strip` command to remove whitespace, but note that it discards only space characters.

Examples

```
> trim("  this is a string\n\t    ")
```

"this is a string"

```
> trim("  this is a string\n\t    ",right)
```

" this is a string"

```
> trim("  12345  ",index)
```

2, 5

5.2.6 Splitting strings into lists of tokens

The `split` command splits a string into a sequence of substrings with respect to a given separator. See Section 9.1.9, p. 172 for other uses of `split`.

- `split` takes two arguments:
  - *str*, a string.
  - *sep*, a string specifying the separator.
- `split(`*str*, *sep*`)` returns a list of substrings $[s_1, s_2, \ldots, s_n]$ of *str* such that `s1+sep+s2+sep+..+sn` returns *str*.
- To trim the returned substrings, you can enter the command `apply(trim,res)` provided that `res` holds the return value of `split` (see Section 5.2.5, p. 57).

Examples

```
> split("this is a phrase"," ")
```

["this", "is", "a", "phrase"]

```
> apply(expr,split("112--34--567--89","--"))
```

[112, 34, 567, 89]

5.2.7 Concatenation of a list of strings

The `concat` and `sum` commands concatenate elements of a list of strings together into a single string.

- `concat` and `sum` both take L, a list of strings.
- `concat(`L`)` and `sum(`L`)` both return the string obtained by *lst* concatenating strings from L together in the given order.

The `join` command concatenates a list of strings after inserting a specified separator string between adjacent elements.

- `join` takes two arguments:
  - *sep*, a string specifying the separator.
  - L, a list of strings $s_1, s_2, \ldots, s_n$.

- `join(`*sep*, *L*`)` returns the string `s1+sep+s2+sep+..+sn`.

The `cumSum` command works on strings like it does on expressions by doing partial concatenation (see Section 6.3.26, p. 90).

- `cumSum` takes *L*, a list of strings.
- `cumSum(`*L*`)` returns a sequence of strings where the element of index *k* is the concatenation of the strings in *L* with indices 0 to *k*.

Examples

```
> sum("Hello, ","is ","that ","you?")
```

"Hello, is that you?"

```
> join(" ",["Hello,","is","that","you?"])
```

"Hello, is that you?"

```
> cumSum("Hello, ","is ","that ","you?")
```

"Hello, ", "Hello, is ", "Hello, is that ", "Hello, is that you?"

5.2.8 ASCII code of a character

The `ord` command finds the ASCII code of a character.

- `ord` takes *str*, a string (or a list of strings). `ord(`*str*`)` returns the ASCII code of the first character of *str* (or the list of the ASCII codes of the first characters of the elements of the list *str*).

Example

```
> ord("a")
```

97

```
> ord("abcd")
```

97

```
> ord(["abcd","cde"])
```

[97, 99]

```
> ord(["a","b","c","d"])
```

[97, 98, 99, 100]

5.2.9 ASCII code of a string

The `asc` command finds the ASCII codes of all the characters in a string.

- `asc` takes *str*, a string.
- `asc(`*str*`)` returns the list of the ASCII codes of the characters of *s*.

Examples

```
> asc("abcd")
```

[97, 98, 99, 100]

```
> asc("a")
```

[97]

5.2.10 String defined by the ASCII codes of its characters

The `char` command translates ASCII codes to strings.

- `char` takes c, an integer representing an ASCII code or a list of ASCII codes.
- `char`(c) returns the string whose character has ASCII code c or whose characters have ASCII codes the elements of the list c.

Example

```
> char([97,98,99,100])
```

"abcd"

```
> char(97)
```

"a"

Note that there are 256 ASCII codes, 0 through 255. If `asc` is given an integer c not in that range, it will use the integer in that range which equals c modulo 256.

```
> char(353)
```

"a"

because $353 - 256 = 97$.

5.2.11 Finding a character in a string

The `inString` command tests to see if a string contains a character.

- `inString` takes two arguments:
  - *str*, a string.
  - c, a character.
- `inString`(*str*, c) returns the index of its first occurrence of the character c in the string *str*, or `-1` if c does not occur in *str*.

Examples

```
> inString("abcded","d")
```

3

```
> inString("abcd","e")
```

−1

5.2.12 Concatenating objects into a string

The `cat` command transforms a sequence of objects into a string.

- `cat` takes *seq*, a sequence of objects.
- `cat(`*seq*`)` returns the concatenation of the string representations of these objects as a single string.

Examples

```
> cat("abcd",3,"d")
```

"abcd3d"

```
> c:=5:;
  cat("abcd",c,"e")
```

"abcd5e"

```
> purge(c);
  cat(15,c,3)
```

"15c3"

5.2.13 Adding an object to a string

The `'+'` command can be used like `cat` (see Section 5.2.12, p. 61), and the `+` operator is the infixed version. (See Section 8.3.1, p. 151 for other uses of `+` and `'+'`.)

- `'+'` takes *seq*, a sequence of objects, at least one of which is a string.
- `'+'(`*seq*`)` returns the concatenation of the string representations of the objects in *seq*.

Remark. Note that `+` is infixed and `'+'` is prefixed.

Examples

```
> '+'("abcd",3,"d")
```

or:

```
> "abcd"+3+"d"
```

"abcd3d"

```
> c:=5
```

then:

```
> "abcd"+c+"d"
```

or:

```
> '+'("abcd",c,"d")
```

"abcd5d"

5.2.14 Transforming a real number into a string

The `cat` command (see Section 5.2.12, p. 61) can also be used to transform a real number into a string, as can `+` (see Section 5.2.13, p. 61).

If `cat` has a real number as an argument, the result will be a string.

Example

```
> cat(123)
```

"123"

Similarly, if you add a real number to an empty string, the result will be a string.

Example

```
> ""+123
```

"123"

5.2.15 Transforming a string into a number

The `expr` command transforms a string representing a valid XCAS statement into the actual statement.

- `expr` takes *str*, a string corresponding to an XCAS statement.
- `expr(`*str*`)` evaluates the statement.

Examples

```
> expr("a:=1")
```

1

```
> a
```

1

In particular, `expr` can transform a string representing a number into the number (see Section 3.1.1, p. 32).

```
> expr("123")
```

123

```
> expr("0123")
```

83

since `0123` represents a base 8 integer (see Section 5.4.1, p. 65) and $1 \cdot 8^2 + 2 \cdot 8 + 3 = 83$.

```
> expr("0x12f")
```

303

since `0x12f` represents a base 16 number and $1 \cdot 16^2 + 2 \cdot 16 + 15 = 303$.

```
> expr("123.4567")
```

123.4567

```
> expr("123e-5")
```

0.00123

5.2.16 Levensthein distance

The `levenshtein` command computes the Levenshtein distance between two words, which is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other.

- `levenshtein` takes two arguments:
  - s_1, a vector or string.
  - s_2, a vector or string.
- `levenshtein`(s_1, s_2) returns the minimum number n of edits required to transform s_1 into s_2.

Example

```
> levenshtein("kitten","sitting")
```

3

5.2.17 Hamming distance

The `hamdist` command computes the Hamming distance between two sequences of equal lengths, which is defined to be the number of elements at same positions which do not match.

- `hamdist` takes two arguments:
  - s_1, a vector or string.
  - s_2, a vector or string.
- `hamdist`(s_1, s_2) returns the number n of characters in s_1 that are different than the corresponding characters in s_2.

Example

```
> hamdist("cats","dogs")
```

3

5.3 Bitwise operators

5.3.1 Basic operators

Bitwise operators operate on the base 2 representations of integers, even if they are not presented in base 2. For example, the bitwise `or` (see Section 5.1.4, p. 53) operator will take two integers and and return an integer whose base 2 digits are the logical *or*s of the corresponding base two digits of the inputs (see Section 5.1.4, p. 53). Thus, to find the bitwise *or* of `6` and `4`, look at their base 2 representations, which are `0b110` (the `0b` prefix indicates that it's in base 2, see Section 3.1.1, p. 32) and `0b100`, respectively. The logical *or* or their rightmost digits is `(0 or 0)=0`. The logical `or` of their

next digits is `(1 or 0)=1`, and the logical `or` of their remaining digits is `(1 or 1)=1`. So the bitwise or of `6` and `4` is `0b110`, which is `6`.

To work with bitwise operators, it is not necessary but may be useful to work with integers in a base which is a power of 2. The integers can be entered in binary (base 2), octal (base 8) or hexadecimal (base 16) (see Section 5.4.1, p. 65). To write an integer in binary, prefix it with `0b`; to write an integer in octal, prefix it with `0` or `0o`; and to write a integer in hexadecimal (base 16), prefix it with `0x`. Integers may also be output in octal or hexadecimal notation (see Section 2.5.7, p. 15, item 2.5.7).

There are bitwise versions of the logical operators `or`, `xor` and `and`, called `bitor`, `bitxor` and `bitand`; they are all prefixed operators which take two arguments, which are both integers.

Examples

```
> bitor(0x12,0x38)
```

or:

```
> bitor(18,56)
```

58

because `18` is written `0x12` in base 16 or `0b010010` in base 2 and `56` is written `0x38` in base 16 or `0b111000` in base 2, hence `bitor(18,56)` is `0b111010` in base 2 and so is equal to 58.

```
> bitxor(0x12,0x38)
```

or:

```
> bitxor(18,56)
```

42

because `18` is written `0x12` in base 16 and `0b010010` in base 2 and `56` is written `0x38` in base 16 and `0b111000` in base 2, `bitxor(18,56)` is written `0b101010` in base 2 and so, is equal to 42.

```
> bitand(0x12,0x38)
```

or:

```
> bitand(18,56)
```

16

because `18` is written `0x12` in base 16 and `0b010010` in base 2 and `56` is written `0x38` in base 16 and `0b111000` in base 2, `bitand(18,56)` is written `0b010000` in base 2 and so is equal to 16.

5.3.2 Bitwise Hamming distance

The Hamming distance between two integers is the number of differences between the bits of the two integers. The `hamdist` command finds the Hamming distance between two integers.

- `hamdist` takes two arguments: m and n, both integers.
- `hamdist`(m, n) returns the Hamming distance between m and n.

Example

```
> hamdist(0x12,0x38)
```

or:

```
> hamdist(18,56)
```

3

because 18 is written 0x12 in base 16 and 0b010010 in base 2 and 56 is written 0x38 in base 16 and 0b111000 in base 2, hence hamdist(18,56) is equal to $1+0+1+0+1+0=3$.

5.4 Writing an integer in a different base

5.4.1 Writing an integer in base 2, 8 or 16

Integers are typically entered and displayed in base 10. You can also enter an integer in base 2 (binary), base 8 (octal) or base 16 (hexadecimal). You can enter:

- a number in base 2 by prefixing it with 0b; the remaining digits have to be 0 or 1.
- an octal number by prefixing it with 0 or 0o; the remaining digits have to be 0-7 since it is base 8.
- a hexadecimal number by prefixing it with 0x; the remaining digits have to be 0-9 or A-F (where A is 10, B is 11, ..., F is 15). Non-capital letters a-f are allowed to be used instead.

Examples

Since 101 in binary is the same as $1\cdot 1+0\cdot 2+1\cdot 2^2=5$ in decimal:

```
> 0b101
```

5

Since 512 in base 8 is the same as $2\cdot 1+1\cdot 8+5\cdot 8^2=330$ in decimal:

```
> 0512
```

330

Since 2F3 in base 16 is the same as $3\cdot 1+15\cdot 16+2\cdot 16^2=755$ in decimal:

```
> 0x2F3
```

755

You can have Xcas print integers in octal or hexadecimal, as well as the default decimal. To change the base used for display, you can click on the red CAS status button and choose from the Integer basis menu (see Section 2.5.7, p. 15, item 2.5.7). If you have Xcas set to display in hexadecimal, you will get the following:

```
> 15
```

0xF

```
> 0x15
```

0x15

5.4.2 Writing an integer in an arbitrary base

The convert or convertir command does various kinds of conversions depending on the option given as the second argument (see Section 10.1.10, p. 195).

One thing that convert can do is to convert integers to arbitrary bases and back to the default base, both with the option base.

Conversion from base 10. An integer can be converted into a list of digits in base b by using the `convert` command.

- `convert` takes three arguments:
  - n, an integer.
  - `base`, the symbol verbatim.
  - b, a positive integer, the value of the base.
- `convert(`n`,base,`b`)` returns the list of digits of the integer n when written in base b. The list of digits will start with the 1s term, then the bs term, the b^2 term, etc.

Examples

```
> convert(123,base,8)
```

$$[3, 7, 1]$$

To verify, input `0173` (see Section 5.4.1, p. 65) or `horner(revlist([3,7,1]),8)` (see Section 11.1.13, p. 217 and Section 6.3.14, p. 85) or `convert([3,7,1],base,8)` which all return 123.

The base used for `convert` can be any integer greater than 1. For example, input:

```
> convert(142,base,12)
```

$$[10, 11]$$

Conversion from an arbitrary base b. A number written in base b can be converted into a base 10 integer by using the `convert` command.

- `convert` takes three arguments:
  - L, a list of integers representing the digits of the integer in base b, assumed to go in order of increasing significance.
  - `base`, the symbol verbatim.
  - b, a positive integer, the value of the base.
- `convert(`L`,base,`b`)` returns the integer which, in base b, has the digits given in L.

Examples

```
> convert([3,7,1],base,8)
```

$$123$$

```
> convert([10,11],base,12)
```

$$142$$

6 Sequences, lists, and sets

6.1 Sequences and lists

6.1.1 Defining a sequence or a list

Recall (see Section 3.2.1, p. 34) that a sequence is represented by a sequence of elements separated by commas, either without delimiters, with parentheses (`(` and `)`) as delimiters, or with `seq[` and `]` as delimiters.

Examples

```
> a,b,c,d
```

or:

```
> (a,b,c,d)
```

or:

```
> seq[a,b,c,d]
```

$$a, b, c, d$$

Similarly (see Section 3.2.3, p. 35) a list (or a vector) is a sequence of elements separated by commas delimited with `[` and `]`.

Examples

```
> [1,2,5]
```

$$[1, 2, 5]$$

To create an empty list, input:

```
> []
```

$$[]$$

Remarks. Lists have more structure than sequences. For example, a list can contain lists (for example, a matrix is a list of lists of the same size, see Section 14.2, p. 325). Lists may be used to represent vectors (lists of coordinates), matrices, or univariate polynomials (lists of coefficients by decreasing order, see Section 11.1.1, p. 211).

Sequences, on the other hand, are flat. An element of a sequence cannot be a sequence.

See Section 6.3, p. 78 for some commands only for lists.

6.1.2 Making a sequence or a list

The `seq` command or `$` operator can create a sequence or a list.

- To create a sequence, `seq` takes three mandatory arguments and one optional argument:
  - *expr*, an expression depending on a parameter.

  - k, the parameter.
  - $a..b$, a range of values. The range can be combined with the parameter into one argument of $k = a..b$.
  - Optionally p, a step size (by default 1 or -1, depending on whether $b > a$ or $b < a$). This is only allowed if the previous two arguments are combined into one, $k = a..b$

  This is MAPLE-like syntax.

- `seq(`*expr*$, k, a..b$`)` or `seq(`*expr*$, k$`=`$a..b\,\langle, p\rangle$`)` returns the sequence formed by the values of `expr`, as k changes from a to b in steps of p.

Alternatively, a sequence can be created with the infixed `$` operator. Namely, *expr*`$(`k`=`$a..b$`)` returns the sequence formed by the values of `expr` as k changes from a to b. As a special case, *expr*`$`n creates a sequence consisting of n copies of *expr*.

There are two ways to create a list with `seq`.

- `seq` can take four mandatory arguments and one optional argument:
  - *expr*, an expression depending on a parameter.
  - k, the parameter.
  - a, the beginning value of the parameter.
  - b, the ending value of the parameter.
  - Optionally p, a step size (by default 1 or -1, depending on whether $b > a$ or $b < a$).

  This is TI-like syntax.

- `seq(`*expr*$, k, a, b\,\langle, p\rangle$`)` returns the list consisting of the values of `expr`, as k changes from a to b in steps of p.

- As a special case, `seq(`*expr*$, n$`)` creates a list consisting of n copies of *expr*.

- Alternatively, `seq` can take two arguments:
  - *expr*, an expression.
  - n, a positive integer.

- `seq(`*expr*$, n$`)` returns the list consisting of n copies of `expr`.

Remarks. In XCAS mode, the precedence of `$` is not the same as it is, for example, in MAPLE. In case of doubt, put the arguments of `$` in parenthesis. For example, the following commands are equivalent:

```
> seq(j^2,j=-1..3)
```

or

```
> (j^2)$(j=-1..3)
```

$$1, 0, 1, 4, 9$$

With MAPLE syntax, `j,a..b,p` is not valid. To specify a step p for the variation of j from a to b, use `j=a..b,p` or use the TI syntax `j,a,b,p` and get the sequence from the list with `op(...)`.

Examples

To create a sequence:

```
> seq(j^3,j,1..4)
```

or:

```
> seq(j^3,j=1..4)
```

or:

```
> (j^3)$(j=1..4)
```

$$1, 8, 27, 64$$

To create a list:

```
> seq(j^3,j,1,4)
```

$$[1, 8, 27, 64]$$

To create a sequence:

```
> seq(j^3,j=-1..4,2)
```

$$-1, 1, 27$$

To create a list:

```
> seq(j^3,j,-1,4,2)
```

$$[-1, 1, 27]$$

```
> seq(j^3,j,0,5,2)
```

$$[0, 8, 64]$$

```
> seq(j^3,j,5,0,-2)
```

or:

```
> seq(j^3,j,5,0,2)
```

$$[125, 27, 1]$$

```
> seq(j^3,j,1,3,0.5)
```

$$[1, 3.375, 8.0, 15.625, 27.0]$$

```
> seq(j^3,j,1,3,1/2)
```

$$\left[1, \frac{27}{8}, 8, \frac{125}{8}, 27\right]$$

To create a list with several copies of the same element:

```
> seq(t,4)
```

$$[t, t, t, t]$$

To create a sequence with several copies of the same element:

```
> seq(t,k=1..4)
```

or:

```
> t$4
```

$$t, t, t, t$$

Examples of sequences being used

Find the third derivative of $\ln(t)$ (see Section 13.2.1, p. 277):

```
> diff(log(t),t$3)
```

$$\frac{2}{t^3}$$

```
> l:=[[2,3],[5,1],[7,2]]
  seq((l[k][0])$(l[k][1]),k=0..size(l)-1)
```

$$(2, 2, 2), (5), (7, 7)$$

then:

```
> eval(ans())
```

$$2, 2, 2, 5, 7, 7$$

Transform a string into a list of its characters:

```
> chn:="abracadbra":;
  seq(chn[j],j,0,size(chn)-1)
```

$$[\text{“a”}, \text{“b”}, \text{“r”}, \text{“a”}, \text{“c”}, \text{“a”}, \text{“d”}, \text{“a”}, \text{“b”}, \text{“r”}, \text{“a”}]$$

6.1.3 Length of a sequence or list

You can find the length of a sequence or list with any of the `size`, `nops` or `length` commands.

- Each of `size`, `nops` and `length` takes S, a sequence or list.
- Each of `size`(S), `nops`(S) and `length`(S) returns the length of L.

Examples

```
> nops(a,e,i,o,u)
```

or:

```
> size(a,e,i,o,u)
```

or:

```
> length(a,e,i,o,u)
```

$$5$$

```
> nops([3,4,2])
```

or:

```
> size([3,4,2])
```

or:

```
> length([3,4,2])
```

$$3$$

6.1.4 Getting the first element of a sequence or list

The `head` command finds the first element of a sequence or list.

- `head` takes S, a sequence or list.
- `head(`S`)` returns the first element of S.

Examples

```
> head(A,B,C,D)
```

$$A$$

```
> head([0,1,2,3])
```

$$0$$

6.1.5 Getting a sequence or list without the first element

The `tail` command removes the first element of a list.

- `tail` takes L, a list.
- `tail(`L`)` returns L without its first element.

Example

```
> tail([0,1,2,3])
```

$$[1, 2, 3]$$

6.1.6 Getting an element of a sequence or a list

The elements of a sequence have indices beginning at 0 in XCAS mode and 1 in other modes (see Section 2.5.2, p. 14).

You can get the an element of index n of a sequence or list by following the sequence or list with $[n]$ (see Section 3.2.4, p. 35).

(Note that `head(`S`)` does the same thing as S`[0]`.)

Examples

```
> (0,3,2)[1]
```

$$3$$

```
> S:=2,3,4,5:;
  S[2]
```

$$4$$

```
> [A,B,C,D][2]
```

$$C$$

For lists, the `at` command can also be used to get the element at a specific position.

- `at` takes two arguments:
  - L, a list.
  - n, an integer.
- `at(`L, n`)` returns the element of S with index n.

Note. `at` cannot be used for sequences, since the second argument would be merged with the sequence.

Example

```
> [0,1,2][1]
```

or:

```
> at([0,1,2],1)
```

1

6.1.7 Finding a subsequence or a sublist

The bracket notation used to find elements of sequences and lists can also be used to extract a range of elements. If S is a sequence or list of size n, then S`[`n_1`..`n_2`]` returns the subsequence or sublist of S consisting of the elements with indices from n_1 to n_2, where $0 \leq n_1 \leq n_2 < s$ (in XCAS syntax mode) or $0 < n_1 \leq n_2 \leq s$ in other syntax modes.

For lists, the `at` command can also be used to get a sublist.

- `at` takes two arguments:
  - L, a list.
  - n_1`..`n_2, a range of integers.
- `at(`L, n_1`..`n_2`)` returns the sublist of L consisting of the elements with indices from n_1 to n_2.

Again, `at` cannot be used for sequences.

An alternative to using `at` for finding a sublist is the `mid` command, which again cannot be used for sequences.

- `mid` takes two mandatory and one optional argument:
  - L, a list.
  - n, the index of the beginning of the sublist.
  - Optionally, l, the length of the sublist.
- `mid(`$L, n\,\langle, l\rangle$`)` returns the sublist of L with index beginning at n. With the option l, the length of the sublist will be l, otherwise it will go to the end of the list L.

Examples

```
> [0,1,2,3,4][1..3]
```

$[1, 2, 3]$

```
> (A,B,C,D,E)[1..3]
```

$$B, C, D$$

```
> at([1,2,3,4,5],2..4
```

$$[3, 4, 5]$$

```
> mid([0,1,2,3,4,5],2,3)
```

$$[2, 3, 4]$$

```
> mid([0,1,2,3,4,5],2)
```

$$[2, 3, 4, 5]$$

6.1.8 Concatenating sequences

The , operator is an infixed operator which concatenates two sequences. (Note that it does not concatenate lists.)

Example

```
> A:=(1,2,3,4);
  B:=(5,6,3,4);
  A,B
```

$$1, 2, 3, 4, 5, 6, 3, 4$$

6.1.9 The + operator applied on sequences and lists

The infixed operator +, with two sequences as arguments, returns the total sum of the elements of the two sequences. This is different than with two lists as arguments, where the term by term sums of the elements of the two lists would be returned. (See Section 8.3.1, p. 151.) To use it as a prefix, it has to be quoted (’+’).

Examples

```
> (1,2,3,4,5,6)+(4,3,5)
```

or:

```
> '+'((1,2,3,4,5,6),(4,3,5))
```

$$33$$

```
> [1,2,3,4,5,6]+[4,3,5]
```

or:

```
> '+'([1,2,3,4,5,6],[4,3,5])
```

$$[5, 5, 8, 4, 5, 6]$$

6.1.10 Transforming sequences into lists and lists into sequences

To transform a sequence into list, you can put square brackets (`[]`) around the sequence. The `makevector` and `nop` commands have the same effect.

- `makevector` (or `nop`) takes S, a sequence.
- `makevector(`S`)` (or `nop(`S`)`) returns the list with the same elements as S in the same order.

The `makesuite` command transforms a list into a sequence. Note that `op` (see Section 8.2.3, p. 149) can do the same thing.

- `makesuite` takes L, a list.
- `makesuite(`L`)` returns the sequence with the same elements as L and the same order.

Examples

```
> [seq(j^3,j=1..4)]
```

or:

```
> [(j^3)$(j=1..4)]
```

or:

```
> nop(j^3$(j=1..4))
```

or:

```
> makevector(j^3$(j=1..4))
```

$$[1, 8, 27, 64]$$

```
> makesuite([0,1,2])
```

or:

```
> op([0,1,2])
```

$$0, 1, 2$$

6.2 Values of a sequence u_n

6.2.1 Array of values of a sequence

The `tablefunc` command fills two columns of a spreadsheet with a table of values of a function. The spreadsheet can be opened with Alt+T (see Section 2.10, p. 31).

- `tablefunc` takes four arguments:
  - $f(x)$, a formula for a function.
  - x, the variable.
  - x_0, the beginning value of x.
  - inc, an increment for x.
- `tablefunc(`$f(x), x, x_0, inc$`)` fills two columns of the spreadsheet, the current column and the following column, starting with the chosen cell. The current column starts with the variable x, followed by the initial value x_0, then x_0+inc, x_0+2inc, ... The following column starts with the formula $f(x)$, followed by $f(x)$ evaluated at the values in the first column. (If the current cell is column C, row n, it will contain x, the cell below it will contain inc, and the cell below it in row k will contain `=`$C(k-1)$`+`$C\$(n+1)$, and the corresponding cells in the next column will contain `=evalf(subst(`$D\$n, C\$n, Ck$`))`.)

Example

To display the values of the sequence $u_n = \sin(n)$, select a cell of a spreadsheet (for example `C0`) and input `tablefunc(sin(n),n,0,1)` in the spreadsheet commandline. You get:

row	C	D
	n	$\sin(n)$
	0	0.0
	1	0.841470984808
	2	0.909297426826
	3	0.14112000806
	4	−0.756802495308
	⋮	⋮

The graphic representation may be plotted with the `plotfunc` command (see 19.2.1).

6.2.2 Solving a recurrence relation or a system

The `seqsolve` command finds the terms of a recurrence relation.

- `seqsolve` takes three arguments:
  - *exprs*, an expression or list of expressions that define the recurrence relation.
  - *vars*, a list of the variables used.
  - a, the starting value.
- `seqsolve(`*exprs*, *vars*, a`)` returns a formula for the nth term of the sequence.

For example, if a recurrence relation is defined by $u_{n+1} = f(u_n, n)$ with $u_0 = a$, the arguments to `seqsolve` will be $f(x, n)$, $[x, n]$ and a. If the recurrence relation is defined by $u_{n+2} = g(u_n, u_{n+1}, n)$ with $u_0 = a$ and $u_1 = b$, the arguments to `seqsolve` will be $g(x, y, n)$, $[x, y, n]$ and $[a, b]$.

The recurrence relation must have a homogeneous linear part, the nonhomogeneous part must be a linear combination of a polynomials in n times geometric terms in n.

The `rsolve` command is an alternate way to find the values of a recurrence relation. Note that `rsolve` is more flexible than `seqsolve` since:

- The sequence does not have to start with u_0.
- The sequence can have several starting values, such as initial condition $u_0^2 = 1$, which is why `rsolve` returns a list.
- The notation for the recurrence relation is similar to how it is written in mathematics.
- `rsolve` takes three arguments:
  - *eqns*, an equation or list of equations that define the recurrence relation.
  - *fns*, the function or list of functions (with their variables) used.
  - *startvals*, the equation or list of equations for the starting values.
- `rsolve(`*eqns*, *fns*, *startvals*`)` returns a list containing a formula for the nth term of the sequence. (If there is more than one sequence, it will return a formula for each one.)

For example, if a recurrence relation is defined by $u_{n+1} = f(u_n, n)$ with $u_0 = a$, the arguments to `rsolve` will be $u(n+1) = f(u(n), n)$, $u(n)$ and $u(0) = a$.

The recurrence relation must either be a homogeneous linear part with a nonhomogeneous part being a linear combination of polynomials in n times geometric terms in n (such as $u_{n+1} = 2u_n + n3^n$), or a linear fractional transformation (such as $u_{n+1} = (u_n - 1)/(u_n - 2)$).

Examples

Find u_n, given that $u_{n+1} = 2u_n + n$ and $u_0 = 3$.

```
> seqsolve(2x+n,[x,n],3)
```

$$-n - 1 + 4 \cdot 2^n$$

Find u_n, given that $u_{n+1} = 2u_n + n3^n$ and $u_0 = 3$.

```
> seqsolve(2x+n*3^n,[x,n],3)
```

$$(n-3) \cdot 3^n + 6 \cdot 2^n$$

Find u_n, given that $u_{n+1} = u_n + u_{n-1}$, $u_0 = 0$ and $u_1 = 1$.

```
> seqsolve(x+y,[x,y,n],[0,1])
```

$$\frac{5\left(\frac{-\sqrt{5}+1}{2}\right)^n - 4\left(\frac{-\sqrt{5}+1}{2}\right)^n\sqrt{5} - 5\left(\frac{-\sqrt{5}+1}{2}\right)^n + 5\left(\frac{\sqrt{5}+1}{2}\right)^n + 4\left(\frac{\sqrt{5}+1}{2}\right)^n\sqrt{5} - 5\left(\frac{\sqrt{5}+1}{2}\right)^n}{20}$$

Find u_n and v_n, given that $u_{n+1} = u_n + 2v_n$, $v_{n+1} = u_n + n + 1$ with $u_0 = 1, v_0 = 1$.

```
> seqsolve([x+2*y,n+1+x],[x,y,n],[0,1])
```

$$\left[\frac{-2n - (-1)^n + 4 \cdot 2^n - 3}{2}, \frac{(-1)^n + 2 \cdot 2^n - 1}{2}\right]$$

Find u_n, given that $u_{n+1} = 2u_n + n$ and $u_0 = 3$.

```
> rsolve(u(n+1)=2*u(n)+n,u(n),u(0)=3)
```

$$[-n + 4 \cdot 2^n - 1]$$

Find u_n, given that $u_{n+1} = 2u_n + n$ and $u_1^2 = 1$.

```
> rsolve(u(n+1)=2*u(n)+n,u(n),u(1)^2=1)
```

$$\left[-n + \frac{3}{2} \cdot 2^n - 1, -n + \frac{1}{2} \cdot 2^n - 1\right]$$

Note that there are two formulas, since the starting formula $u_1^2 = 1$ gives two possible starting values: $u_1 = 1$ and $u_1 = 2$.

Find u_n, given that $u_{n+1} = 2u_n + n3^n$ and $u_0 = 3$.

```
> rsolve(u(n+1)=2*u(n)+n*3^n,u(n),u(0)=3)
```

$$[n \cdot 3^n + 6 \cdot 2^n - 3 \cdot 3^n]$$

Find u_n, given that $u_{n+1} = (u_n - 1)/(u_n - 2)$ and $u_0 = 4$.

```
> rsolve(u(n+1)=(u(n)-1)/(u(n)-2),u(n),u(0)=4)
```

$$\left[\frac{\left(20\sqrt{5} + 60\right)\left(\frac{\sqrt{5}-3}{2}\right)^n + 60\sqrt{5} - 140}{40\left(\frac{\sqrt{5}-3}{2}\right)^n + 20\sqrt{5} - 60}\right]$$

Find u_n given that $u_{n+1} = u_n + u_{n-1}$ with $u_0 = 0$, $u_1 = 1$.

```
> rsolve(u(n+1)=u(n)+u(n-1),u(n),u(0)=0,u(1)=1)
```

$$\left[-\frac{\sqrt{5}}{5}\left(\frac{-\sqrt{5}+1}{2}\right)^n + \frac{1}{5}\sqrt{5}\left(\frac{\sqrt{5}+1}{2}\right)^n\right]$$

Find u_n and v_n, given that $u_{n+1} = u_n + v_n$, $v_{n+1} = u_n - v_n$ with $u_0 = 0$ and $v_0 = 1$.

```
> rsolve([u(n+1)=u(n)+v(n),v(n+1)=u(n)-v(n)],[u(n),v(n)],[u(0)=1, v(0)=1])
```

$$\left[\frac{1}{2}\left(-\sqrt{2}+1\right)\left(-\sqrt{2}\right)^{n+1-1}+\frac{1}{2}\left(\sqrt{2}+1\right)\cdot 2^{\frac{n+1-1}{2}} \quad \frac{1}{2}\left(-\sqrt{2}\right)^{n+1-1}+\frac{1}{2}\cdot 2^{\frac{n+1-1}{2}}\right]$$

6.2.3 Table of values and graph of a recurrent sequence

The `tableseq` command fills a column of a spreadsheet with a recurrence relation. The spreadsheet can be opened with Alt+T (see Section 2.10, p. 31).

`tableseq` takes three arguments, which can be different depending on how many terms are involved in the recurrence relation.

- For a one term recurrence relation, `tableseq` takes three arguments:
  - $f(x)$, a formula which defines the recurrence, through $u_{n+1} = f(u_n)$.
  - x, the variable.
  - u_0, the initial term of the sequence.
- `tableseq(`$f(x), x, u_0$`)` fills the current column of the spreadsheet, starting with the selected cell (or cell 0 if the entire column is selected), with the formula $f(x)$, the next cell with the variable x, followed by the terms $u_0, u_1, \ldots$ of the sequence. (If the current cell is column C, row n, these latter cells will actually contain (if in row k) `=evalf(subst(`$C\$n, C\$(n+1), C(k-1)$`))`, which means if you change the value in one cell, the values in the later cells will change accordingly.) See also Section 19.8.4, p. 487, for a graphic representation of a one-term recurrence sequence.
- More generally, for a recurrence relation where each term depends on the previous k terms, `tableseq` takes three arguments:
  - $f(x_1, x_2, \ldots, x_k)$, a formula which defines the recurrence, through $u_{n+1} = f(u_n, \ldots, u_{n-k})$.
  - $[x_1, \ldots, x_k]$, a list of variables.
  - $[u_0, \ldots, u_{k-1}]$, a list of the beginning k terms.
- `tableseq(`$f(x_1, \ldots, x_k), [x_1, \ldots, x_k], [u_0, \ldots, u_{k-1}]$`)` fills the current column of the spreadsheet, starting with the selected cell (or cell 0 if the entire column is selected), with the formula $f(x_1, x_2, \ldots, x_k)$, followed by the variables $x_1, \ldots, x_k$, followed by the terms $u_0, u_1, \ldots$ of the sequence.

Examples

To display the values of the sequence $u_0 = 3.5, u_{n+1} = \sin(u_n)$, select a cell of the spreadsheet, say `B0`, and input `tableseq(sin(x),x,3.5)` in the command line. By pressing Enter, The corresponding column of the spreadsheet is filled with the recurrence relation and an initial portion of the sequence:

row	B
0	$\sin(x)$
1	x
2	3.5
3	−0.35078322769
4	−0.343633444925
5	−0.336910330426
...	...

To display the values of the Fibonacci sequence $u_0 = 1, u_1 = 1, \ldots, u_{n+2} = u_n + u_{n+1}$, select a cell, say B0, and input `tableseq(x+y,[x,y],[1,1])` in the commandline. You get:

row	B
0	$x + y$
1	x
2	y
3	1
4	1
5	2
...	...

6.3 Operations on lists

6.3.1 Sizes of lists within a list

Recall that one thing that distinguishes lists from sequences is that a list can be an element of another list. The `sizes` command finds the length of elements of a list, as long as each element is a list.

- `sizes` takes L, a list each of whose elements is a list.
- `sizes(`L`)` returns the list of the lengths of the elements of L.

Example

```
> sizes([[3,4],[2]])
```

$$[2, 1]$$

6.3.2 Creating a list by using a function

The `makelist` command creates lists built from values of a function.

- `makelist` takes three mandatory arguments and one optional argument:
  - f, a function (see Section 8.2.1, p. 148).
  - a and b, two real numbers.
  - Optionally p, a step size (by default 1 if $b > a$ and -1 if $b < a$).
- `makelist(`$f, a, b\,\langle, p\rangle$`)` returns the list $[f(a), f(a+p), \ldots, f(a+kp)]$ with k such that

$$a < a + kp \leq b < a + (k+1)p \quad \text{or} \quad a > a + kp \geq b > a + (k+1)p.$$

Examples

(In these examples, purge `x` if `x` is not symbolic.)

```
> makelist(x->x^2,3,5)
```

or:

```
> makelist(x->x^2,3,5,1)
```

or:

```
> h(x):=x^2:;
  makelist(h,3,5,1)
```

$$[9, 16, 25]$$

```
> makelist(x->x^2,3,6,2)
```

$$[9, 25]$$

```
> makelist(4,1,3)
```

$$[4, 4, 4]$$

The above command line regards 4 as the constant function, and so creates a list with entries 4, from integers 1 to 3. The command

```
> [4$3]
```

is an equivalent.

6.3.3 Creating a list with zeros

The `newList` makes a list of all zeros.

- `newList` takes n, a positive integer.
- `newList(`n`)` returns a list of n zeros.

Example

```
> newList(3)
```

$$[0, 0, 0]$$

6.3.4 Creating a list of integers

The `range` command creates lists of equally spaced numbers. It can take one, two or three arguments.

- With one argument, `range` takes n, a positive integer.
- `range(`n`)` returns the list $[0, 1, \ldots, n-1]$.
- Alternatively, `range` takes two mandatory and one optional argument:
  - a and b, two real numbers with $a < b$ (unless the third argument p is provided and negative).
  - Optionally, p, a nonzero real number used for the step size (by default 1). If $p < 0$, then a must be larger than b.
- `range(`$a, b \langle, p \rangle$`)` returns the list $[a, a+p, \ldots]$ up to, but not including, b.

Examples

```
> range(5)
```

$$[0, 1, 2, 3, 4]$$

```
> range(4,10)
```

$$[4, 5, 6, 7, 8, 9]$$

```
> range(2.3,7.4)
```

$$[2.3, 3.3, 4.3, 5.3, 6.3, 7.3]$$

```
> range(4,13,2)
```

$$[4, 6, 8, 10, 12]$$

```
> range(10,4,-1)
```

$$[10, 9, 8, 7, 6, 5]$$

You can use the `range` command to create a list of values $f(k)$, where k is an integer satisfying a certain condition. (See Section 25.3.3, p. 659 for the `for` loop used below.)

You can list the values of an expression in a variable which goes over a range defined by `range`. For example:

```
> [k^2+k for k in range(10)]
```

$$[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]$$

You can list the values of an expression in a variable which goes over a range defined by `range` and which satisfies a given condition. For example:

```
> [k for k in range(4,10) if isprime(k)]
```

$$[5, 7]$$

(See Section 7.1.14, p. 112 for `isprime`.)

```
> [k^2+k for k in range(1,10,2) if isprime(k)]
```

$$[12, 30, 56]$$

6.3.5 Selecting elements of a list

The `select` command selects elements of a list meeting the given conditions.

- `select` takes two arguments:
  - f, a boolean function.
  - L, a list.
- `select(`f, L`)` returns the sublist of L consisting of the elements c such that $f(c)$ returns `true`.

Example

```
> select(x->(x>=2),[0,1,2,3,1,5])
```

$$[2, 3, 5]$$

6.3.6 Obtaining left and right portions of a list

The `left` and `right` commands find the left and right parts of a list. (See Section 5.2.4, p. 56, Section 8.2.3, p. 149, Section 6.5.1, p. 98, Section 6.6.2, p. 100, Section 9.3.4, p. 183 and Section 9.3.5, p. 183 for other uses of `left` and `right`.)

- `left` takes two arguments:
  - L, a list.
  - n, a positive integer.

- `left`(L, n) returns the first n elements of L.
- `right` takes two arguments:
  - L, a list.
  - n, a positive integer.
- `right`(L, n) returns the last n elements of L.

Examples

```
> left([0,1,2,3,4,5,6,7,8],3)
```

$$[0, 1, 2]$$

```
> right([0,1,2,3,4,5,6,7,8],4)
```

$$[5, 6, 7, 8]$$

6.3.7 Modifying the elements of a list

The `subsop` command can be used to modify elements in a list.

- `subsop` takes two arguments:
  - L, a list.
  - i=*value*, an index and a new value.
- (In all but Maple mode).
  `subsop`$(L, i$=*value*$)$ returns the list L with the value at index i replaced by *value*.
- (In Maple mode; the only difference is the order of the arguments).
  `subsop`$(i$=*value*$, L)$ returns the list L with the value at index i replaced by *value*.

Remark. If the second argument is i=`NULL`, then the element at index i is removed from L. You can also redefine elements (or define new elements, but not remove elements) with `:=`.

Examples

Input in Xcas mode, the index of the first element is 0:

```
> subsop([0,1,2],1=5)
```

or:

```
> L:=[0,1,2];L[1]:=5
```

$$[0, 5, 2]$$

Input in Xcas mode, the index of the first element is 0:

```
> subsop([0,1,2],'1=NULL')
```

$$[0, 2]$$

Input in MuPAD or TI mode, the index of the first element is 1:

```
> subsop([0,1,2],2=5)
```

or:

```
> L:=[0,1,2];L[2]:=5
```

$$[0, 5, 2]$$

When using `:=` to insert an element in a list, the list will be padded with zeros if necessary.

```
> L:=[]
```

then:

```
> L[3]:=5
```

$$[0, 0, 0, 5]$$

In Maple mode the arguments are permuted and the index of the first element is 1.

```
> subsop(2=5,[0,1,2])
```

or:

```
> L:=[0,1,2];L[2]:=5
```

$$[0, 5, 2]$$

6.3.8 Removing elements from a list

Elements can be removed from a list by using either the `suppress` command (which removes a single element) or the `remove` command (which removes all elements meeting a given conditions).

- `suppress` takes two arguments:
  - L, a list.
  - i, a nonnegative integer.
- `suppress`(L, i) returns the list L with the element at index i removed.
- `remove` takes two arguments:
  - f, a boolean function.
  - L, a list.
- `remove`(f, L) returns the sublist of L with the elements c such that $f(c)$`==true` removed.

Examples

```
> suppress([3,4,2],1)
```

$$[3, 2]$$

```
> remove(x->(x>=2),[0,1,2,3,1,5])
```

$$[0, 1, 1]$$

You can use `remove` to remove characters from a string. For example, to remove all instances of a given letter from a string, enter the following code in a program editor level (see Section 25.1.2, p. 647 for writing functions):

```
f(s,c):={
  local ordc,l,r,x,k;
  ordc:=ord(c);
  l:=length(s)-1;
  purge(x,k);
  r:=remove(x->(ord(x)==ordc),seq(s[k],k,0,l));
  return char(ord(r));
}
```

Here `s` is the input string an `c` is the letter to be removed. Now, for example:

```
> f("abracadabra","a")
```

"brcdbr"

6.3.9 Inserting an element into a list or a string

The `insert` command inserts elements into a list or string.

- `insert` takes three arguments:
  - L, a list or a string.
  - i, an integer (the index).
  - x, an object.
- `insert`(L, i, x) returns L with x inserted at index i and the necessary elements shifted to the right.

Examples

```
> insert([3,4,2],2,5)
```

$[3, 4, 5, 2]$

```
> insert("342",2,"5")
```

"3452"

`insert` returns an error if the index is too large.

```
> insert([3,4,2],4,5)
```

insert([3,4,2],4,5)
Error: Invalid dimension

6.3.10 Appending an element at the end of a list

The `append` command adds an element to the end of a list.

- `append` takes two arguments:
  - L, a list.
  - x, an object.
- `append`(L, x) returns a list L with the additional element x at the end.

Examples

```
> append([3,4,2],1)
```

$$[3, 4, 2, 1]$$

```
> append([1,2],[3,4])
```

$$[1, 2, [3, 4]]$$

6.3.11 Prepending an element at the beginning of a list

The `prepend` command adds an element to the beginning of a list.

- `prepend` takes two arguments:
  - x, an object.
  - L, a list.
- `prepend`(x, L) returns a list containing x as the first element followed by the elements of L.

Examples

```
> prepend([3,4,2],1)
```

$$[1, 3, 4, 2]$$

```
> prepend([1,2],[3,4])
```

$$[[3, 4], 1, 2]$$

6.3.12 Concatenating two lists or a list and an element

The `concat` or `augment` command combines to lists or adds an element to a list.

- `concat` takes two arguments: L_1 and L_2, two lists or a list and an element (in any order).
- `concat`(L_1, L_2) returns a list consisting of the elements of L_1 (or L_1 itself if it is not a list) followed by the elements of L_2 (or L_2 itself).

Examples

```
> concat([3,4,2],[1,2,4])
```

or:

```
> augment([3,4,2],[1,2,4])
```

$$[3, 4, 2, 1, 2, 4]$$

```
> concat([3,4,2],5)
```

or:

```
> augment([3,4,2],5)
```

$$[3, 4, 2, 5]$$

```
> concat(2,[5,4,3])
```

or:

```
> augment(2,[5,4,3])
```

$$[2, 5, 4, 3]$$

```
> concat([[3,4,2]],[[1,2,4]])
```

or:

```
> augment([[3,4,2]],[[1,2,4]])
```

$$\begin{bmatrix} 3 & 4 & 2 & 1 & 2 & 4 \end{bmatrix}$$

6.3.13 Flattening a list

The `flatten` command replaces any sublists of a list by their elements.

- `flatten` takes L, a list.
- `flatten`(L) returns a list which is the result of recursively replacing any elements that are lists by the elements, resulting in a list with no lists as elements.

Example

```
> flatten([[1,[2,3],4],[5,6]])
```

$$[1, 2, 3, 4, 5, 6]$$

Remark. If the original list is a matrix, you can also use the `mat2list` command for this (see Section 6.3.33, p. 95).

6.3.14 Reversing order in a list

The `revlist` command reverses the elements of a list or sequence.

- `revlist` takes L, a list or sequence.
- `revlist`(L) returns a list or sequence with the same elements as L in the reverse order.

Examples

```
> revlist([0,1,2,3,4])
```

$$[4, 3, 2, 1, 0]$$

```
> revlist([0,1,2,3,4],3)
```

$$3, [0, 1, 2, 3, 4]$$

6.3.15 Rotating a list

The `rotate` command rotates a list; namely it moves elements from one side and puts them on the other.

- `rotate` takes one mandatory argument and one optional argument:
  - L, a list.
  - Optionally, n, an integer (by default $n = -1$).
- `rotate(`L`)` returns the list formed by rotating L n places to the left if $n > 0$ or $-n$ places to the right if $n < 0$. Elements leaving the list from one side come back on the other side. By default $n = -1$ and the last element becomes the first.

Examples

```
> rotate([0,1,2,3,4])
```

$$[4, 0, 1, 2, 3]$$

```
> rotate([0,1,2,3,4],2)
```

$$[2, 3, 4, 0, 1]$$

```
> rotate([0,1,2,3,4],-2)
```

$$[3, 4, 0, 1, 2]$$

6.3.16 Shifting the elements of a list

The `shift` command shifts the elements of a list.

- `shift` takes one mandatory argument and one optional argument:
  - L, a list.
  - Optionally, n, an integer (by default $n = -1$).
- `shift(`L`)` returns the list formed by shifting the elements of L n places to the left if $n > 0$ or $-n$ places to the right if $n < 0$. Elements leaving the list from one side are replaced by zeros on the other side.

Examples

```
> shift([0,1,2,3,4])
```

$$[0, 0, 1, 2, 3]$$

```
> shift([0,1,2,3,4],2)
```

$$[2, 3, 4, 0, 0]$$

```
> shift([0,1,2,3,4],-2)
```

$$[0, 0, 0, 1, 2]$$

6.3.17 Sorting

The `sort` command sorts lists and expression in various ways.

- `sort` takes one mandatory argument and one optional argument:
  - L, a list or expression.
  - Optionally, f, a boolean function of two variables (by default, f if the function `(x,y) -> x<=y`).
- `sort(`L`,` f`)` (for a list L) returns a copy of L sorted according to the order given by f. By default, this means that it will be sorted in increasing order.
- `sort(`L`,` f`)` (for an expression L) returns a copy of L with the terms in sums and products collected and sorted.

Note that using `(x,y)->x>=y` for f will to sort the list in decreasing order. This may also be used to sort a list of lists (that `sort` with one argument does not know how to sort).

Examples

```
> sort([3,4,2])
```

$$[2, 3, 4]$$

```
> sort(exp(2*ln(x))+x*y-x+y*x+2*x)
```

$$2xy + \mathrm{e}^{2\ln x} + x$$

```
> simplify(exp(2*ln(x))+x*y-x+y*x+2*x)
```

$$x^2 + 2xy + x$$

```
> sort([3,4,2],(x,y)->x>=y)
```

$$[4, 3, 2]$$

6.3.18 Sorting a list by increasing order

The `SortA` command sorts a list or a matrix in increasing order.

- `SortA` takes L, a list.
- If L is not a matrix, then `SortA(`L`)` returns a copy of L with the elements in increasing order (if L is not a matrix).
- If L is a matrix, then `SortA(`L`)` returns a copy of L with columns sorted according to increasing order in the first row.

Examples

```
> SortA([3,4,2])
```

$$[2, 3, 4]$$

```
> SortA([[3,4,2],[6,4,5]])
```

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 4 \end{bmatrix}$$

6.3.19 Sorting a list by decreasing order

The `SortD` command sorts a list or a matrix in decreasing order.

- `SortD` takes L, a list.
- If L is not a matrix, then `SortA(`L`)` returns a copy of L with the elements in decreasing order.
- If L is a matrix, then `SortA(`L`)` returns a copy of L with columns sorted according to decreasing order in the first row.

Examples

```
> SortD([3,4,2])
```

$$[4, 3, 2]$$

```
> SortD([[3,4,2],[6,4,5]])
```

$$\begin{bmatrix} 4 & 3 & 2 \\ 4 & 6 & 5 \end{bmatrix}$$

6.3.20 Sorting by a permutation

The `sortperm` command returns the permutation which sorts the given list in ascending order or the list sorted by the given permutation.

- `sortperm` takes one or two arguments:
  - V, a nonempty list.
  - P, a permutation (optional).
- If called with one argument, `sortperm` returns the permutation which sorts the list V in ascending order. If called with two arguments, it returns a copy of V sorted according to P.
- `sortperm` is useful when several lists of equal lengths need to be sorted in the same order. Sorting by permutation is optimally efficient.

Examples

```
> V:=[30,25,40,10,20]; P:=sortperm(V)
```

$$[3, 4, 1, 0, 2]$$

```
> S:=["ab","cd","ad","bc","de"]; sortperm(S,P)
```

$$[\text{“bc”}, \text{“de”}, \text{“cd”}, \text{“ab”}, \text{“ad”}]$$

6.3.21 Counting elements equal to a given value

The `count_eq` command counts the number of elements of a list equal to a given value.

- count_eq takes two arguments:
  - x, a real number.
  - L, a list or matrix of real numbers.
- `count_eq(`x, L`)` returns the number of elements of L which are equal to x.

Example

```
> count_eq(12,[2,12,45,3,7,78])
```

1

6.3.22 Counting elements smaller than a given value

The `count_inf` command counts the number of elements of a list strictly less than a given value.

- count_inf takes two arguments:
  - x, a real number.
  - L, a list or matrix of real numbers.
- `count_inf`(x, L) returns the number of elements of L which are strictly less than x.

Example

```
> count_inf(12,[2,12,45,3,7,78])
```

3

6.3.23 Counting elements greater than a given value

The `count_sup` command counts the number of elements of a list strictly greater than a given value.

- count_sup takes two arguments:
  - x, a real number.
  - L, a list or matrix of real numbers.
- `count_sup`(x, L) returns the number of elements of L which are strictly greater than x.

Example

```
> count_sup(12,[2,12,45,3,7,78])
```

2

6.3.24 Sum of elements of a list

The `sum` or `add` command adds the elements of a list or sequence of numbers.

- `sum` takes L, a list or sequence of numbers.
- `sum`(L) returns the sum of the elements of L.

Example

```
> sum(2,3,4,5,6)
```

20

6.3.25 Sum of list (or matrix) elements transformed by a function

The `count` command adds the values of a function applied to the elements of a list or matrix.

- `count` takes two arguments:
  - f, a real-valued or boolean-valued function.
  - L, a list or a matrix.
- `count(`f, L`)` returns the sum of $f(x)$ for all elements x in L.
  If f is a boolean-valued function, this is just the number of elements for which the boolean is true.

Examples

```
> count((x)->x,[2,12,45,3,7,78])
```

147

because $2 + 12 + 45 + 3 + 7 + 78 = 147$.

```
> count((x)->x<12,[2,12,45,3,7,78])
```

3

```
> count((x)->x==12,[2,12,45,3,7,78])
```

1

```
> count((x)->x>12,[2,12,45,3,7,78])
```

2

```
> count(x->x^2,[3,5,1])
```

35

Indeed, $3^2 + 5^2 + 1^1 = 35$.

```
> count(id,[3,5,1])
```

9

Indeed, `id` is the identity functions and $3 + 5 + 1 = 9$.

```
> count(1,[3,5,1])
```

3

Indeed, `1` is the constant function equal to 1 and $1 + 1 + 1 = 3$.

6.3.26 Cumulated sum of the elements of a list

The `cumSum` command finds the partial sums of a list or sequence.

- `cumSum` takes L, a list or sequence of numbers or of strings.
- `cumSum(`L`)` returns the list or sequence with same length as L and whose kth element is the sum or concatenation of elements 0 through k of L.

Examples

```
> cumSum(sqrt(2),3,4,5,6)
```

$$\sqrt{2}, \sqrt{2}+3, \sqrt{2}+3+4, \sqrt{2}+3+4+5, \sqrt{2}+3+4+5+6$$

```
> normal(cumSum(sqrt(2),3,4,5,6))
```

$$\sqrt{2}, \sqrt{2}+3, \sqrt{2}+7, \sqrt{2}+12, \sqrt{2}+18$$

```
> cumSum(1.2,3,4.5,6)
```

$$1.2, 4.2, 8.7, 14.7$$

```
> cumSum([0,1,2,3,4])
```

$$[0, 1, 3, 6, 10]$$

```
> cumSum("a","b","c","d")
```

"a", "ab", "abc", "abcd"

```
> cumSum(["a","ab","abc","abcd"])
```

["a", "aab", "aababc", "aababcabcd"]

6.3.27 Products

The `product` command can find the products of elements of an expression (see Section 6.3.27, p. 91), the elements of a list (see Section 6.3.27, p. 92), the elements of the columns of a matrix (see Section 14.3.6, p. 342), and the term-by-term (Hadamard) product of two matrices (see Section 14.3.8, p. 343).

Product of values of an expression. The `product` or `mul` command can find the product of the values of an expression as the variable changes.

- `product` takes four mandatory arguments and one optional argument:
  - *expr*, an expression.
  - x, the name of a variable.
  - a and b, real numbers.
  - Optionally p, a real number representing a step size. (By default $p = 1$).
- `product(`*expr*$x, a, b\langle p \rangle$`)` returns the product of the values of *expr* as x goes from a to b in steps of p.

  This syntax is for compatibility with MAPLE.

Examples

```
> product(x^2+1,x,1,4)
```

or:

```
> mul(x^2+1,x,1,4)
```

$$1700$$

Indeed, $(1^2+1)(2^2+1)(3^2+1)(4^2+1) = 1700$.

```
> product(x^2+1,x,1,5,2)
```

or:

```
> mul(x^2+1,x,1,5,2)
```

520

Indeed, $(1^2+1)(3^2+1)(5^2+1) = 520$.

Product of elements of a list. The `product` or `mul` command can find the products of elements of a list.

- `product` takes one mandatory and one optional argument:
  - L, a list of numbers.
  - Optionally, L_2, another list of numbers the same length as L.
- `product(`L`)` returns the product of the elements of L.
- `product(`L, L_2`)` returns the term by term product of the elements of L and L_2.

(See also Section 14.3.8, p. 343.)

Examples

```
> product([2,3,4])
```

or:

```
> mul([2,3,4])
```

24

```
> product([[2,3,4],[5,6,7]])
```

$[10, 18, 28]$

```
> product([2,3,4],[5,6,7])
```

or:

```
> mul([2,3,4],[5,6,7])
```

$[10, 18, 28]$

6.3.28 Applying a function of one variable to the elements of a list

The `apply` and `map` commands can both apply a function to a list of elements, but take arguments in different orders (that is required for compatibility reasons). The `apply` command also works on matrices (see Section 14.2.6, p. 340) and the `map` command also works on polynomials in internal sparse format (see Section 11.1.3, p. 211).

- `apply` takes two arguments:
  - f, a function.
  - L, a list.
- `apply(`f, L`)` returns a list whose elements are $f(x)$ for the elements x of L.

Note that `apply` returns a list (`[]`) even if the second argument is not a list.

- `map` takes two arguments:
  - L, a list or a polynomial in internal format (see Section 11.1.2, p. 211).
  - f, a function.
- `map(`L, f`)` returns a list whose elements are $f(x)$ for the elements x of L.

Examples

```
> apply(x->sqrt(x),[16,9,4,1])
```

or:

```
> map([16,9,4,1],x->sqrt(x))
```

$$[4, 3, 2, 1]$$

```
> apply(x->x^2,[3,5,1])
```

or:

```
> map([3,5,1],x->x^2)
```

(or first define the function $h(x) = x^2$.)

```
> h(x):=x^2
```

then:

```
> apply(h,[3,5,1])
```

or:

```
> map([3,5,1],h)
```

$$[9, 25, 1]$$

Define the function $g(x) = [x, x^2, x^3]$.

```
> g:=(x)->[x,x^2,x^3]
```

then:

```
> apply(g,[3,5,1])
```

or:

```
> map([3,5,1],g)
```

$$\begin{bmatrix} 3 & 9 & 27 \\ 5 & 25 & 125 \\ 1 & 1 & 1 \end{bmatrix}$$

Remark. If `x` is not a symbol, then it needs to be purged. Note that if L_1, L_2 and L_3 are lists, then `sizes([`L_1, L_2, L_3`])` is equivalent to `map(size,[`L_1, L_2, L_3`])`.

6.3.29 Applying a bivariate function to the elements of two lists

The `zip` command applies a bivariate function to the elements of two lists.

- `zip` takes three arguments:
  - f, a function of two variables.

  - L_1 and L_2, two lists of the same size n.

- `zip(`f, L_1, L_2`)` returns a list of size n whose kth element is f applied to the kth elements of L_1 and L_2.

Examples

```
> zip('sum',[a,b,c,d],[1,2,3,4])
```

$$[a+1, b+2, c+3, d+4]$$

```
> zip((x,y)->x^2+y^2,[4,2,1],[3,5,1])
```

or:

```
> f:=(x,y)->x^2+y^2:;
  zip(f,[4,2,1],[3,5,1])
```

$$[25, 29, 2]$$

```
> f:=(x,y)->[x^2+y^2,x+y]:;
  zip(f,[4,2,1],[3,5,1])
```

$$\begin{bmatrix} 25 & 7 \\ 29 & 7 \\ 2 & 2 \end{bmatrix}$$

6.3.30 Folding sequences

The `foldl` (left-fold) and `foldr` (right-fold) commands take an infixed operator or function of two variables and apply them across a sequence of inputs through left and right association.

- `foldl` takes a sequence of arguments:
  - R, an infixed operator or function of two variables.
  - I, an initial value.
  - $a_1, a_2, \ldots, a_k$, an arbitrary number of additional arguments.
- `foldl(`$R, I, a_1, \ldots, a_k$`)` returns $R(\ldots R(R(I, a_1), a_2) \ldots, a_k)$.
- `foldr` takes a sequence of arguments:
  - R, an infixed operator or function of two variables.
  - I, an initial value.
  - $a_1, a_2, \ldots, a_j$, an arbitrary number of additional arguments.
- `foldr(`$R, I, a_1, a_2, \ldots, a_k$`)` returns $R(a, \ldots (R(a_1, R(a_2, \ldots R(a_{k-1}, R(a_k, I)))))$.

Examples

```
> foldl('^',2,3,5)
```

$$32768$$

which is equal to $2^{(3^5)}$.

```
> foldr('^',2,3,5)
```

$$847288609443$$

which is equal to $3^{(5^2)}$.

6.3.31 List of differences of consecutive terms

The `deltalist` command finds lists of differences.

- `deltalist` takes L, a list.
- `deltalist(`L`)` returns the list of the difference of all pairs of consecutive terms of L.

Example

```
> deltalist([5,8,1,9])
```

$$[3, -7, 8]$$

6.3.32 Creating a matrix from a list

The `listmat` command turns a list into a matrix by chopping up the list into separate rows.

- `list2mat` takes two arguments:
  - L, a list.
  - p, a postive integer.
- `list2mat(`L, p`)` returns the matrix whose first row consists of the first p elements of L, whose next row consists of the next p elements of L, etc. If there are not enough elements to fill up a row, zeros are added.

Examples

```
> list2mat([5,8,1,9,5,6],2)
```

$$\begin{bmatrix} 5 & 8 \\ 1 & 9 \\ 5 & 6 \end{bmatrix}$$

```
> list2mat([5,8,1,9],3)
```

$$\begin{bmatrix} 5 & 8 & 1 \\ 9 & 0 & 0 \end{bmatrix}$$

6.3.33 Creating a list from a matrix

The `mat2list` flattens a matrix into a list. (See also Section 6.3.13, p. 85.)

- `mat2list` takes A, a matrix.
- `mat2list(`A`)` returns the list of the coefficients of A.

Example

```
> mat2list([[5,8],[1,9]])
```

$$[5, 8, 1, 9]$$

6.4 Operations on sets and lists

6.4.1 Defining sets

Sets and lists are both collections of elements, and so have some operations in common. But lists are different from sets, because for a list, the order is important and the same element can be repeated in a list, while for sets order in not important and each element is unique. See Section 6.3, p. 78 for operations on lists.

Recall (see Section 3.2.2, p. 34) that to define a set of elements, you can put the elements, separated by commas, within delimiters `%{` and `%}` or `set[` and `]`.

Also, (see Section 3.2.3, p. 35) to define a list of elements, you can put the elements, separated by commas, within delimiters `[` and `]`. Lists are also called vectors.

Examples

```
> set[1,2,3,4]
```

or:

```
>
```

$$[\![1, 2, 3, 4]\!]$$

```
> [1,2,5]
```

$$[1, 2, 5]$$

6.4.2 Testing if a value is in a list or a set

The `member` and `contains` commands determine whether or not an object is in a list; the difference between them is the order of the arguments (required for compatibility reasons).

- `member` takes two arguments:
  - c, a value.
  - L, a list or a set.
- `member`(c, L) returns 0 if c is not an element of L and otherwise returns n if c is in L and its first position has index $n - 1$. `0` if c is not in `L`.
- `contains` takes two arguments:
  - L, a list or a set.
  - c, a value.
- `contains`(L, c) returns 0 if c is not an element of L and otherwise returns n if c is in L and its first position has index $n - 1$. `0` if c is not in `L`.

Examples

```
> member(2,[0,1,2,3,4,2])
```

or:

```
> member(2,%{0,1,2,3,4,2%})
```

or:

```
> contains([0,1,2,3,4,2],2)
```

or:

```
> contains(%{0,1,2,3,4,2%},2)
```

$$3$$

6.4.3 Union of two sets or of two lists

The `union` operator is an infixed operator that finds the union of the elements of two sets or lists; the result will always be a set.

Examples

```
> set[1,2,3,4] union set[5,6,3,4]
```

or:

```
>
```

$$[\![1,2,3,4,5,6]\!]$$

```
> [1,2,3] union [2,5,6]
```

$$[\![1,2,3,5,6]\!]$$

6.4.4 Intersection of two sets or of two lists

The `intersect` operator is an infixed operator that finds the intersection of the elements of two sets or lists; the result will always be a set.

Examples

```
> set[1,2,3,4] intersect set[5,6,3,4]
```

or:

```
>
```

or:

```
> [1,2,3,4] intersect [5,6,3,4]
```

$$[\![3,4]\!]$$

6.4.5 Difference of two sets or of two lists

The `minus` operator is an infixed operator that finds the set difference of the elements of two sets or lists; the result will always be a set.

Examples

```
> set[1,2,3,4] minus set[5,6,3,4]
```

or:

```
>
```

or:

```
> [1,2,3,4] minus [5,6,3,4]
```

$$[\![1, 2]\!]$$

6.4.6 Cartesian products

Defining an n-tuple. To define an n-tuple (rather than a list of n objects), you can put the elements, separated by commas, inside the delimiters `tuple[` and `]`. For example:

```
> set[tuple[1,3,4],tuple[1,3,5],tuple[2,3,4]]
```

$$[\![\text{tuple}\,[1, 3, 4]\,, \text{tuple}\,[1, 3, 5]\,, \text{tuple}\,[2, 3, 4]]\!]$$

The Cartesian product of two sets. You can compute the Cartesian product of two sets with the infixed `*` operator. For example:

```
> set[1,2]*set[3,4]
```

$$[\![\text{tuple}[1, 3], \text{tuple}[1, 4], \text{tuple}[2, 3], \text{tuple}[2, 4]]\!]$$

```
> set[1,2]*set[3,4]*set[5,6]
```

$$[\![\text{tuple}[1, 3, 5], \text{tuple}[1, 3, 6], \text{tuple}[1, 4, 5], \text{tuple}[1, 4, 6], \text{tuple}[2, 3, 5], \text{tuple}[2, 3, 6], \text{tuple}[2, 4, 5], \text{tuple}[2, 4, 6]]\!]$$

6.5 Ranges of values

6.5.1 Definition of a range of values

The `..` is an infixed operator which sets a range of values; given two real numbers a and b, the range of values between them is denoted $a\,..\,b$.

Remark. Note that the order of the boundaries of the range is significant. For example, if you input

```
> B:=2..3; C:=3..2
```

then `B` and `C` are not equal; `B==C` returns `0`.

Examples

```
> 1..4
```

$$1..4$$

```
> 1.2..sqrt(2)
```

$$1.2..\sqrt{2}$$

Remark. Since `..` is an operator, the parts of an expression can be picked out of it (see Section 8.2.3, p. 149). In particular, the `left` and `right` commands find the left and right endpoints of a range (see Section 5.2.4, p. 56, Section 8.2.3, p. 149, Section 6.6.2, p. 100, Section 6.3.6, p. 80, Section 9.3.4, p. 183 and Section 9.3.5, p. 183 for other uses of `left` and `right`.) For example:

```
> left(2..5)
```

$$2$$

```
> right(2..5)
```

$$5$$

6.5.2 Center of a range of values

The `interval2center` command finds the midpoint of a range of values.

- `interval2center` takes R, a range of values interval or a list of ranges of values.
- `interval2center(`R`)` returns the center of this range or the list of centers of these ranges.

Examples

```
> interval2center(3..5)
```

$$4$$

```
> interval2center([2..4,4..6,6..10])
```

$$[3, 5, 8]$$

6.5.3 Ranges of values defined by their center

The `center2interval` command finds ranges of values determined by their centers.

- `center2interval` takes one mandatory argument and one optional argument:
  - L, a list of real numbers.
  - Optionally, c, a real number (by default $L[0] - (L[1] - L[0])/2$).
- `center2interval(`L $\langle, c\rangle$`)` returns a list of ranges of values; the midpoints of the elements of the lists are endpoints of the ranges; so, for example, the second range will go from $(L[0] + L[1])/2$ to $(L[1] + L[2])/2$, etc. By default the first and last range are centered on $L[0]$ and $L[-1]$. With an argument of c, however, the first range will begin at c.

Examples

```
> center2interval([3,5,8])
```

$$2.0..4.0, 4.0..6.5, 6.5..9.5$$

```
> center2interval([3,5,8],2.5)
```

$$2.5..4.0, 4.0..6.5, 6.5..9.5$$

6.6 Intervals

6.6.1 Defining intervals

An interval is a range of real numbers, whose end points will be floats with at least 15 significant digits. The i command creates intervals, with the arguments in square brackets.

- `i` takes two arguments: a and b, two real numbers.
- `i[`a, b`]` returns the interval between a and b.

  If $a > b$, then `i[`a, b`]` returns `i[evalf(`b`,15)-epsilon,evalf(`a`,15)+epsilon]` (see Section 2.5.7, p. 15, item 2.5.7).

Intervals can also be created by following a decimal number with a question mark. If the decimal number contains n digits, the interval will be centered at a and have width $2 \cdot 10^{-n}$.

Examples

```
> i[1,13/11]
```

[1.00000000000000..1.18181818181819]

```
> i[pi,sqrt(3)]
```

[1.73205080756886..3.14159265358980]

```
> 0.123?
```

[0.121999999999999..0.124000000000000]

```
> 789.123456?
```

[0.789123454999990e3..0.789123456999998e3]

6.6.2 Obtaining endpoints of an interval

The `left` and `right` commands can find the left and right endpoints of an interval. (See Section 8.2.3, p. 149, Section 6.3.6, p. 80, Section 6.5.1, p. 98, Section 9.3.4, p. 183 and Section 9.3.5, p. 183 for other uses of `left` and `right`.)

- `left` and `right` take one argument: I, an interval.
- `left(`I`)` returns the left endpoint of the interval I.
- `right(`I`)` returns the right endpoint of the interval I.

Examples

```
> left(i[2,5])
```

2.00000000000000

```
> right(i[2,5])
```

5.00000000000000

6.6.3 Interval arithmetic

You can apply the usual arithmetic operators, such as `+`, `-`, `*` and `/`, to intervals.

The result of adding two intervals is the interval whose endpoints are the sums of the left end points and the right end points. For example:

```
> i[1,4]+i[2,3]
```

$$[3.00000000000000..7.00000000000000]$$

The negative of an interval is the result of taking the negative of the end points of the interval. The new end points will have to be switched. For example:

```
> -i[2,3]
```

$$[-3.00000000000000..-2.00000000000000]$$

The product of two intervals is the interval whose endpoints are the product the left endpoints of the two intervals and the product of the right endpoints of the two intervals. The smallest product will be the left end point of the product interval, and the largest product will be the right end point of the product interval. For example:

```
> i[1,4]*i[2,3]
```

$$[2.00000000000000..0.120000000000000e2]$$

```
> i[-2,4]*i[3,5]
```

$$[-0.100000000000000e2..0.200000000000000e2]$$

The reciprocal of an interval is the interval determined by the reciprocals of the end points. For example:

```
> 1/i[2,3]
```

$$[0.333333333333333..0.500000000000000]$$

```
> 1/i[-6,-3]
```

$$[-0.333333333333333..-0.166666666666667]$$

If the original interval has zero as an end point, then the reciprocal interval will have plus or minus infinity as one of the end points. If one end point is positive and the other is negative, then the reciprocal will simply be the interval from -infinity to infinity. For example:

```
> 1/i[0,2]
```

$$[0.500000000000000..+\infty]$$

```
> 1/i[-1,0]
```

$$[-\infty..-1.00000000000000]$$

```
> 1/i[-2,3]
```

$$[-\infty..+\infty]$$

You can also, if you want, do the usual operations such as subtraction, division, powers and roots.

6.6.4 Midpoint of an interval

The `midpoint` operator finds the midpoint of an interval.

- `midpoint` takes I, an interval.
- `midpoint`(I) returns the midpoint of I.

Example

```
> midpoint(i[2,3])
```

2.50000000000000

6.6.5 Union of intervals

In Xcas, the union of two intervals is their convex hull. The `union` operator is a binary infixed operator that finds the union of two intervals.

Examples

```
> i[1,3] union i[2,4]
```

[1.00000000000000..4.00000000000000]

```
> i[2,4] union i[6,9]
```

[2.00000000000000..9.00000000000000]

6.6.6 Intersection of intervals

The `intersect` operator is a binary infixed operator that finds the intersection of two intervals.

Example

```
> i[1,3] intersect i[2,4]
```

[2.00000000000000..3.00000000000000]

6.6.7 Testing if an object is in an interval

The `contains` command determines if an object is in an interval.

- `contains` takes two arguments:
  - I, an interval.
  - *obj*, an object.
- `contains`(I, *obj*) returns 1 if *obj* is in I, meaning that either *obj* is a number which is contained in I or `obj` is an interval which is a subset of I. It returns 0 otherwise.

Examples

```
> contains(i[0,2],1)
```

$$1$$

```
> contains(i[0,2],3)
```

$$0$$

```
> contains(i[0,2],i[1,2])
```

$$1$$

6.6.8 Converting a number into an interval

The `convert` command (see Section 10.1.10, p. 195) can convert an expression which evaluates to a number to the smallest interval which contains the number.

- `convert` takes two mandatory arguments and one optional argument:
  - *expr*, a number which evaluates to a number.
  - *interval*, a reserved word.
  - Optionally, n, an integer greater than 15 giving the desired number of digits.
- `convert(`*expr*`,interval` $\langle, n \rangle$`)` returns the smallest interval containing the value of *expr*.

Examples

```
> convert(sin(3)+1,interval)
```

$$[1.14112000805985..1.14112000805990]$$

```
> convert(sin(3)+1,interval,20)
```

$$[1.14112000805986722 20..1.14112000805986722 22]$$

6.6.9 Converting box constraints from matrix to interval form

The `box_constraints` command can convert bounds of given variables from the matrix form to interval form.

- `box_constraints` takes two mandatory arguments:
  - *vars*, a list of variables $[x_1, x_2, \ldots, x_n]$.
  - *bnds*, a matrix with n rows and two columns in which the kth row contains the lower and upper bound l_k and u_k of x_k, respectively.
- `box_constraints(`*vars*`,`*bnds*`)` returns $(x_1{=}l_1..u_1, x_2{=}l_2..u_2, \ldots, x_n{=}l_n..u_n)$, i.e. the variables and their ranges contained in a single sequence.

Example

```
> box_constraints([x,y],[[-2,2],[-5,5]])
```

$$x = -2..2, y = -5..5$$

7 Numbers

7.1 Integers (and Gaussian Integers)

Xcas can manage integers with unlimited precision, such as the following (see Section 12.1.1, p. 268):

```
> factorial(100)
```

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000

Gaussian integers are numbers of the form $a + ib$, where a and b are in $\mathbb{Z}$. For most functions in this section, use Gaussian integers in place of integers.

7.1.1 GCD

The `gcd` command finds the greatest common divisor (GCD) of a set of integers or polynomials. (See also Section 11.2.5, p. 227 for polynomials.) It can be called with one or two arguments.

- `gcd` can take one argument: *seq*, a sequence or list of integers or polynomials.
- `gcd(`*seq*`)` returns the GCD of the elements of *seq*.
- Alternatively, `gcd` can take two arguments: s and t, two lists of the same length containing integers or polynomials (alternatively, a matrix m with two rows whose elements are integers or polynomials).
- `gcd(`s,t`)` (or `gcd(`m`)`) returns the list whose kth element is the GCD of the kth elements of s and t (or the kth column of m).

The `Gcd` command is the inert form of `gcd`; namely, it evaluates to `gcd`, for later evaluation.

Examples

With one argument:

```
> gcd(18,15)
```

3

```
> gcd(18,15,21,36)
```

3

```
> gcd([18,15,21,36])
```

3

```
> gcd(-5-12*i,11-10*i)
```

3 + 2i

With two arguments:

```
> gcd([6,10,12],[21,5,8])
```

or:

```
> gcd([[6,10,12],[21,5,8]])
```

$$[3, 5, 4]$$

Find the greatest common divisor of $4n + 1$ and $5n + 3$ when $n \in \mathbb{N}$.

```
> f(n):=gcd(4*n+1,5*n+3)
```

Then (in a program editor level):

```
essai(n):={
  local j,a,L;
  L:=NULL;
  for (j:=-n;j<n;j++) {
    a:=f(j);
    if (a!=1) {
      L:=L,[j,a];
    }
  }
  return L;
}
```

Now:

```
> essai(20)
```

$$[-16, 7], [-9, 7], [-2, 7], [5, 7], [12, 7], [19, 7]$$

From this information, a reasonable conjecture would be that $\gcd(4n + 1, 5n + 3) = 7$ if $n = 7k - 2$ for some $k \in \mathbb{Z}$ and $\gcd(4n + 1, 5n + 3) = 1$ otherwise.

Since $\gcd(a, b) = \gcd(a, b - ca)$ for integers a, b and c; we have $\gcd(4n+1, 5n+3) = \gcd(4n+1, 5n+3-(4n+1)) = \gcd(4n+1, n+2) = \gcd(4n+1-4(n+2), n+2) = \gcd(-7, n+2) = \gcd(7, n+2)$, and so $\gcd(4n + 1, 5n + 3) = 7$ if 7 divides $n + 2$, namely $n + 2 = 7k$ or $n = 7k - 2$, and $\gcd(4n + 1, 5n + 3) = 1$ otherwise. This proves the conjecture.

An inert form of GCD:

```
> Gcd(18,15)
```

$$\gcd(18, 15)$$

```
> eval(Gcd(18,15))
```

$$3$$

(See Section 9.1.1, p. 169.)

7.1.2 GCD of a list of integers

The `lgcd` command finds the GCD of a list of integers or polynomials.

- `lgcd` takes L, a list of integers (or polynomials).
- `lgcd(`L`)` returns the GCD of all the integers (or polynomials) in the list L.

Example

```
> lgcd([18,15,21,36])
```

$$3$$

7.1.3 Least common multiple

The `lcm` command finds the least common multiple (LCM) of a set of integers or polynomials. (See also Section 11.2.7, p. 229 for polynomials.)

- `lcm` can take *seq*, a sequence or list of integers or polynomials, as its only argument.
- `lcm(`*seq*`)` returns the LCM of the elements of s.
- Alternatively, `lcm` can take s and t, two lists of the same length containing integers or polynomials (alternatively, a matrix m with two rows whose elements are integers or polynomials).
- `lcm(`s,t`)` (or `lcm(`m`)`) returns the list whose kth element is the LCM of the kth elements of s and t (or the kth column of m).

Examples

```
> lcm(18,15)
```

$$90$$

```
> lcm(-5-12*i,11-10*i)
```

$$-53+8\mathrm{i}$$

```
> lcm(18,15,21,36)
```

$$1260$$

```
> lcm([18,15,21,36])
```

$$1260$$

```
> lcm([6,10,12],[21,5,8])
```

or:

```
> lcm([[6,10,12],[21,5,8]])
```

$$[42, 10, 24]$$

7.1.4 Decomposition into prime factors

The `ifactor` command factors an integer into its prime factors. (Note that a prime factor of a Gaussian integer is only determined up to a factor of ± 1 or $\pm\mathrm{i}$.)

- `ifactor` takes n, an integer or a list of integers.
- `ifactor(`n`)` returns n in factored form (or a list of the integers in factored form).

Examples

```
> ifactor(90)
```

$$5 \cdot 2 \cdot 3^2$$

```
> ifactor(-90)
```

$$-5 \cdot 2 \cdot 3^2$$

```
> ifactor(14+23*i)
```

$$\mathrm{i}\,(2-\mathrm{i})^2\,(5+2\mathrm{i})$$

```
> ifactor([36,52])
```

$$\left[2^2 \cdot 3^2, 13 \cdot 2^2\right]$$

7.1.5 List of prime factors

The `ifactors` command decomposes an integer into prime factors.

- `ifactors` takes n, an integer or a list of integers.
- `ifactors(`n`)` decomposes the integer n (or the integers of the list) into prime factors, given as a list (or a list of lists) in which each prime factor is followed by its multiplicity.

Examples

```
> ifactors(90)
```

$$[2, 1, 3, 2, 5, 1]$$

since $90 = 2^1 \cdot 3^2 \cdot 5^1$.

```
> ifactors(-90)
```

$$[-1, 1, 2, 1, 3, 2, 5, 1]$$

```
> ifactors(31+22*i)
```

$$[\mathrm{i}, 1, 2-\mathrm{i}, 1, 4-\mathrm{i}, 2]$$

```
> ifactors([36,52])
```

$$\begin{bmatrix} 2 & 2 & 3 & 2 \\ 2 & 2 & 13 & 1 \end{bmatrix}$$

7.1.6 Matrix of factors

The `maple_ifactors` command decomposes an integer into prime factors and returns the result in Maple syntax.

- `maple_ifactors` takes n, an integer or a list of integers.
- `maple_ifactors(`n`)` decomposes the integer n (or the integers of the list) into prime factors, given as a list following the Maple syntax; namely, a list starting with $+1$ or -1 (for the sign), then a matrix with 2 columns whose rows are the prime factors and their multiplicity (or a list of such lists).

Examples

```
> maple_ifactors(90)
```

$$\begin{bmatrix} 1, & \begin{bmatrix} 2 & 1 \\ 3 & 2 \\ 5 & 1 \end{bmatrix} \end{bmatrix}$$

```
> maple_ifactor([36,52])
```

$$\begin{bmatrix} 1 & \begin{bmatrix} 2 & 2 \\ 3 & 2 \end{bmatrix} \\ 1 & \begin{bmatrix} 2 & 2 \\ 13 & 1 \end{bmatrix} \end{bmatrix}$$

7.1.7 Divisors of a number

The `idivis` or `divisors` command finds the divisors of an (ordinary) integer.

- `idivis` takes n, an integer or list of integers.
- `idivis`(n) returns the list of the divisors of the integer n (or of a list of such lists).

Examples

```
> idivis(36)
```

$$[1, 2, 3, 4, 6, 9, 12, 18, 36]$$

```
> idivis([36,22])
```

$$[[1, 2, 3, 4, 6, 9, 12, 18, 36], [1, 2, 11, 22]]$$

7.1.8 Integer Euclidean quotient

The quotient and remainder of ordinary integers a and b are respectively integers q and r, where $a = b\,q + r$ and $0 \leq r < b$.

The quotient and remainder of Gaussian integers a and b are respectively Gaussian integers q and r where $r = a - b\,q$ is as small as possible. It can be proven that r can be found so that $|r|^2 \leq |b|^2/2$.

The `iquo` or `intDiv` command finds the integer quotient of two integers.

- `iquo` takes two arguments: a and b, integers.
- `iquo`(a, b) returns the quotient q of a and b.

The `div` operator is the infixed version of `iquo`.

Examples

```
> iquo(148,5)
```

29

```
> iquo(factorial(148),factorial(145)+2)
```

3176375

```
> iquo(25+12*i,5+7*i)
```

$3 - 2\mathrm{i}$

Here $r = a - b\,q = -4 + i$ and $|-4+\mathrm{i}|^2 = 17 < |5+7\,\mathrm{i}|^2/2 = 74/2 = 37$.

```
> 148 div 5
```

29

7.1.9 Integer Euclidean remainder

The `irem` or `remain` command finds the remainder of two integers (see Section 7.1.8, p. 108).

- `irem` takes two arguments: a and b, integers.
- `irem`(a, b) returns the remainder r of a divided by b.

Examples

```
> irem(148,5)
```

3

```
> irem(factorial(148),factorial(45)+2)
```

1116153397282299330183389178030083019921209420472396393 12

```
> irem(25+12*i,5+7*i)
```

$-4 + \mathrm{i}$

Here $r = a - b\,q = -4 + \mathrm{i}$ and $|-4+\mathrm{i}|^2 = 17 < |5+7\,\mathrm{i}|^2/2 = 74/2 = 37$.

The `smod` or `mods` command finds the symmetric remainder of two (ordinary) integers.

- `smod` takes two arguments: a and b, integers.
- `smod`(a, b) returns the symmetric remainder s of the Euclidean division of a and b; namely, the value s with $a = b\,q + s$ and $-b/2 < s \leq b/2$.

Example

```
> smod(148,5)
```

-2

The `mod` or `%` operator is an infixed operator which takes an integer to a modular integer.

- `mod` has two operands: a and b, ordinary integers.
- a `mod` b returns $r\%b$ in $\mathbb{Z}/b\mathbb{Z}$, where r is the remainder of the Euclidean division of the arguments a and b.

Example

```
> 148 mod 5
```

or:

```
> 148 % 5
```

$$(-2) \,\%\, 5$$

Note that the result `-2 % 5` is not an integer (-2) but an element of $Z/5Z$ (see Section 11.8, p. 253 for the possible operations in $Z/5Z$).

7.1.10 Euclidean quotient and Euclidean remainder of two integers

The `iquorem` command finds both the quotient and remainder of two integers (see Section 7.1.8, p. 108).

- `iquorem` takes two arguments: a and b, integers.
- `iquorem`(a, b) returns the list $[q, r]$, where q is the quotient and r the remainder of a divided by b.

Examples

```
> iquorem(148,5)
```

$$[29, 3]$$

```
> iquorem(25+12*i,5+7*i)
```

$$[3 - 2\mathrm{i}, -4 + \mathrm{i}]$$

7.1.11 Testing evenness

The `even` command tests an integer to see if it is even. (A Gaussian integer $a + \mathrm{i}b$ is even exactly when a and b are both even and odd otherwise.)

- `even` takes n, an integer.
- `even`(n) returns 1 if n is even and 0 if n is odd.

Examples

```
> even(148)
```

$$1$$

```
> even(149)
```

$$0$$

```
> even(2+4*i)
```

$$1$$

7.1.12 Testing oddness

The `odd` command tests an integer to see if it is odd.

- `odd` takes n, an integer.
- `odd(`n`)` returns 1 if n is odd and returns `0` if n is even.

Examples

```
> odd(148)
```

0

```
> odd(149)
```

1

7.1.13 Testing pseudo-primality

A *pseudo-prime* is a number with a large probability of being prime[1]. For numbers less than 10^{14}, pseudo-prime and prime are equivalent.

The `is_pseudoprime` command is a test for a pseudo-prime.

- `is_pseudoprime` takes n, an integer.
- `is_pseudoprime(`n`)` returns 0, 1 or 2.
  - If it returns 0, then n is not prime.
  - If it returns 1, then n is a prime.
  - If it returns 2, then n is pseudo-prime (most probably prime).

Examples

```
> is_pseudoprime(100003)
```

1

```
> is_pseudoprime(9856989898997)
```

2

```
> is_pseudoprime(14)
```

0

```
> is_pseudoprime(98569898989977897 89)
```

1

[1] cf. Rabin's Algorithm and Miller-Rabin's Algorithm in the Algorithmic part (menu Help ▶ Manuals ▶ Programming).

7.1.14 Testing primality

The `is_prime`, `isprime` and `isPrime` commands are tests for primality.

- `is_prime` takes n, an integer.
- `is_prime(`n`)` returns 1 if n is prime and 0 if n is not prime.

`isprime` and `isPrime` are the same as `is_prime`, except they return `true` or `false`.

Examples

```
> is_prime(100003)
```

1

```
> isprime(100003)
```

true

```
> is_prime(98569898989987)
```

1

```
> is_prime(14)
```

0

```
> isprime(14)
```

false

Use the command `pari("isprime",n,1)` (see Section 7.2.10, p. 128) to get a primality certificate (see the documentation PARI/GP with the menu Help ▶ Manuals ▶ PARI-GP) and `pari("isprime",n,2)` to use the APRCL test.

Examples

```
> isprime(9856989898997789789)
```

true

```
> pari("isprime",9856989898997789789,1)
```

$$\begin{bmatrix} 2 & 2 & 1 \\ 19 & 2 & 1 \\ 941 & 2 & 1 \\ 1873 & 2 & 1 \end{bmatrix}$$

which are the coefficients giving the proof of primality by the $p-1$ Selfridge-Pocklington-Lehmer test.

7.1.15 Smallest pseudo-prime greater than n

The `nextprime` command finds pseudo-primes larger than a given target.

- `nextprime` takes n, an integer.
- `nextprime(`n`)` returns the smallest pseudo-prime (or prime) greater than n.

Example

```
> nextprime(75)
```

$$79$$

7.1.16 Greatest pseudo-prime less than n

The `prevprime` command finds pseudo-primes less than a given target.

- `prevprime` takes n, an integer greater than 2.
- `prevprime(`n`)` returns the largest pseudo-prime (or prime) less than n.

Example

```
> prevprime(75)
```

$$73$$

7.1.17 The nth pseudo-prime number

The `ithprime` command finds pseudo-primes.

- `ithprime` takes n, a positive integer.
- `ithprime(`n`)` returns the nth pseudo-prime number.

Examples

```
> ithprime(75)
```

$$379$$

```
> ithprime(k)$(k=1..20)
```

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71$$

7.1.18 Counting pseudo-primes less than or equal to n

The `nprimes` command counts the number of pseudo-primes less than or equal to a given target.

- `nprimes` takes n, a non-negative integer.
- `nprimes(`n`)` returns the number of pseudo-primes (or primes) less than or equal to n.

Examples

```
> nprimes(5)
```

$$3$$

```
> nprimes(10)
```

$$4$$

7.1.19 Bézout's identity

Bézout's Identity states that for any integers a and b, there exist integers u and v such that $\gcd(a,b) = au + bv$. The `iegcd` or `igcdex` command computes the coefficients u and v.

- `iegcd` takes two arguments: a and b, integers.
- `iegcd(`a, b`)` returns the list $[u, v, d]$, where $au + bv = d$ and $d = \gcd(a, b)$.

Example

```
> iegcd(48,30)
```

$$[2, -3, 6]$$

In other words, $2 \cdot 48 + (-3) \cdot 30 = 6$.

7.1.20 Chinese remainders

Theorem 1 (Chinese remainders)**.** *If $p_1, p_2, \ldots, p_n$ are relatively prime, then for any integers $a_1, a_2, \ldots, a_n$ there is a number c such that $c \equiv a_1 \pmod{p_1}$, $c \equiv a_2 \pmod{p_2}, \ldots, c \equiv a_n \pmod{p_n}$.*

The `ichinrem` or `ichrem` command finds this value of c.

- `ichinrem` takes one or more arguments: Each argument is a pair of integers a_k and p_k either as a list $[a_k, p_k]$ or as a modular integer $a_k \% p_k$.
- `ichinrem(`$[a_1, p_1], [a_2, p_2], \ldots, [a_n, p_n]$`)` if possible returns a list $[c, L]$, where $L = \operatorname{lcm}(p_1, p_2, \ldots, p_n)$ and c satisfies $c \equiv a_k \pmod{p_k}$ for $k = 1, \ldots, n$.

Note that any multiple of $L = \operatorname{lcm}(p_1, p_2, \ldots, p_n)$ can be added to c and the equalities will still be true. If the p_k are relatively prime, then a solution c exists by Theorem 1; what's more, any two solutions will be congruent modulo the product of the p_ks.

If all of the arguments are given as modular integers, then the result will also be given as a modular integer $c \% l$.

The `chrem` command does the same thing as `ichinrem`, but the input is given in a different form.

- `chrem` takes two arguments: $[a_1, \ldots, a_n]$ and $[p_1, \ldots, p_n]$, lists of integers of the same size.
- `chrem(`$[a_1, \ldots, a_n], [p_1, \ldots, p_n]$`)` returns $[c, L]$, as for `ichinrem`.

Be careful with the order of the parameters. Indeed, `chrem([a,b],[p,q])`, `ichrem([a,p],[b,q])` and `ichinrem([a,p],[b,q])` all produce the same output.

Examples

Solve:

$$\begin{aligned} x &\equiv 3 \pmod{5} \\ x &\equiv 9 \pmod{13} \end{aligned}$$

```
> ichinrem([3,5],[9,13])
```

or:

```
> ichrem([3,5],[9,13])
```

$$[48, 65]$$

so $x \equiv 48 \pmod{65}$.

You can also input:

```
> ichrem(3%5,9%13)
```

$$(-17) \,\%\, 65$$

Note that $48 \equiv -17 \pmod{65}$.

Recalling that `chrem` takes its arguments in a different form, you can also enter:

```
> chrem([3,9],[5,13])
```

$$[48, 65]$$

Solve:

$$\begin{aligned} x &\equiv 3 \pmod{5} \\ x &\equiv 4 \pmod{7} \\ x &\equiv 1 \pmod{9} \end{aligned}$$

```
> ichinrem([3,5],[4,7],[1,9])
```

$$[298, 315]$$

hence $x \equiv 298 \pmod{315}$.

Alternative input:

```
> ichinrem([3%5,4%7,1%9])
```

$$(-17) \,\%\, 315$$

Note that $298 \equiv -17 \pmod{315}$.

Again, with the arguments in a different form, you can also enter:

```
> chrem([3,4,1],[5,7,9])
```

$$[298, 315]$$

Remark. These three commands, `ichinrem`, `ichrem` and `chrem`, may also be used to find the coefficients of a polynomial whose equivalence classes are known modulo several integers by using polynomials with integer coefficients instead of integers for the a_k.

For example, to find $ax + b$ modulo $315 = 5 \cdot 7 \cdot 9$ under the assumptions

$$\begin{aligned} a &\equiv 3 \pmod{5} \\ a &\equiv 4 \pmod{7} \\ a &\equiv 1 \pmod{9} \end{aligned}$$

and

$$\begin{aligned} b &\equiv 1 \pmod{5} \\ b &\equiv 2 \pmod{7} \\ b &\equiv 3 \pmod{9} \end{aligned}$$

Example

```
> ichinrem((3x+1)%5,(4x+2)%7,(x+3)%9)
```

$$((-17) \,\%\, 315)\, x + 156 \,\%\, 315$$

hence `a=-17 (mod 315)` and `b=156 (mod 315)`.

As before, `chrem` takes the same input in a different format.

```
> chrem([3x+1,4x+2,x+3],[5,7,9])
```

$$[298x + 156, 315]$$

Note that $298 \equiv -17 \pmod{315}$.

7.1.21 Solving $au + bv = c$ in $\mathbb{Z}$

The `iabcuv` solves a linear Diophantine equation in two variables.

- The `iabcuv` command takes three arguments: a, b and c, integers.
- `iabcuv`(a, b, c) returns the list $[u, v]$ where $au + bv = c$.

Note that c must be a multiple of $\gcd(a, b)$ for the existence of a solution.

Example

```
> iabcuv(48,30,18)
```

$$[6, -9]$$

7.1.22 Solving $a^2 + b^2 = p$ in $\mathbb{Z}$

Any prime number congruent to 1 modulo 4 can be written as a sum of two squares. The `pa2b2` command finds such a decomposition.

- `pa2b2` takes p, a prime number which is congruent to 1 modulo 4.
- `pa2b2`(p) returns a list of integers $[a, b]$, where $p = a^2 + b^2$.

Example

```
> pa2b2(17)
```

$$[4, 1]$$

indeed, $17 = 4^2 + 1^2$.

7.1.23 Solving Diophantine equations

The `isolve` command attempts to solve the given equations over the integers. Note that it automatically solves for all of the indeterminates present in the equations.

- `isolve` takes one mandatory argument and two optional arguments:
  - *eq*, an equation or list of equations.

  - Optionally, *symb*, a (sequence or list of) symbol(s) which are used as the names for global variables present in the solution. These names default to _Z0, _Z1, ... for general integers and to _N0, _N1, ... for positive integers.
  - Optionally, `seq=false`, which makes `isolve` return only particular/fundamental solution(s) found by the solver. By default, `seq=true`, which makes `isolve` return sequences (classes) of solutions whenever possible.

- `isolve` can solve the following types of equations:
  - (systems of) linear equation(s).
  - general quadratic equations with two indeterminates.
  - equations of the type $Q(x, y, z) = 0$, where Q is a ternary quadratic form.
  - equations of the type $f(x) = g(y)$, where $f, g \in \mathbb{Z}[X]$ are monic polynomials with degrees m and n such that $\gcd(m, n) > 1$ and $f(x) - g(y)$ is irreducible in $\mathbb{Q}[X, Y]$.

Examples

Linear equations and systems can be solved.

```
> isolve(5x+42y+8=0)
```

$$[x = -10 + 42\_Z0, y = 1 - 5\_Z0]$$

```
> isolve([x+y-z=4,x-2y+3z=3],m)
```

$$[x = m, y = -4m + 15, z = -3m + 11]$$

To find the general solution to Pell-type equation $x^2 - 23y^2 = 1$:

```
> sol:=isolve(x^2-23y^2=1,n)
```

$$\left[x = \frac{\left(24 + 5\sqrt{23}\right)^n + \left(24 - 5\sqrt{23}\right)^n}{2}, y = \frac{\left(24 + 5\sqrt{23}\right)^n - \left(24 - 5\sqrt{23}\right)^n}{2\sqrt{23}}\right]$$

To check that it is indeed the solution, enter:

```
> simplify(subs(x^2-23y^2-1,sol))
```

$$0$$

Now to obtain e.g. the first four solutions, enter:

```
> simplify(apply(unapply(apply(rhs,sol),n),[1,2,3,4]))
```

$$[[24, 5], [1151, 240], [55224, 11515], [2649601, 552480]]$$

To obtain only the fundamental solution, enter:

```
> isolve(x^2-23y^2=1,seq=false)
```

$$[x = 24, y = 5]$$

The following two examples demonstrate solving quadratic equations with two indeterminates.

```
> isolve(x^2-5x*y+4y^2=16)
```

$$\begin{bmatrix} x=-4, & y=-5 \\ x=0, & y=-2 \\ x=4, & y=0 \\ x=10, & y=2 \\ x=21, & y=5 \\ x=4, & y=5 \\ x=0, & y=2 \\ x=-4, & y=0 \\ x=-10, & y=-2 \\ x=-21, & y=-5 \end{bmatrix}$$

```
> isolve(x^2-3x*y+y^2-x=2,n)
```

$$[[x=\frac{\left(\frac{\sqrt{5}-3}{2}\right)^n\left(-15\sqrt{5}-33\right)}{10}+\frac{\left(\frac{-\sqrt{5}-3}{2}\right)^n\left(15\sqrt{5}-33\right)}{10}-\frac{2}{5}, \\ y=\frac{\left(\frac{\sqrt{5}-3}{2}\right)^n\left(-3\sqrt{5}-6\right)}{5}+\frac{\left(\frac{-\sqrt{5}-3}{2}\right)^n\left(3\sqrt{5}-6\right)}{5}-\frac{3}{5}]]$$

```
> isolve(8x^2-24x*y+18y^2+5x+7y+16=0)
```

$$\begin{bmatrix} x=-174\_Z1^2+17\_Z1-2, & y=-116\_Z1^2+21\_Z1-2 \\ x=-174\_Z1^2+41\_Z1-4, & y=-116\_Z1^2+37\_Z1-4 \end{bmatrix}$$

Integral zeros of ternary quadratic forms can be found.

```
> isolve(x^2+11y^2+6x*y-3z^2=0,a,b,c)
```

$$\left[x=c\left(-44a^2+12b^2\right), y=c\left(13a^2-3b^2-6ab\right), z=c\left(-11a^2-3b^2+2ab\right)\right]$$

The components of the above solution can be divided by the GCD of $-44a^2+12b^2$, $13a^2-3b^2-6ab$, and $-11a^2-3b^2+2ab$, thus producing a parametrization for the pairwise-coprime solutions given $c=1$.

Certain polynomial equations of the type $f(x)=g(y)$ can be fully solved, as in the following example.

```
> isolve(x^2-3x+5=y^8-y^7+9y^6-7y^5+4y^4-y^3)
```

$$\begin{bmatrix} x=-3, & y=-1 \\ x=6, & y=-1 \\ x=0, & y=1 \\ x=3, & y=1 \\ x=660, & y=5 \\ x=-657, & y=5 \end{bmatrix}$$

The above list contains all integer solutions to the given equation.

7.1.24 Euler indicatrix

The *Euler phi* function (also called the *Euler totient* function) finds the number of positive integers less than a given integer and relatively prime. The `euler` command computes the Euler phi function.

- `euler` takes n, a non-negative integer.
- `euler(`n`)` returns the number of integers larger than 1, less than n and relatively prime to n.

Example

```
> euler(21)
```

12

In other words the set of integers less than 21 and coprime with 21, $\{1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20\}$, has 12 elements.

Theorem 2 (Fermat). *If p is a prime number, then for any integer a, $a^{p-1} \equiv 1 \pmod{p}$.*

Euler introduced his phi function to generalize Theorem 2.

Theorem 3 (Euler). *If a and n are relatively prime, then $a^{\texttt{euler}(n)} \equiv 1 \pmod{n}$.*

Example

```
> powmod(5,12,21)
```

1

(See Section 11.8.10, p. 257.)

7.1.25 Legendre symbol

If n is prime, the Legendre symbol of $a \in \mathbb{Z}$ is written $\left(\frac{a}{n}\right)$ and defined by:

$$\left(\frac{a}{n}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{n} \\ 1 & \text{if } a \not\equiv 0 \pmod{n} \text{ and } a \equiv b^2 \pmod{n} \\ -1 & \text{if } a \not\equiv 0 \pmod{n} \text{ and } a \not\equiv b^2 \pmod{n} \end{cases}$$

The Legendre symbol satisfies the following properties.

- If n is prime, then $a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}$.
- The following holds for odd and positive numbers p and k:

$$\left(\frac{p}{q}\right).\left(\frac{q}{p}\right) = (-1)^{\frac{p-1}{2}.\frac{q-1}{2}}, \quad \left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}, \quad \left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}.$$

The `legendre_symbol` command computes the Legendre symbol.

- `legendre_symbol` takes two arguments: a and n, integers.
- `legendre_symbol`(a, n) returns the Legendre symbol $\left(\frac{a}{n}\right)$.

Examples

```
> legendre_symbol(26,17)
```

1

```
> legendre_symbol(27,17)
```

−1

```
> legendre_symbol(34,17)
```

0

7.1.26 Jacobi symbol

The Jacobi symbol is a generalization of the Legendre symbol $\left(\frac{a}{n}\right)$ when n is not prime. Let

$$n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$$

be the prime factorization of n. The Jacobi symbol of a is defined by:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k}$$

Where the left hand side is the Jacobi symbol and the right hand side contains Legendre symbols. The `jacobi_symbol` command computes the Jacobi symbol.

- `jacobi_symbol` takes two arguments: a and n, integers.
- `jacobi_symbol(`a, n`)` returns the Jacobi symbol $\left(\frac{a}{n}\right)$.

Examples

```
> jacobi_symbol(25,12)
```

$$1$$

```
> jacobi_symbol(35,12)
```

$$-1$$

```
> jacobi_symbol(33,12)
```

$$0$$

7.1.27 Listing all compositions of an integer into k parts

A composition of a positive integer n is an ordered set of non-negative integers which sum to n. For example, three compositions of 4 are

$$4 = 1 + 3,$$
$$4 = 3 + 1,$$
$$4 = 1 + 1 + 2.$$

These compositions have two, two and three elements, respectively. The `icomp` command finds all compositions of an integer with a given number of elements.

- `icomp` accepts two mandatory arguments and one optional argument:
  - n, a positive integer.
  - k, a positive integer not larger than n.
  - Optionally, either `zeros=true` (which is the default) or `zeros=false`.
- `icomp(`n, k ⟨`,zeros=`*bool*⟩`)` returns the list of all compositions of n into k parts, where a part can be 0 if *bool* = `true` (the default), or where each part is greater than 0 if *bool* = `false`.

Remark. Using `icomp` with too large values of n can easily clutter your working memory because the number of compositions rises exponentially.

Examples

```
> icomp(4,2)
```

$$\begin{bmatrix} 4 & 0 \\ 3 & 1 \\ 2 & 2 \\ 1 & 3 \\ 0 & 4 \end{bmatrix}$$

```
> icomp(5,3,zeros=false)
```

$$\begin{bmatrix} 3 & 1 & 1 \\ 2 & 2 & 1 \\ 1 & 3 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \\ 1 & 1 & 3 \end{bmatrix}$$

7.1.28 Day of the week

The `dayofweek` command finds the day of the week for any date after 15 October 1582.

- `dayofweek` takes as arguments three arguments:
  - d, an integer representing the day of the month.
  - m, an integer representing the month.
  - y, an integer representing the year.

  The date represented should be after 15 October 1582.
- `dayofweek`(d, m, y) returns an integer from 0 to 6 ($0 =$ Sunday, $1 =$ Monday, $\dots$).

Examples

```
> dayofweek(1,10,2014)
```

$$3$$

This means that 1 October, 2014 was a Wednesday.

```
> dayofweek(15,10,1582)
```

$$5$$

This indicates that 15 October 1582 was on a Friday.

Historical note. The Gregorian calendar, the calendar used by most of the world, was introduced on 15 October 1582. Before that, the Julian calendar was used, which had a leap year every four years and so used years with an average of 365.25, which is slightly off from the actual value of about 365.242 days. To deal with this, the Gregorian calendar was introduced, where a leap year is a year which is divisible by 4, but not divisible by 100 unless it is also divisible by 400. This gives an average length of year that is accurate to within 1 day every 3000 years. Many countries switched from the Julian calendar to the Gregorian calendar after 4 October 1582 in the Julian calendar, and the next day was 15 October 1582.

7.2 Rational numbers

7.2.1 Transform a floating point number into a rational

Rational numbers can be approximated by floating point numbers, but since floating point numbers are not exact, they cannot typically be converted back to the original rational number. However, the `float2rational` or `exact` command will try convert a floating point to a nearby rational number.

- `float2rational` takes d, a floating point number.
- `float2rational(`d`)` returns a rational number q close to d; namely such that $|d - q| <$ epsilon, where epsilon is defined in the CAS configuration (Cfg menu, see Section 2.5.7, p. 15, item 2.5.7) or with the `cas_setup` command (see Section 2.5.10, p. 18).

Examples

```
> float2rational(0.3670520231)
```

Output for `epsilon=1e-10`:

$$\frac{127}{346}$$

```
> evalf(363/28)
```

$$12.9642857143$$

```
> float2rational(12.9642857143)
```

$$\frac{363}{28}$$

If two representations are mixed, for example:

```
> 1/2+0.7
```

the rational is converted to a float.

$$1.2$$

```
> 1/2+float2rational(0.7)
```

$$\frac{6}{5}$$

7.2.2 Integral and fractional part of a rational number

Rational numbers are often broken up into integer and fractional parts, where the fractional part has absolute value less than 1; i.e., the absolute value of the top integer is smaller than that of the bottom integer. Such a fraction is called a *proper* fraction. The `propfrac` or `propFrac` command writes a fraction as an integer plus a proper fraction.

- `propfrac` takes r, a rational number.
- `propfrac(`r`)` returns

$$q + \frac{r}{b} \quad \text{with} \quad 0 \le r < b$$

where $r = \frac{a}{b}$ is in lowest terms and and $a = b\,q + r$.

(For rational expressions, see Section 11.6.8, p. 251.)

Examples

```
> propfrac(42/15)
```

$$2 + \frac{4}{5}$$

```
> propfrac(43/12)
```

$$3 + \frac{7}{12}$$

7.2.3 Numerator of a fraction after simplification

The `numer` or `getNum` command finds the numerator of a fraction.

- `numer` takes r, a fraction.
- `numer`(r) returns the numerator of r after it has been reduced to lowest terms. (For rational expressions, see Section 11.6.2, p. 249 and Section 11.6.1, p. 249.)

Examples

```
> numer(42/12)
```

or:

```
> getNum(42/12)
```

$$7$$

To avoid simplification, the argument must be quoted (see Section 9.1.4, p. 170).
(For rational fractions see Section 11.6.1, p. 249).

```
> numer('42/12')
```

or:

```
> getNum('42/12')
```

$$42$$

7.2.4 Denominator of a fraction after simplification

The `denom` or `getDenom` command finds the denominator of a fraction.

- `denom` takes r, a fraction.
- `denom`(r) returns the denominator of r after it has been reduced to lowest terms. (For rational expressions see Section 11.6.4, p. 250).

Example

```
> denom(42/12)
```

or:

```
> getDenom(42/12)
```

$$2$$

To avoid simplification, the argument must be quoted (see Section 9.1.4, p. 170).
(For rational expressions see Section 11.6.3, p. 249).

```
> denom('42/12')
```

or:

```
> getDenom('42/12')
```

12

7.2.5 Numerator and denominator of a fraction

The `f2nd` or `fxnd` command finds the numerator and denominator of a fraction.

- `f2nd` takes r, a fraction.
- `f2nd(`r`)` returns the list of the numerator and denominator of r after it has been reduced to lowest terms. (For rational expressions see Section 11.6.5, p. 250).

Example

```
> f2nd(42/12)
```

$[7, 2]$

7.2.6 Simplifying a pair of integers

The `simp2` command reduces a fraction to lowest terms, where the fraction is given as a separate numerator and denominator. (See also Section 11.6.6, p. 250.)

- `simp2` takes one or two arguments: $[a, b]$, a list of two integers or simply the two integers a, b.
- `simp2([`a, b`])` or `simp2(`a, b`)` returns the integers after they have been divided by their greatest common divisor; i.e., the corresponding fraction will be in lowest terms.

Examples

```
> simp2(18,15)
```

$[6, 5]$

```
> simp2([42,12])
```

$[7, 2]$

7.2.7 Continued fraction representation of a real

Any real number a can be written as a continued fraction:

$$a = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

, which is often abbreviated $[a_0; a_1, a_2, a_3, \dots]$. The `dfc` command writes a real number as a continued fraction.

- `dfc` takes one mandatory argument and one optional argument:
  - a, a real number.
  - Optionally, n an integer or *epsilon*, a positive real number.
- `dfc(`a`)` returns the list of the continued fraction representation of `a` with precision `epsilon`, which is given by Section 2.5.7, p. 15, item 2.5.7.
- `dfc(`a`,`*epsilon*`)` returns the list of the continued fraction representation which approximates `a` or `evalf(a)` with the specified precision `epsilon`.
- `dfc(`a, n`)` returns the list of the continued fraction representation of `a` of order `n`.

Remarks.

- The `convert` command with the option `confrac` (see Section 10.1.10, p. 195) has a similar functionality: in that case the value of epsilon is the value defined in the CAS configuration and the answer may be stored in an optional third argument.
- If the last element of the result is a list, the representation is ultimately periodic, and the last element is the period. It means that the real is a root of an equation of order 2 with integer coefficients. So If `dfc(a)=[a0,a1,a2,[b0,b1]]` then:

$$a = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_0 + \cdots}}}}}.$$

- If the last element of the result is not an integer, it represents a remainder r. For example, if `dfc(a)` is equal to `[a0,a1,a2,r]`, then:

$$a = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \frac{1}{r}}}.$$

  Be aware that this remainder has lost most of its accuracy.

Examples

```
> dfc(sqrt(2),5)
```

$$[1, 2, [2]]$$

```
> dfc(evalf(sqrt(2)),1e-9)
```

or:

```
> dfc(sqrt(2),1e-9)
```

$$[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]$$

```
> convert(sqrt(2),confrac,'dev')
```

With epsilon = 1e−9 in the CAS configuration, we obtain:

$$[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]$$

and `[1,2,2,2,2,2,2,2,2,2,2,2,2]` is stored in `dev`.

```
> dfc(9976/6961,5)
```

$$\left[1, 2, 3, 4, 5, \frac{43}{7}\right]$$

Input to verify:

```
> 1+1/(2+1/(3+1/(4+1/(5+7/43))))
```

$$\frac{9976}{6961}$$

```
> convert(9976/6961,confrac,'l')
```

With epsilon = 1e−9 in the CAS configuration, we obtain:

$$[1, 2, 3, 4, 5, 6, 7]$$

and `[1,2,3,4,5,6,7]` is stored in `l`.

```
> dfc(pi,5)
```

$$\left[3, 7, 15, 1, 292, \frac{-113\pi + 355}{33102\pi - 103993}\right]$$

```
> dfc(evalf(pi),5)
```

If floats are hardware floats, e.g. for Digits=12, we obtain:

$$[3, 7, 15, 1, 292, 1.57581843574]$$

```
> dfc(evalf(pi),1e-9)
```

or:

```
> dfc(pi,1e-9)
```

or, with epsilon = 1e−9 in the CAS configuration:

```
> convert(pi,confrac,'ll')
```

$$[3, 7, 15, 1, 292]$$

and `[3,7,15,1,292]` is stored in `ll`.

7.2.8 Transforming a continued fraction representation into a real

The `dfc2f` command transforms a continued fraction into a real number.

- `dfc2f` takes L, a list representing a continued fraction, which can be:
  - a list of integers for a rational number.
  - a list whose last element is a list for an ultimately periodic representation, i.e. a quadratic number, that is a root of a second order equation with integer coefficients.
  - a list with a remainder r as last element ($a = a_0 + 1/\ldots + 1/a_n + 1/r$).
- `dfc2f(`L`)` returns the rational number or the quadratic number whose continued fraction representation is L.

Examples

```
> dfc2f([1,2,[2]])
```

$$\frac{1}{\frac{1}{\sqrt{2}+1}+2}+1$$

After simplification with `normal`, we obtain $\sqrt{2}$.

```
> dfc2f([1,2,3])
```

$$\frac{10}{7}$$

```
> normal(dfc2f([3,3,6,[3,6]]))
```

$$\sqrt{11}$$

```
> dfc2f([1,2,3,4,5,6,7])
```

$$\frac{9976}{6961}$$

Input to verify:

```
> 1+1/(2+1/(3+1/(4+1/(5+1/(6+1/7)))))
```

$$\frac{9976}{6961}$$

```
> dfc2f([1,2,3,4,5,43/7])
```

$$\frac{9976}{6961}$$

Input to verify:

```
> 1+1/(2+1/(3+1/(4+1/(5+7/43))))
```

$$\frac{9976}{6961}$$

7.2.9 Bernoulli numbers

The Bernoulli polynomial B_n is defined by:

$$B_0 = 1, \quad {B_n}'(x) = n\, B_{n-1}(x), \quad \int_0^1 B_n(x)\,\mathrm{d}x = 0$$

The nth Bernoulli number is $B_n = B_n(0)$, and is also given by the formula:

$$\frac{t}{\mathrm{e}^t - 1} = \sum_{n=0}^{+\infty} \frac{B(n)}{n!} t^n.$$

The `bernoulli` command computes the Bernoulli numbers.

- `bernoulli` takes n, an integer.
- `bernoulli(`n`)` returns the nth Bernoulli number, B_n.

Example

```
> bernoulli(6)
```

$$\frac{1}{42}$$

7.2.10 Access to PARI/GP commands

PARI/GP[2] is a computer algebra system which focuses on number theory. Xcas can use the PARI/GP functions with the `pari` command.

The arguments of `pari` depends on the PARI/GP function it is using.

- `pari` with a string as first argument (the PARI command name) executes the corresponding PARI command with the remaining arguments. For example `pari("weber",1+i)` executes the PARI command `weber(1+i)`.
- `pari` without any argument exports all PARI/GP functions to Xcas with the prefix `pari_`. If the name of a PARI function is not also the name of an Xcas command, that function will also be exported without the prefix.

For example, after calling `pari()`, the commands `pari_weber(1+i)` and `weber(1+i)` will execute the PARI command `weber(1+i)`.

The documentation of PARI/GP is available with the menu Help ▶ Manuals.

7.3 Real numbers

7.3.1 Evaluating a real at a given precision

A real number is an exact number and its numeric evaluation at a given precision is a floating number represented in base 2. The precision of a floating number is the number of bits of its mantissa, which is at least 53 (hardware float numbers, also known as `double`).

Floating numbers are displayed in base 10 with a number of digits controlled by the user either by assigning the Digits variable or by modifying the CAS configuration (see Section 2.5.7, p. 15, item 2.5.7). By default, Digits is equal to 12.

The number of digits displayed controls the number of bits of the mantissa; if Digits is less than 15, 53 bits are used, if Digits is strictly greater than 15, the number of bits is a roundoff of Digits times $\log_2(10)$.

An expression can be coerced into a floating number with the `evalf` command (see Section 7.3.1, p. 128). The `evalf` command may have an optional second argument which will specify the precision to use.

Note that if an expression contains a floating number, evaluation will try to convert other arguments to floating point numbers in order to coerce the whole expression to a single floating number.

Examples

```
> 1+1/2
```

$$\frac{3}{2}$$

```
> 1.0+1/2
```

$$1.5$$

[2] https://pari.math.u-bordeaux.fr/

```
> exp(pi*sqrt(20))
```

$$e^{2\pi\sqrt{5}}$$

```
> evalf(exp(pi*2*sqrt(5)))
```

$$1263794.75367$$

```
> 1.1^20
```

$$6.72749994932$$

```
> sqrt(2)^21
```

$$\sqrt{2}\cdot 2^{10}$$

Input for a result with 30 digits:

```
> Digits:=30
```

Input for the numeric value of $e^{\pi\sqrt{163}}$:

```
> evalf(exp(pi*sqrt(163)))
```

$$0.262537412640768743999999999985\times 10^{8}$$

Note that `Digits` is now set to 30. You could have entered:

```
> evalf(exp(pi*sqrt(163)),30)
```

if you didn't want to change the value of `Digits`.

7.3.2 Standard arithmetic operators for real numbers

The `+`, `-`, `*`, `/`, and `^` operators are the usual infixed operators to do addition, subtraction, multiplication, division and raising to a power.

Examples

```
> 3+2
```

$$5$$

```
> 3-2
```

$$1$$

```
> 3*2
```

$$6$$

```
> 3/2
```

$$\frac{3}{2}$$

```
> 3.2/2.1
```

$$1.52380952381$$

```
> 3^2
```

9

```
> 3.2^2.1
```

11.5031015682

Remark. Use the square key or the cube key if your keyboard has one; for example, 3^2 returns 9.

Remarks on non integral powers. If x is not an integer, then $a^x = \mathrm{e}^{x\ln(a)}$, hence if x is not rational, then a^x is well-defined only for $a > 0$. If x is rational and $a < 0$, the principal branch of the logarithm is used, leading to a complex number. Note the difference between $\sqrt[n]{a}$ and $a^{\frac{1}{n}}$ when n is an odd integer.

Example

To draw the graph of $y = \sqrt[3]{x^3 - x^2}$, input:

```
> plotfunc(ifte(x>0,(x^3-x^2)^(1/3),-(x^2-x^3)^(1/3)),x,xstep=0.01)
```

You might also input:

```
> plotimplicit(y^3=x^3-x^2)
```

but this is much slower and much less accurate.

7.3.3 Prefixed division on reals

The `rdiv` command is the prefixed form of the usual division operator.

Examples

```
> rdiv(3,2)
```

$$\frac{3}{2}$$

```
> rdiv(3.2,2.1)
```

1.52380952381

7.3.4 nth root

The `root` command finds roots of numbers.

- `root` takes two arguments: n and a, numbers.
- `root`(n, a) returns the nth root of a (i.e. $a^{1/n}$). If $a < 0$, the nth root is a complex number with argument $2\pi/n$.

Examples

```
> root(3,2)
```

$$2^{\frac{1}{3}}$$

```
> root(3,2.0)
```

$$1.25992104989$$

```
> root(3,sqrt(2))
```

$$2^{\frac{1}{6}}$$

7.3.5 Exponential integral function

The exponential integral Ei is defined for non-zero real numbers x by

$$\mathrm{Ei}(x) = \int_{-\infty}^{x} \frac{\mathrm{e}^t}{t}\,\mathrm{d}t.$$

For $x > 0$, this integral is improper but the principal value exists. Also, $\mathrm{Ei}(0) = -\infty$, $\mathrm{Ei}(-\infty) = 0$.

Since $\frac{\mathrm{e}^x}{x} = \frac{1}{x} + 1 + \frac{x}{2!} + \frac{x^2}{3!} + \cdots$, the Ei function can be extended to $\mathbb{C} \setminus \{0\}$ (with a branch cut on the positive real axis) by

$$\mathrm{Ei}(z) = \ln(z) + \gamma + x + \frac{x^2}{2 \cdot 2!} + \frac{x^3}{3 \cdot 3!} + \cdots$$

where $\gamma = 0.57721566490\ldots$ is the Euler-Mascheroni constant.

The `Ei` command takes one or two arguments.

With one argument, the `Ei` command computes the exponential integral.

- `Ei` takes z, a complex number.
- `Ei(`z`)` returns the value of the exponential integral at z.

Examples

```
> Ei(1.0)
```

$$1.89511781636$$

```
> Ei(-1.0)
```

$$-0.219383934396$$

```
> Ei(1.)-Ei(-1.)
```

$$2.11450175075$$

```
> int((exp(x)-1)/x,x=-1..1.)
```

$$2.11450175075$$

The following input approximates the Euler's constant γ:

```
> evalf(Ei(-1)-sum((-1)^n/n/n!,n=1..100))
```

$$0.577215664902$$

Another type of exponential integral is

$$E_1(x) = \int_x^{+\infty} \frac{\mathrm{e}^{-t}}{t}\,\mathrm{d}t = \int_1^{+\infty} \frac{\mathrm{e}^{-tx}}{t}\,\mathrm{d}t$$

which satisfies $E_1(x) = -\operatorname{Ei}(-x)$. This can be generalized to

$$E_n(x) = \int_1^{+\infty} \frac{\mathrm{e}^{-ntx}}{t}\,\mathrm{d}t.$$

These functions satisfy

$$\begin{aligned} E_1(x) &= -\operatorname{Ei}(x), \\ E_2(x) &= \mathrm{e}^{-x} + x\,\operatorname{Ei}(-x) = \mathrm{e}^{-x} - x\,E_1(x), \end{aligned}$$

and, for $n \geq 2$,

$$E_n(x) = \frac{\mathrm{e}^{-x} - x\,E_{n-1}(x)}{n-1}.$$

With two arguments, the `Ei` command computes the above version of the exponential integral.

- `Ei` takes two arguments:
  - z, a complex number.
  - n, a positive integer.
- `Ei(`z, n`)` returns the value of $E_n(z)$.

Examples

```
> Ei(1.0,1)
```

0.219383934396

```
> Ei(3.0,2)
```

0.0106419250853

7.3.6 Logarithmic integral function

The logarithmic integral function is defined by

$$\operatorname{Li}(x) = \operatorname{Ei}(\ln(x)) = \int_{t=0}^{\mathrm{e}^x} \frac{1}{\ln(t)}\,\mathrm{d}t$$

The `Li` command computes the logarithmic integral.

- `Li` takes z, a complex number.
- `Li(`z`)` returns the value of the logarithmic integral $\operatorname{Li}(z)$.

Example

```
> Li(2.0)
```

1.04516378012

7.3.7 Cosine integral function

The cosine integral function is defined by

$$\mathrm{Ci}(x) = \int_{+\infty}^{x} \frac{\cos(t)}{t}\,\mathrm{d}t = \ln(t) + \gamma + \int_{0}^{x} \frac{\cos(t)-1}{t}\,\mathrm{d}t.$$

It can be shown that $\mathrm{Ci}(0) = -\infty$, $\mathrm{Ci}(-\infty) = \mathrm{i}\pi$ and $\mathrm{Ci}(+\infty) = 0$.
The `Ci` command computes the cosine integral function.

- `Ci` takes z, a complex number.
- `Ci(`z`)` returns the value of the cosine integral function $\mathrm{Ci}(z)$.

Examples

```
> Ci(1.0)
```

$$0.337403922901$$

```
> Ci(-1.0)
```

$$0.337403922901 + 3.14159265359\mathrm{i}$$

```
> Ci(1.0)-Ci(-1.0)
```

$$-3.14159265359\mathrm{i}$$

7.3.8 Sine integral function

The sine integral function is defined by

$$\mathrm{Si}(x) = \int_{0}^{x} \frac{\sin(t)}{t}\,\mathrm{d}t.$$

It can be shown that $\mathrm{Si}(0) = 0$, $\mathrm{Si}(-\infty) = -\frac{\pi}{2}$ and $\mathrm{Si}(+\infty) = \frac{\pi}{2}$. Note that Si is an odd function.
The `Si` command computes the sine integral function.

- `Si` command takes z, a complex number.
- `Si(`z`)` returns the value of the sine integral function $\mathrm{Si}(z)$.

Example

```
> Si(1.0)
```

$$0.946083070367$$

```
> Si(-1.0)
```

$$-0.946083070367$$

7.3.9 Heaviside step function

The Heaviside function is the step function $H(x) = \begin{cases} 0 & \text{for } x < 0, \\ 1 & \text{for } x \geq 0. \end{cases}$

The `Heaviside` command computes the Heaviside function.

- `Heaviside` takes x, a real number.
- `Heaviside(`x`)` returns the value of the Heaviside function $H(x)$.

Examples

```
> Heaviside(2)
```

$$1$$

```
> Heaviside(-4)
```

$$0$$

7.3.10 Dirac distribution

The Dirac δ distribution is the distributional derivative of the Heaviside function. This means that

$$\int_{-\infty}^{+\infty} \delta(x)\,\mathrm{d}x = 1$$

and, in fact,

$$\int_a^b \delta(x)\,\mathrm{d}x = \begin{cases} 1 & \text{if } 0 \in [a,b], \\ 0 & \text{otherwise.} \end{cases}$$

The defining property of the Dirac distribution is that

$$\int_{-\infty}^{+\infty} \delta(x) f(x)\,\mathrm{d}x = f(0)$$

and consequently, for $c \in [a,b]$,

$$\int_a^b \delta(x-c) f(x)\,\mathrm{d}x = f(c).$$

The `Dirac` command represents the Dirac distribution.

- `Dirac` takes one mandatory argument and one optional argument:
  - x, a symbol or an expression.
  - Optionally, n, a nonnegative integer.
- `Dirac(`$x\,\langle, n\,\rangle$`)` returns $\delta^{(n)}(x)$, where $\delta^{(n)}$ is the nth derivative of δ.

Note that x can be a real number, for which `Dirac` returns 0 if $x \neq 0$ and ∞ otherwise. However, since δ is a distribution, not a function, computing its value at a point makes little sense.

Examples

```
> int(Dirac(x-1)*sin(x),x,-1,2)
```

$$\sin(1)$$

```
> int(Dirac(x-1,1)*sin(x),x,-inf,inf)
```

$$-\cos(1)$$

7.3.11 Error function

The error function erf is defined by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \mathrm{e}^{-t^2} \mathrm{d}t,$$

where the constant $\frac{2}{\sqrt{\pi}}$ is chosen so that $\operatorname{erf}(+\infty) = 1$ and $\operatorname{erf}(-\infty) = -1$, since

$$\int_0^{+\infty} \mathrm{e}^{-t^2} \mathrm{d}t = \frac{\sqrt{\pi}}{2}.$$

The `erf` command computes the error function.

- `erf` takes a, a number.
- `erf(`a`)` returns the value of $\operatorname{erf}(a)$.

Examples

```
> erf(1)
```

$$\operatorname{erf}(1)$$

```
> erf(1.0)
```

$$0.84270079295$$

```
> erf(1/(sqrt(2)))*1/2+0.5
```

$$0.841344746069$$

Remark. The relation between `erf` and `normal_cdf` (see Section 20.4.7, p. 541) is:

$$\texttt{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)$$

Indeed, making the change of variable $t = u\sqrt{2}$ in $\texttt{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^x \mathrm{e}^{-t^2/2} \mathrm{d}t$ yields:

$$\texttt{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} \mathrm{e}^{-u^2} \mathrm{d}u = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right).$$

7.3.12 Complementary error function

The complementary error function is defined by

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} \mathrm{e}^{-t^2} \mathrm{d}t = 1 - \operatorname{erf}(x).$$

Hence $\operatorname{erfc}(0) = 1$, since

$$\int_0^{+\infty} \mathrm{e}^{-t^2} \mathrm{d}t = \frac{\sqrt{\pi}}{2}.$$

The `erfc` command computes the complementary error function.

- `erfc` takes a, a number.
- `erfc(`a`)` returns the value of the complementary error function $\operatorname{erfc}(a)$.

Examples

```
> erfc(1)
```

$$1 - \operatorname{erf}(1)$$

```
> 1-erfc(1/(sqrt(2)))*0.5
```

$$0.841344746069$$

Remark. The relation between `erfc` and `normal_cdf` (see Section 20.4.7, p. 541) is:

$$\texttt{normal\_cdf}(x) = 1 - \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right).$$

7.3.13 Gamma function

The Gamma function is defined by

$$\Gamma(x) = \int_0^{+\infty} \mathrm{e}^{-t} t^{x-1} \mathrm{d}t, \text{ if } x > 0.$$

If x is a positive integer, Γ is computed by applying the recurrence $\Gamma(x+1) = x\,\Gamma(x)$ with $\Gamma(1) = 1$. Hence $\Gamma(n+1) = n!$ which is used to generalize the factorial (see Section 12.1.1, p. 268).

The `Gamma` command computes the Gamma function.

- `Gamma` takes a, a number.
- `Gamma(`a`)` returns the value $\Gamma(a)$.

Examples

```
> Gamma(5)
```

$$24$$

```
> Gamma(0.7)
```

$$1.29805533265$$

```
> Gamma(-0.3)
```

$$-4.32685110883$$

Indeed, $\Gamma(0.7) = -0.3 \cdot \Gamma(-0.3)$.

```
> Gamma(-1.3)
```

$$3.32834700679$$

Indeed, $\Gamma(0.7) = -0.3 \cdot \Gamma(-0.3) = -0.3 \cdot (-1.3) \cdot \Gamma(-1.3)$.

If $a = \frac{n}{d} \in \mathbb{Q} \setminus \mathbb{Z}$ where $d > 0$, then the exact value $\Gamma(a)$ is computed from $\Gamma\left(\frac{m}{d}\right)$, where $0 < 2m < d$ and either $m - n$ or $m + n$ is divisible by d. (If $d = 2$, then the value $\Gamma(a)$ does not involve another Gamma value.) In particular, this leads to simplification of certain products of Gamma values.

```
> Gamma(1/2)
```

$$\sqrt{\pi}$$

```
> Gamma(-15/8)
```

$$\frac{64\Gamma\left(\frac{1}{8}\right)}{105}$$

```
> normal(Gamma(-13/4)/Gamma(3/4))
```

$$\frac{256}{585}$$

```
> normal(Gamma(1/4)*Gamma(3/4))
```

$$\pi\sqrt{2}$$

7.3.14 Upper incomplete γ function

The upper incomplete γ function is defined by

$$\Gamma(a, b) = \int_b^{+\infty} \mathrm{e}^{-t}\, t^{a-1}\, \mathrm{d}t.$$

The `ugamma` command computes the upper incomplete γ function.

- `ugamma` takes two arguments:
  - a, a number.
  - b, a positive real number.
- `ugamma(`a, b`)` returns the value of $\Gamma(a, b)$.

Examples

```
> ugamma(3.0,2.0)
```

$$1.35335283237$$

```
> ugamma(-1.3,2)
```

$$0.0142127568837$$

7.3.15 Lower incomplete γ function

The lower incomplete γ function is defined by

$$\gamma(a,b) = \int_0^b \mathrm{e}^{-t}\, t^{a-1}\, \mathrm{d}t.$$

The `igamma` command computes the lower incomplete γ function.

- `igamma` takes two mandatory arguments and one optional argument:
  - a, a number.
  - b, a positive real number.
  - Optionally, the number `1`.
- `igamma`(a,b) returns $\gamma(a,b)$.
- `igamma`$(a,b$`,1)` returns a normalized version of the function; namely $\gamma(a,b)/\Gamma(a)$.

Examples

```
> igamma(4.0,3.0)
```

$$2.11660866731$$

```
> igamma(4.0,3.0,1)
```

$$0.352768111218$$

since $\Gamma(4) = 6$ and $2.11660866731/6 = 0.352768111218$.

7.3.16 Beta function

The β function is defined by

$$\beta(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1} = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)}.$$

This is defined for x and y positive reals (to ensure the convergence of the integral) and by extension for x and y if they are not negative integers. Notably, $\beta(1,1) = 1$, $\beta(n,1) = \frac{1}{n}$ and $\beta(n,2) = \frac{1}{n(n+1)}$.

The `Beta` command computes the β function.

- `Beta` takes two arguments: a and b, real numbers.
- `Beta`(a,b) returns the value of the $\beta(a,b)$.

Examples

```
> Beta(5,2)
```

$$\frac{1}{30}$$

```
> Beta(x,y)
```

$$\frac{\Gamma\left(x\right)\Gamma\left(y\right)}{\Gamma\left(x+y\right)}$$

```
> Beta(5.1,2.2)
```

$$0.0242053671402$$

7.3.17 Derivatives of the DiGamma function

The DiGamma function is the derivative of the logarithm of the Γ function (see Section 7.3.13, p. 136),

$$\psi(z) = \frac{\mathrm{d}}{\mathrm{d}z}\ln(\Gamma(z)) = \frac{\Gamma'(z)}{\Gamma(z)}.$$

This function is used to evaluated sums of rational functions having poles at integers.

The `Psi` function computes the DiGamma function and its derivatives.

- `Psi` takes one mandatory argument and one optional argument:
  - a, a real number.
  - Optionally, n, a non-negative integer.
- `Psi(`a`)` returns the value of the DiGamma function $\psi(a)$.
- `Psi(`a, n`)` returns the nth derivative of the DiGamma function at $x = a$.

Examples

```
> Psi(3)
```

$$\frac{3}{2} - \gamma$$

```
> evalf(Psi(3))
```

$$0.922784335098$$

```
> Psi(3,1)
```

$$\frac{\pi^2}{6} - \frac{5}{4}$$

7.3.18 ζ function

The ζ function is defined by

$$\zeta(x) = \sum_{n=1}^{+\infty} \frac{1}{n^x}$$

for $x > 1$, and by its meromorphic continuation for $x < 1$.

The `Zeta` command computes the ζ function.

- `Zeta` takes x, a real number.
- `Zeta(`x`)` returns the value of the ζ function $\zeta(x)$.

Examples

```
> Zeta(2)
```

$$\frac{\pi^2}{6}$$

```
> Zeta(4)
```

$$\frac{\pi^4}{90}$$

7.3.19 Airy functions

The Airy functions of the first and second kind are defined by

$$\mathrm{Ai}(x) = \frac{1}{\pi}\int_0^{+\infty} \cos\left(t^3/3 + xt\right) \mathrm{d}t, \quad \mathrm{Bi}(x) = \frac{1}{\pi}\int_0^{+\infty} \left(\mathrm{e}^{-t^3/3} + \sin\left(t^3/3 + xt\right)\right) \mathrm{d}t.$$

Let f and g be two entire series solutions of $w'' - xw = 0$. Then

$$\mathrm{Ai}(x) = \mathrm{Ai}(0)f(x) + \mathrm{Ai}'(0)g(x), \quad \mathrm{Bi}(x) = \sqrt{3}(\mathrm{Ai}(0)f(x) - \mathrm{Ai}'(0)g(x)),$$

where $f(x) = \sum_{k=0}^{\infty} 3^k \left(\frac{\Gamma(k+\frac{1}{3})}{\Gamma(\frac{1}{3})}\right) \frac{x^{3k}}{(3k)!}$ and $g(x) = \sum_{k=0}^{\infty} 3^k \left(\frac{\Gamma(k+\frac{2}{3})}{\Gamma(\frac{2}{3})}\right) \frac{x^{3k+1}}{(3k+1)!}$.

The `Airy_Ai` and `Airy_Bi` commands compute the Airy functions.

- `Airy_Ai` and `Airy_Bi` take one argument: x, a real number.
- `Airy_Ai(`x`)` and `Airy_Bi(`x`)` return the values of the Airy functions.

Examples

```
> Airy_Ai(1)
```

0.135292416313

```
> Airy_Bi(1)
```

1.20742359495

```
> Airy_Ai(0)
```

0.355028053888

```
> Airy_Bi(0)
```

0.614926627446

7.4 Complex numbers

Note that complex numbers are also used to represent points in the plane (see Section 26.5.2, p. 685). Some functions and operators which work on complex numbers also work on points.

7.4.1 Usual arithmetic operators for complex numbers

The `+`, `-`, `*`, `/`, `^` operators are the usual operators to perform addition, subtraction, multiplication, division and for raising to a power. For example:

```
> (1+2*i)^2
```

$-3 + 4\mathrm{i}$

7.4.2 Real and imaginary parts of a complex number

The `re` (or `real`) and `im` (or `imag`) commands find the real and imaginary parts of a complex number.

- `re` takes a, a complex number (or point).
- `re(`a`)` returns the real part of the complex number a (or the projection of the point a onto the x axis).
- `im` takes a, a complex number (or point).
- `im(`a`)` returns the imaginary part of the complex number a (or the projection of the point a onto the y axis).

Examples

```
> re(3+4*i)
```

$$3$$

```
> im(3+4*i)
```

$$4$$

7.4.3 Writing a complex number z in rectangular form

The `evalc` command will ensure that a complex number is in rectangular form.

- `evalc` takes z, a complex number.
- `evalc(`z`)` returns z written as `re(`z`)+i*im(`z`)`.

Example

```
> evalc(sqrt(2)*exp(i*pi/4))
```

$$1 + \mathrm{i}$$

7.4.4 Modulus and argument of a complex number

A complex number z can be written in polar form $re^{i\theta}$, where r is the modulus and θ is the argument. The angle θ is only determined up to a multiple of 2π; there will be a unique value in the interval $(-\pi, \pi]$, the value in this interval is called the *principal value* of the argument.

The `abs` and `arg` commands find the modulus and argument of a complex number, respectively (see also Section 8.3.2, p. 155).

- `abs` takes z, a complex number.
- `abs(`z`)` returns the modulus $|z|$.
- `arg` takes z, a complex number.
- `arg(`z`)` returns the principal value of the argument of z.

Examples

```
> abs(3+4*i)
```

$$5$$

```
> arg(3+4*i)
```

$$\arctan\left(\frac{4}{3}\right)$$

```
> arg(3.0+4.0*i)
```

$$0.927295218002$$

7.4.5 Normalized complex number

The `normalize` or `unitV` command finds the unit complex number in the same direction as a given complex number.

- `normalize` takes z, a non-zero complex number.
- `normalize(`z`)` returns the unit complex number with the same direction as z, namely z divided by the modulus of z.

Example

```
> normalize(3+4*i)
```

$$\frac{3+4\mathrm{i}}{5}$$

7.4.6 Conjugate of a complex number

The `conj` command finds the conjugate of a complex number.

- `conj` takes z, a complex number.
- `conj(`z`)` returns the complex conjugate of z.

Example

```
> conj(3+4*i)
```

$$3-4\mathrm{i}$$

7.4.7 Multiplication by the complex conjugate

The denominator of a complex expression can be made a real number by multiplying the numerator and denominator of the expression by the complex conjugate of the denominator. The `mult_c_conjugate` can perform this multiplication.

- `mult_c_conjugate` takes *expr*, a complex expression.
- `mult_c_conjugate(`*expr*`)` returns the following:

– If *expr* is a fraction with a complex (non-real) denominator, then this expression is returned with the numerator and denominator multiplied by the complex conjugate of the denominator.

– If *expr* is a fraction with a real denominator (if *expr* is not a fraction, it is regarded as a fraction with a denominator of 1), then this expression is returned with the numerator and denominator multiplied by the complex conjugate of the numerator.

Examples

```
> mult_c_conjugate((2+i)/(2+3*i))
```

$$\frac{(2+\mathrm{i})\,(2-3\mathrm{i})}{(2+3\mathrm{i})\,(2-3\mathrm{i})}$$

```
> mult_c_conjugate((2+i)/2)
```

$$\frac{(2+\mathrm{i})\,(2-\mathrm{i})}{2\,(2-\mathrm{i})}$$

7.4.8 Barycenter of complex numbers

The *barycenter*, or center of mass, of a set of points $A_1, A_2, \ldots, A_n$ with masses $\alpha_1, \alpha_2, \ldots, \alpha_n$ is

$$\frac{\alpha_1 A_1 + \cdots + \alpha_n A_n}{\alpha_1 + \cdots + \alpha_n}.$$

This formula makes sense even if the α_j are not positive real numbers, and is still called the barycenter of the weighted points.

The `barycenter` command computes the barycenter of a set of weighted points.

- `barycenter` takes an arbitrary number of arguments: each argument is a list $l_j = [A_j, \alpha_j]$ containing a point A_j (or the affix of a point) and a weight α_j for the point. The sum of the weights needs to be non-zero. These lists can also be given as two columns of a matrix.
- `barycenter`$(l_1, l_2, \ldots, l_n)$ returns the barycenter of the points A_j weighted by the real coefficients α_j. If $\sum \alpha_j = 0$, `barycenter` returns an error.

Remark. Note that the barycenter command returns a point, not an affix. To output the point affix, you must input `affix(barycenter(`l_1, l_2`))` (see Section 26.12.1, p. 720).

Example

```
> affix(barycenter([1+i,2],[1-i,1]))
```

or:

```
> affix(barycenter([[1+i,2],[1-i,1]]))
```

$$\frac{3+\mathrm{i}}{3}$$

7.5 Algebraic numbers

7.5.1 Definition

A *real algebraic number* is a real root of a polynomial with integer coefficients. A *complex algebraic number* is a root of a polynomial with coefficients which are Gaussian integers.

7.5.2 Minimum polynomial of an algebraic number

The minimal polynomial of an algebraic number is the monic polynomial of smallest degree with integer coefficents which has the algebraic number as a root.

The `pmin` command finds the minimum polynomial of an algebraic number.

- `pmin` takes one mandatory argument and one optional argument:
  - α, an algebraic number.
  - Optionally, x, a variable name to use as the variable in the polynomial.
- `pmin(`α`)` returns the minimal polynomial for α, where the polynomial is given as a list of the coefficients (see Section 11.1.1, p. 211).

  `pmin(`α, x`)` returns the minimal polynomial for α as a symbolic expression with the variable x.

Examples

```
> pmin(sqrt(2)+sqrt(3))
```

$$[\![1, 0, -10, 0, 1]\!]$$

```
> pmin(sqrt(2)+sqrt(3),x)
```

$$x^4 - 10x^2 + 1$$

Note that $(\sqrt{2} + \sqrt{3})^2 = 5 + 2\sqrt{6}$ and so $((\sqrt{2} + \sqrt{3})^2 - 5)^2 = 24$, which can be rewritten as $(\sqrt{2} + \sqrt{3})^4 - 10(\sqrt{2} + \sqrt{3})^2 + 1 = 0$.

```
> pmin(sqrt(2)+i*sqrt(3))
```

$$[\![1, 0, 2, 0, 25]\!]$$

```
> pmin(sqrt(2)+i*sqrt(3),z)
```

$$z^4 + 2z^2 + 25$$

```
> pmin(sqrt(2)+2*i)
```

$$[\![1, 0, 4, 0, 36]\!]$$

```
> pmin(sqrt(2)+2*i,z)
```

$$z^4 + 4z^2 + 36$$

8 Operators and functions

8.1 Operators or infixed functions

An operator is an infixed function. For example, the arithmetic functions `+`, `-`, `*`, `/`, and `^` are operators. (See Section 7.3.2, p. 129 and Section 7.4.1, p. 140.)

8.1.1 Special Xcas operators

Sequences. `$` is the infixed version of `seq` (see Section 6.1.2, p. 67). For example (do not forget to put parenthesis around the arguments):

```
> (2^k)$(k=0..3)
```

or:

```
> seq(2^k,k=0..3)
```

$$1, 2, 4, 8$$

Modular numbers. `mod` or `%` defines a modular number; $a \mod n$ is the equivalence class of a in $\mathbb{Z}/n\mathbb{Z}$. For example:

```
> 5 % 7
```

or:

```
> 5 mod 7
```

$$(-2)\,\%\,7$$

Function composition. `@` is used to compose functions; $(f@g)(x) = f(g(x))$. For example:

```
> (sin@exp)(x)
```

$$\sin\left(\mathrm{e}^{x}\right)$$

`@@` is used to compose a function with itself several times (like a power, replacing multiplication by composition), e.g. $(f@@3)(x) = f(f(f(x)))$. For example:

```
> (sin@@4)(x)
```

$$\sin\left(\sin\left(\sin\left(\sin x\right)\right)\right)$$

Set operations. `minus`, `union` and `intersect` return the difference, the union and the intersection of two sets, respectively. (See Section 3.2.2, p. 34). For example:

```
> A:=set[1,2,3,4]:; B:=set[3,4,5,6]:;
> A minus B
```

$$[\![1, 2]\!]$$

```
> A union B
```

$$[\![1,2,3,4,5,6]\!]$$

```
> A intersect B
```

$$[\![3,4]\!]$$

Defining functions. `->` is used to define a function, which can be assigned a name. For example:

```
> (x->x^2)(3)
```

$$9$$

```
> f:=x->x^2:; f(3)
```

$$9$$

Assignment. `=>` is the infixed version of `sto` (see Section 3.3.2, p. 36) and so is used to store an expression in a variable. For example:

```
> 2=>a
```

$$2$$

`:=` is used to store an expression in a variable, but the variable comes first (the argument order is switched from `=>`). For example:

```
> a:=2
```

$$2$$

`=<` to store an expression in a variable, but the storage is done by reference if the target is a matrix element or a list element. This is faster if you modify objects inside an existing list or matrix of large size, because no copy is made, the change is done in place. Use with care, all objects pointing to this matrix or list will be modified. For example:

```
> L:=[2,3]:; L2:=L:;
```

then:

```
> L[0]=<5:;
> L, L2
```

$$[5,3]\,,[5,3]$$

8.1.2 Defining custom operators

The `user_operator` command lets you define an operator or delete an operator you previously defined. When you use an operator you defined, you have to make sure that you leave spaces around the operator.

- To define an operator, `user_operator` takes three arguments:
  - *name*, a string which is the name of the operator.
  - *fn*, a function of one or two variables with values in $\mathbb{R}$ or in `true`, `false`.

  - *type*, to specify what kind of an operator you are defining. The possible values are:
    * `Binary`, to define an infixed operator. In this case, `fn` must be a function of two variables.
    * `Prefix` (or `Unary`), to define a prefixed operator. In this case, *fn* must be a function of one variable.
    * `Postfix`, to define a postfixed operator. In this case *fn* must be a function of one variable.

- `user_operator(`*name*, *fn*, *type*`)` returns 1 if the definition was successful and otherwise returns 0.
- To delete an operator, `user_operator` takes two arguments:
  - *name*, a string which is the name of the operator.
  - `Delete`, the symbol.
- `user_operator(`*name*,`Delete)` deletes the operator.

Examples

Let R be defined on $\mathbb{R} \times \mathbb{R}$ by $x \ R \ y = xy + x + y$. To input R:

```
> user_operator("R",(x,y)->x*y+x+y,Binary)
```

1

(Do not forget to put spaces around `R`.)

```
> 5 R 7
```

47

Let S be defined on $\mathbb{N}$ by "for x and y integers, $x \ S \ y$ means that x and y are *not* coprime." To input S:

```
> user_operator("S",(x,y)->(gcd(x,y))!=1,Binary)
```

1

(Do not forget to put spaces around `S`.)

```
> 5 S 7
```

false

```
> 8 S 12
```

true

Let T be defined on $\mathbb{R}$ by $Tx = x^2$. To input T:

```
> user_operator("T",x->x^2,Prefix)
```

1

(Do not forget to put a space before `T`.)

```
> T 4
```

16

Let U be defined on $\mathbb{R}$ by $xU = 5x$. To input U:

```
> user_operator("U",x->5*x,Postfix)
```

1

(Do not forget to put a space before `T`.)

```
> 3 U
```

15

8.2 Functions and expressions with symbolic variables

8.2.1 Difference between a function and an expression

Functions are often defined with expressions; for example, the command line `f(x):=x^-1` defines a function `f`, whose value at x is given by $x^2 + 1$. (The function `f` can also be defined by `f:=x->x^2-1`.) But the function is not the same as the expression; the variable `x` is only a placeholder for the function; it is not part of actual definition of the function. Compare this with `g:=x^2-1`, where `g` is a variable which stores the expression `x^2-1` and so the identifier `x` is part of the definition of `g`. To find the value of f for $x = 2$, you can enter `f(2)`, but to use `g` to find the same value you have to do an explicit substitution and enter `subst(g,x=2)`.

When a command expects a function as argument, this argument should be either the definition of the function (e.g. `x->x^2-1`) or a variable name assigned to a function (e.g. `f` previously defined by `f(x):=x^2-1`).

When a command expects an expression as argument, this argument should be either the definition of the expression (for example `x^2-1`), or a variable name assigned to an expression (e.g. `g` previously defined by `g:=x^2-1`), or the evaluation of a function (e.g. `f(x)` where `f` is the previously defined function by `f(x):=x^2-1`).

8.2.2 Transforming an expression into a function

The `unapply` command transforms an expression into a function.

- `unapply` takes two arguments:
  - *expr*, an expression.
  - x, the name of a variable or sequence of names of variables.
- `unapply(`*expr*$, x$`)` returns the function defined by the expression *expr* and variable(s) x, as in x`->`*expr*.

Examples

```
> unapply(exp(x+2),x)
```

$$x \mapsto \mathrm{e}^{x+2}$$

```
> unapply(x*y-x-y,(x,y))
```

$$(x, y) \mapsto xy - x - y$$

Remark. When a function being is defined, the right side of the assignment is not evaluated, hence `g:=sin(x+1); f(x):=g` does not define the function $f : x \to \sin(x+1)$ but defines the function $f : x \to g$. To define the former function, `unapply` should be used, as in the following example:

```
> g:=sin(x+1); f:=unapply(g,x)
```

$$\sin(x+1), x \mapsto \sin(x+1)$$

Hence, the variable `g` is assigned to a symbolic expression and the variable `f` is assigned to a function.

Examples

```
> f:=unapply(lagrange([1,2,3],[4,8,12]),x)
```

(See Section 17.1.1, p. 435.)

$$x \mapsto 4(x-1)+4$$

```
> f:=unapply(integrate(log(t),t,1,x),x)
```

$$x \mapsto x\ln x - x + 1$$

```
> f:=unapply(integrate(log(t),t,1,x),x):;
  f(x)
```

$$x\ln x - x + 1$$

Remark. Suppose that f is a function of 2 variables $f : (x, w) \to f(x, w)$, and that g is the function defined by $g : w \to h_w$, where h_w is the function defined by $h_w(x) = f(x, w)$.

`unapply` can also be used to define g.

Example

```
> f(x,w):=2*x+w:;
  g(w):=unapply(f(x,w),x):;
  g(3)
```

$$x \mapsto 2x+3$$

8.2.3 Top and leaves of an expression

An expression can be represented by a tree. The top of the tree is either an operator or a function and the leaves of the tree are the arguments of the operator or function (see also Section 6.1.10, p. 74).

The `sommet` command finds the top of an expression.

- `sommet` takes *expr*, an expression.
- `sommet(`*expr*`)` returns the top of *expr*.

Examples

```
> sommet(sin(x+2))
```

$$\sin$$

```
> sommet(x+2*y)
```

$$+$$

The `op` or *feuille* command finds the list of the leaves of an expression.

- `op` takes *expr*, an expression.
- `op(`*expr*`)` returns the leaves of *expr*.

Examples

```
> op(sin(x+2))
```

or:

```
> feuille(sin(x+2))
```

$$x+2$$

```
> op(x+2*y)
```

or:

```
> feuille(x+2*y)
```

$$x, 2y$$

If the top of an expression *expr* is an infixed operator, the left hand side will be *expr*`[1]` and the right hand side will be *expr*`[2]`. The `left` and `right` commands are alternative commands to find the sides (see Section 5.2.4, p. 56, Section 6.5.1, p. 98, Section 6.6.2, p. 100, Section 6.3.6, p. 80, Section 9.3.4, p. 183 and Section 9.3.5, p. 183 for other uses of `left` and `right`.)

- `left` and `right` take one argument: *expr*, an expression whose top is an infixed operator.
- `left(`*expr*`)` returns the left side of the operator.
- `right(`*expr*`)` returns the right side of the operator.

Examples

```
> sommet(y=x^2)
```

$$=$$

```
> left(y=x^2)
```

$$y$$

```
> right(y=x^2)
```

$$x^2$$

Remark. If a function is defined by a program (see Section 25.1.2, p. 647) then the top will be the function `'program'` and the leaves will be a sequence consisting of the arguments of the defined function, followed by a sequence of `0`s (one for each argument) followed by the body of the function. For example, define the `pgcd` function:

```
pgcd(a,b):={
  local r;
  while (b!=0) {
    r:=irem(a,b);
    a:=b; b:=r;
  }
  return a;
}
```

Then:

```
> sommet(pgcd)
```

program

```
> feuille(pgcd)[0]
```

or:

```
> op(pgcd)[0]
```

a, b

```
> feuille(pgcd)[1]
```

or:

```
> op(pgcd)[1]
```

$0, 0$

```
> feuille(pgcd)[2]
```

or:

```
> op(pgcd)[2]
```

```
{ local r;
  while(b<>0) {
    r:=irem(a,b);
    a:=b;
    b:=r;
  }
  return a;
}
```

8.3 Functions

8.3.1 Context-dependent functions.

The + operator. The `+` operator is infixed and `'+'` is its prefixed version. The `+` operator will add numbers (see Section 7.3.2, p. 129), concatenate strings (see Section 5.2.13, p. 61), and convert a number to a string if necessary. Addition makes sense for other objects, and `+` can flexibly deal with them; the result of using the `+` operator depends on the nature of its arguments.

Examples

```
> 1+2+3+4
```

or:

```
> '+'(1,2,3,4)
```

or:

```
> (1,2)+(3,4)
```

or:

```
> (1,2,3)+4)
```

$$10$$

(See Section 6.1.9, p. 73.)

```
> 1+i+2+3*i
```

or:

```
> '+'(1,i,2,3*i)
```

$$3+4\mathrm{i}$$

```
> [1,2,3]+[4,1]
```

or:

```
> [1,2,3]+[4,1,0]
```

or:

```
> '+'([1,2,3],[4,1])
```

$$[5,3,3]$$

```
> [1,2]+[3,4]
```

or:

```
> '+'([1,2],[3,4])
```

$$[4,6]$$

```
> [[1,2],[3,4]]+[[1,2],[3,4]]
```

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

```
> [1,2,3]+4
```

or:

```
> '+'([1,2,3],4)
```

$$[\![1,2,7]\!]$$

(This is a polynomial; see Section 11.1.1, p. 211.)

```
> [1,2,3]+(4,1)
```

or:

```
> '+'([1,2,3],4,1)
```

$$[\![1,2,8]\!]$$

```
> "Hel"+"lo"
```

or:

```
> '+'("Hel","lo")
```

"Hello"

The -, * and / operators. The -, * and / operators (and their prefixed versions '-', '*' and '/'), like the + operator, are flexible and operate on compound objects (such as lists and sequences), but do not concatenate strings.

Examples of - and '-'.

```
> (1,2)-(3,4)
```

$$-4$$

```
> (1,2,3)-4
```

$$2$$

```
> [1,2,3]-[4,1]
```

or:

```
> [1,2,3]-[4,1,0]
```

or:

```
> '-'([1,2,3],[4,1])
```

$$[-3, 1, 3]$$

```
> [1,2]-[3,4]
```

or:

```
> '-'([1,2],[3,4])
```

$$[-2, -2]$$

```
> [[3,4],[1,2]]-[[1,2],[3,4]]
```

$$\begin{bmatrix} 2 & 2 \\ -2 & -2 \end{bmatrix}$$

```
> [1,2,3]-4
```

or:

```
> '-'([1,2,3],4)
```

$$[\![1, 2, -1]\!]$$

```
> [1,2,3]-(4,1)
```

$$[\![1, 2, -2]\!]$$

Examples of `*` and `'*'`.

```
> (1,2)*(3,4)
```

or:

```
> (1,2,3)*4
```

or:

```
> 1*2*3*4
```

or:

```
> '*'(1,2,3,4)
```

$$24$$

```
> 1*i*2*3*i
```

or:

```
> '*'(1,i,2,3*i)
```

$$-6$$

```
> [10,2,3]*[4,1]
```

or:

```
> [10,2,3]*[4,1,0]
```

or:

```
> '*'([10,2,3],[4,1])
```

$$42$$

These compute the scalar product.

```
> [1,2]*[3,4]
```

or:

```
> '*'([1,2],[3,4])
```

$$11$$

These compute the scalar product.

```
> [[1,2],[3,4]]*[[1,2],[3,4]]
```

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

```
> [1,2,3]*4
```

or:

```
> '*'([1,2,3],4)
```

$$[4, 8, 12]$$

```
> [1,2,3]*(4,2)
```

or:

```
> '*'([1,2,3],4,2)
```

or:

```
> [1,2,3]*8
```

$$[8, 16, 24]$$

```
> (1,2)+i*(2,3)
```

or:

```
> 1+2+i*2*3
```

$$3 + 6\mathrm{i}$$

Examples of / and '/'.

```
> [10,2,3]/[4,1]
```

$$\left[\frac{5}{2}, 2\right]$$

```
> [1,2]/[3,4]
```

or:

```
> '/'([1,2],[3,4])
```

$$\left[\frac{1}{3}, \frac{1}{2}\right]$$

```
> 1/[[1,2],[3,4]]
```

or:

```
> '/'(1,[[1,2],[3,4]]
```

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

```
> [[1,2],[3,4]]*1/[[1,2],[3,4]]
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
> [[1,2],[3,4]]/[[1,2],[3,4]]
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

(This is term-by-term division.)

8.3.2 Standard functions

The `max` command finds the maximum of a sequence of real numbers.

- `max` takes *seq*, a sequence (or list) of real numbers.
- `max(`*seq*`)` returns the largest number in the sequence *seq*.

Example

```
> max(0,1,2,-1,-2)
```

$$2$$

The `min` command finds the minimum of a sequence of real numbers.

- `min` takes *seq*, a sequence (or list) of real numbers.
- `min(`*seq*`)` returns the smallest number in the sequence *seq*.

Example

```
> min(0,1,2,-1,-2)
```

$$-2$$

The `abs` command finds the absolute value of a real or complex number.

- `abs` takes x, a real or complex number.
- `abs(`x`)` returns the absolute value of x.

Examples

```
> abs(-5)
```

$$5$$

```
> abs(3+4*i)
```

$$5$$

The `sign` command finds the sign of a real number (+1 if it is positive, 0 if it is zero, and -1 if it is negative).

- `sign` takes x, a real number.
- `sign(`x`)` returns the sign of x.

Examples

```
> sign(-4)
```

$$-1$$

```
> sign(0)
```

$$0$$

The `floor` or `iPart`) command finds the floor of a real number; namely, the largest integer less than or equal to the number.

- `floor` takes x, a real number.
- `floor(`x`)` returns the floor of x.

Examples

```
> floor(4.1)
```

$$4$$

```
> floor(-4.1)
```

$$-5$$

The `round` command rounds a number to the nearest integer, rounding up in the case of a half-integer.

- `round` takes x, a real number.
- `round(`x`)` returns the nearest integer to x.

Examples

```
> round(3.4)
```

3

```
> round(-3.4)
```

−3

```
> round(3.5)
```

4

The `ceil` or `ceiling` command finds the ceiling of a real number; namely, the smallest integer greater than or equal to the number.

- `ceil` takes x, a real number.
- `ceil`(x) returns the ceiling of x.

Examples

```
> ceiling(4.1)
```

5

```
> ceiling(-4.1)
```

−4

The `frac` or `fPart` command finds the fractional part of a number; informally, the part of the number to the right of the decimal point with the appropriate plus or minus sign. For a positive real number x, the fractional part is x minus the floor of x; for a negative real number x, the fractional part is x minus the ceiling of x.

- `frac` takes x, a real number.
- `frac`(x) returns the fractional part of x.

Examples

```
> frac(3.24)
```

0.24

```
> frac(-3.24)
```

−0.24

The `trunc` command truncates a real number; namely, it removes the fractional part. The truncated number added to the fractional part will equal the original number.

- `trunc` takes x, a real number.
- `trunc`(x) returns the truncated value of x.

Examples

```
> trunc(3.24)
```

$$3$$

```
> trunc(-3.24)
```

$$-3$$

The `id` command is the identity function.

- `id` takes one argument or a sequence of:
  seq, whose elements can be any type.
- `id(`*seq*`)` returns *seq*.

Example

```
> id(a,1,"abc",[1,2,3])
```

$$a, 1, \text{“abc”}, [1, 2, 3]$$

The `sq` command squares its argument.

- `sq` takes x, any object that can be multiplied by itself.
- `sq(`x`)` returns x^2.

Examples

```
> sq(5)
```

$$25$$

```
> sq(x+y)
```

$$(x+y)^2$$

```
> sq([[1,2],[3,4]])
```

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

(This is a matrix product; see Section 14.2, p. 325).

```
> sq([1,2,3])
```

$$14$$

(This is the dot product of $[1, 2, 3]$ with itself.)
The `sqrt` command finds the square root of its argument.

- `sqrt` takes x, any object for which the $1/2$ power makes sense.
- `sqrt(`x`)` returns $x^{1/2}$.

Examples

```
> sqrt(9)
```

$$3$$

```
> sqrt((x+y)^2)
```

$$|x+y|$$

```
> simplify(sqrt([[2,3],[3,5]]))
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

The `surd` command finds roots of quantities.

- `surd` takes two arguments: x and n, numbers.
- `surd`(x, n) returns the nth root of x; i.e., $x^{1/n}$.

Example

```
> surd(15.625,3)
```

$$2.5$$

The `exp` command computes the exponential function.

- `exp` takes x, a number.
- `exp`(x) returns e^x.

Example

```
> exp(1.0)
```

$$2.71828182846$$

The `log` or `ln` command computes the natural logarithm.

- `log` takes x, a number.
- `log`(x) returns the natural logarithm of x.

Example

```
> log(2.0)
```

$$0.69314718056$$

The `log10` command computes the common logarithm.

- `log10` takes x, a number.
- `log10`(x) returns the base-10 logarithm of x.

Example

```
> log10(1000)
```

3

The `log2` command computes the binary logarithm.

- `log2` takes x, a number.
- `log2`(x) returns the base-2 logarithm of x.

Example

```
> log2(1024)
```

10

The `logb` computes the logarithm to a specified base.

- `logb` takes two arguments: x and b, non-zero numbers.
- `logb`(x, b) returns the base-b logarithm of x.

Example

```
> logb(10.0,2)
```

3.32192809489

Trigonometric functions.

- The `sin` command is the sine function.
- The `cos` command is the cosine function.
- The `tan` command is the tangent function, $\tan x = \frac{\sin x}{\cos x}$.
- The `cot` command is the cotangent function, $\cot x = \frac{\cos x}{\sin x}$.
- The `sec` command is the secant function, $\sec x = \frac{1}{\cos x}$.
- The `csc` command is the cosecant function, $\csc x = \frac{1}{\sin x}$.
- These commands take one argument: x, a number.

  The number x will by default represent an angle measured in radians, but you can set XCAS to use degrees (see Section 2.5.3, p. 14) by setting the variable `angle_radian` to 0; resetting it to 1 will change the angle measure to radians again.
- `sin`(x) returns the sine of x.

Examples

Input with `angle_radian` equal to 1:

```
> sin(pi/4)
```

$$\frac{\sqrt{2}}{2}$$

Input with `angle_radian` equal to 0:

```
> sin(30)
```

$$\frac{1}{2}$$

`cos(`x`)` returns the cosine of x. For example, input with `angle_radian` equal to 1:

```
> cos(pi/6)
```

$$\frac{\sqrt{3}}{2}$$

Input with `angle_radian` equal to 0:

```
> cos(90)
```

$$0$$

`tan(`x`)` returns the tangent of x. For example, input with `angle_radian` equal to 1:

```
> tan(pi/4)
```

$$1$$

Input with `angle_radian` equal to 0:

```
> tan(60)
```

$$\sqrt{3}$$

`cot(`x`)` returns the cotangent of x. For example, input with `angle_radian` equal to 1:

```
> cot(pi/6)
```

$$\frac{2\sqrt{3}}{2}$$

Input with `angle_radian` equal to 0:

```
> cot(45)
```

$$1$$

`sec(`x`)` returns the secant of x. For example, input with `angle_radian` equal to 1:

```
> sec(pi/3)
```

$$2$$

Input with `angle_radian` equal to 0:

```
> sec(30)
```

$$\frac{2}{\sqrt{3}}$$

`csc(`x`)` returns the cosecant of x. For example, input with angle_radian equal to 1:

```
> csc(pi/4)
```

$$\frac{2}{\sqrt{2}}$$

Input with `angle_radian` equal to 0:

```
> csc(30)
```

$$2$$

Inverse trigonometric functions. The `asin`, `acos`, `atan`, `acot`, `asec`, `acsc` commands are the inverse trigonometric functions. The latter are defined by:

- `asec(x)=acos(1/x)`,
- `acsc(x)=asin(1/x)`,
- `acot(x)=atan(1/x)`.

`arcsin`, `arccos` and `arctan` are synonyms for `asin`, `acos` and `atan`, respectively.

- These functions take one argument: x, a number.
- They return a number which can represent an angle; by default, the angles will be in radians, but you can set Xcas to use degrees (see Section 2.5.3, p. 14) by setting the variable `angle_radian` to 0; resetting it to 1 will change the angle measure to radians again.

Examples

Input with `angle_radian` equal to 1:

```
> asin(1/2)
```

$$\frac{\pi}{6}$$

Input with `angle_radian` equal to 0:

```
> asin(1)
```

$$\frac{\pi}{2}$$

`acos(`x`)` returns the arccosine of x. For example: Input with `angle_radian` equal to 1:

```
> acos(sqrt(3)/2)
```

$$\frac{1}{6}\pi$$

Input with `angle_radian` equal to 0:

```
> acos(-1/2)
```

$$120$$

`atan(`x`)` returns the arctangent of x. For example, input with `angle_radian` equal to 1:

```
> atan(sqrt(3))
```

$$\frac{\pi}{3}$$

Input with `angle_radian` equal to 1:

```
> atan(1)
```

$$45$$

`acot`(x) returns the arccotangent of x. For example, input with `angle_radian` equal to 1:

```
> acot(sqrt(3))
```

$$\frac{\pi}{6}$$

Input with `angle_radian` equal to 0:

```
> acot(1/sqrt(3))
```

$$60$$

`asec`(x) returns the arcsecant of x. For example, input with `angle_radian` equal to 1:

```
> asec(1)
```

$$0$$

Input with `angle_radian` equal to 0:

```
> asec(sqrt(2))
```

$$45$$

`acsc`(x) returns the arccosecant of x. For example, input with `angle_radian` equal to 1:

```
> acsc(1)
```

$$\frac{\pi}{2}$$

Input with `angle_radian` equal to 0:

```
> acsc(2)
```

$$30$$

Hyperbolic functions. The `sinh`, `cosh`, and `tanh` commands compute the hyperbolic sine, hyperbolic cosine, and hyperbolic tangent functions.

- These functions take one argument: x, a number.
- `sinh`(x) returns the hyperbolic sine of x.

Examples

```
> sinh(1.0)
```

$$1.17520119364$$

`cosh`(x) returns the hyperbolic cosine of x. For example:

```
> cosh(0)
```

$$1$$

`tanh`(x) returns the hyperbolic tangent of x. For example:

```
> tanh(-1.0)
```

$$-0.761594155956$$

Inverse hyperbolic functions. The `asinh`, `acosh`, and `atanh` commands compute the inverse hyperbolic functions.

`arcsinh`, `arccosh` and `arctanh` are synonyms for `asinh`, `acosh` and `atanh`, respectively.

- These functions take one argument: x, a number.
- `asinh(`x`)`, `acosh(`x`)`, and `atanh(`x`)` return the inverse hyperbolic sine, cosine, and tangent of x, respectively.

Examples

```
> asinh(2)
```

$$\ln\left(2+\sqrt{5}\right)$$

`acosh(`x`)` returns the inverse hyperbolic cosine of x. For example:

```
> acosh(1)
```

$$0$$

`atanh(`x`)` returns the inverse hyperbolic tangent of x. For example:

```
> atanh(1/2)
```

$$\frac{\ln(3)}{2}$$

8.3.3 Defining algebraic functions

Defining a function from $\mathbb{R}^p$ to $\mathbb{R}^q$. If *expr* is an expression possibly involving a variable x, use it to define a function f either by

$$f(x)\texttt{:=}expr$$

or

$$f\texttt{:=}x\texttt{->}expr$$

(see Section 3.4.1, p. 43). Note that the expression after `->` is not evaluated. You should use `unapply` (see Section 8.2.2, p. 148) if you expect the second member to be evaluated before the function is defined.

Example

To define $f(x) = x\sin(x)$, input:

```
> f(x):=x*sin(x)
```

or:

```
> f:=x->x*sin(x)
```

Then:

```
> f(pi/4)
```

$$\frac{\pi\sqrt{2}}{8}$$

You can similarly define a function of several variables, by replacing x by a sequence $(x_1,\ldots,x_p)$ or a list $[x_1,\ldots,x_p]$ of variables.

Example

To define $f(x) = x\sin(y)$, input:

```
> f(x,y):=x*sin(y)
```

or:

```
> f:=(x,y)->x*sin(y)
```

Then:

```
> f(2,pi/6)
```

$$1$$

You can also define a function with values in $\mathbb{R}^q$ by replacing *expr* by a sequence $(expr_1, \ldots, expr_q)$ or list $[expr_1, \ldots, expr_q]$ of expressions.

Examples

Define the function $h(x, y) = (x\cos(y), x\sin(y))$.

```
> h(x,y):=(x*cos(y),x*sin(y))
```

Then:

```
> h(2,pi/4)
```

$$\sqrt{2}, \sqrt{2}$$

Define the function $h(x, y) = [x\cos(y), x\sin(y)]$.

```
> h(x,y):=[x*cos(y),x*sin(y)];
```

or:

```
> h:=(x,y)->[x*cos(y),x*sin(y)];
```

or:

```
> h(x,y):={ [x*cos(y),x*sin(y)] };
```

or:

```
> h:=(x,y)->return [x*cos(y),x*sin(y)];
```

or:

```
> h(x,y):={ return [x*cos(y),x*sin(y)]; }
```

Then:

```
> h(2,pi/4)
```

$$\left[\sqrt{2}, \sqrt{2}\right]$$

Defining families of function from $\mathbb{R}^{p-1}$ to $\mathbb{R}^q$ using a function from $\mathbb{R}^p$ to $\mathbb{R}^q$. Suppose that the function $f : (x, y) \to f(x, y)$ is defined, and you want to define a family of functions $g(t)$ such that $g(t)(y) := f(t, y)$ (i.e. t is viewed as a parameter). Since the expression after `->` (or `:=`) is not evaluated, you should not define $g(t)$ by `g(t):=y->f(t,y)`; you have to use the `unapply` command (see Section 8.2.2, p. 148).

For example, to define $f : (x, y) \to x\sin(y)$ and $g(t) : y \to f(t, y)$:

```
> f(x,y):=x*sin(y); g(t):=unapply(f(t,y),y)
```

then:

```
> g(2)
```

$$y \mapsto 2\sin y$$

As another example, suppose that you want to define the function $h : (x, y) \to [x\cos(y), x\sin(y)]$ and then you want to define the family of functions $k(t)$ having t as parameter such that $k(t)(y) := h(t, y)$. To define the function $h(x, y)$:

```
> h(x,y):=(x*cos(y),x*sin(y))
```

To define properly the function $k(t)$:

```
> k(t):=unapply(h(x,t),x)
```

then:

```
> k(2)
```

$$x \mapsto (x\cos(2), x\sin(2))$$

8.3.4 Composing functions

In Xcas, the composition of functions is done with the infixed operator `@` (see Section 8.1.1, p. 145). For example:

```
> (sq@sin+id)(x)
```

$$\sin^2(x) + x$$

The repeated composition of a function with itself several times is done with the infixed operator `@@` (see Section 8.1.1, p. 145). For example:

```
> (sin@@3)(x)
```

$$\sin(\sin(\sin x))$$

8.3.5 Defining a function with history

The `as_function_of` command creates a function defined by an expression, even if the desired variable already has a value.

- `as_function_of` takes two arguments:
  - x, a variable.
  - *exprvar*, another variable containing an expression which itself may involve x.
- `as_function_of(`*exprvar*$, x$`)` returns a function defined by the expression that *exprvar* contains.

Example

```
> a:=sin(x)
```

$$\sin(x)$$

```
> b:=sqrt(1+a^2)
```

$$\sqrt{1+\sin^2 x}$$

```
> c:=as_function_of(b,a)
```

```
(a)->{ return(sqrt(1+a^2)); }
```

```
> c(x)
```

$$\sqrt{1+x^2}$$

Remark. If the variable `b` has been assigned several times, the first assignment of `b` following the last assignment of `a` will be used. Moreover, the order used is the order of validation of the commandlines, which may not be reflected by the XCAS interface if you reused previous commandlines.

Example

```
> a:=2:;
  b:=2*a+1:;
  b:=3*a+2:;
  c:=as_function_of(b,a)
```

$$(a)->\{\ return(sqrt(1+a^2));\ \}$$

So `c(x)` is equal to $2x+1$. But:

```
> a:=2:;
  b:=2*a+1:;
  a:=2:;
  b:=3*a+2:;
  c:=as_function_of(b,a)
```

$$(a)->\{\ return(sqrt(2+3*a^2));\ \}$$

So `c(x)` is equal to $3x+2$.

Hence the line where `a` is defined must be reevaluated before the good definition of `b`.

8.4 Getting information about univariate real functions

8.4.1 Domain of a function

The `domain` command finds the domain of a function.

- `domain` takes one mandatory argument and one optional argument:
  - *expr*, an expression involving a single variable.
  - Optionally, x, the variable, which by default will be `x`.
- `domain(`*expr*$\langle, x\rangle$`)` returns the domain of the function defined by *expr*.

Examples

```
> domain(ln(x+1))
```

$$x > -1$$

```
> domain(asin(2*t),t)
```

$$t \geq -\frac{1}{2} \wedge t \leq \frac{1}{2}$$

Find the domain of the function $f(x) = \sqrt{1-\sqrt{2-\sqrt{x-3}}}$:

```
> domain(sqrt(1-sqrt(2-sqrt(x-3))),x)
```

$$x \geq 4 \wedge x \leq 7$$

8.4.2 Table of variations of a function

The table of variations of a function consists of

1. The first row, for the variable, which gives the endpoints of subintervals of the domain, as well as any critical points and inflection points.
2. The second row, for the derivative, which gives the values of the derivative at the values in the first row (or limits as the variable approaches one of the values) and between them the sign (+ or −) of the derivative in the corresponding subinterval.
3. The third row, for the function, which gives the values of the function at the values in the first row, and between them whether the function is increasing or decreasing in the corresponding subinterval.
4. The fourth row, for the second derivative, which gives the values of the second derivative at the values in the first row, and between them whether the second derivative is positive or negative (and hence whether the graph is concave up or concave down) in the subinterval.

The `tabvar` command finds the table of variations of a function.

- `tabvar` takes one mandatory argument and one optional argument.
  - *expr*, an expression of a single variable.
  - Optionally, x, the variable (by default, $x =$ `x`).
- `tabvar(`*expr*$\langle$`,`$x\rangle$`)` returns the table of variations of the function $f(x) =$ *expr* and draws the graph on the `DispG` screen, accessible with the menu Cfg ▶ Show ▶ DispG.

Examples

```
> tabvar(x^2-x-2,x)
```

plotfunc(x^2-x-2,x=((-3.393824) .. 4.574184))
Inside Xcas you can see the function with Cfg>Show>DispG.

$$\begin{bmatrix} x & -\infty & & \frac{1}{2} & & +\infty \\ y'=2x-1 & -\infty & - & 0 & + & +\infty \\ y=x^2-x-2 & +\infty & \downarrow & -\frac{9}{4} & \uparrow & +\infty \\ y'' & 2 & +(\cup) & 2 & +(\cup) & 2 \end{bmatrix}$$

```
> tabvar((2*t-1)/(t-1),t)
```

plotfunc((2*t-1)/(t-1),t=((-2.893824) .. 5.074184))
Inside Xcas you can see the function with Cfg>Show>DispG.

$$\begin{bmatrix} t & -\infty & & 1 & 1 & & +\infty \\ y'=-\frac{1}{(t-1)^2} & 0 & - & \| & \| & - & 0 \\ y=\frac{2t-1}{t-1} & 2 & \downarrow & -\infty & +\infty & \downarrow & 2 \\ y'' & 0 & -(\cap) & \| & \| & +(\cup) & 0 \end{bmatrix}$$

Note that in the second example the value 1 appears twice in the first row, so that both one-sided limits of `y` can be displayed at the vertical asymptote $t = 1$. The values of 2 for `y` at $-\infty$ and ∞ indicate a horizontal asymptote of $y = 2$.

9 Algebraic expressions

9.1 Evaluation and substitution

9.1.1 Expression evaluation

The `eval` command is used to evaluate an expression. Since XCAS always evaluates expressions entered in the command line, `eval` is mainly used to evaluate a sub-expression in the expression editor (see Section 2.9, p. 28).

Examples

```
> a:=2
```

2

```
> eval(2+3*a)
```

or:

```
> 2+3*a
```

8

9.1.2 Changing the evaluation level

When it evaluates expressions, the maximum number of recursions that XCAS will do it called the *evaluation level*. This is 25 by default, but you can change the default level with the eval box in the CAS configuration screen (see section 2.5.7).

The `eval_level` command will change the evaluation level for the current session.

- `eval_level` takes one optional argument:

  Optionally n, a positive integer.

- `eval_level()` returns the current evaluation level.
- `eval_level(`n`)` sets the evaluation level to n.

Example

```
> purge(a,b,c);
  a:=b+1; b:=c+1; c:=3;
> eval_level()
```

25

```
> a,b,c
```

5, 4, 3

```
> eval_level(1);
  a,b,c
```

$$b+1, c+1, 3$$

```
> eval_level(2);
  a,b,c
```

$$c+2, 4, 3$$

```
> eval_level(3);
  a,b,c
```

$$5, 4, 3$$

```
> eval_level()
```

$$3$$

9.1.3 Algebraic expression evaluation

In Maple, `evala` is used to evaluate an expression with algebraic extensions. In Xcas, `evala` is not necessary, it behaves like `eval` (see Section 9.1.1, p. 169), but it is included for Maple compatibility.

9.1.4 Preventing evaluation

You can prevent an expression from being evaluated by *quoting* it, either by preceding it with `'` or with the `quote` or `hold`) command.

Remark. If a is a variable, then a`:=quote(`a`)` (or a`:=hold(`a`)`) is equivalent to `purge(`a`)` (for the sake of Maple compatibility). It returns the value of this variable (or the hypothesis done on this variable).

Example

```
> a:=2;quote(2+3*a)
```

or:

```
> a:=2;'2+3*a'
```

$$2, 2+3a$$

9.1.5 Forcing evaluation

`unquote` is used for evaluation inside a quoted expression.

For example in an assignment, the variable is automatically quoted (not evaluated) so that the user does not have to quote it explicitly each time he want to modify its value. In some circumstances, you might want to evaluate it.

```
> purge(b);a:=b;unquote(a):=3
```

The variable `b` begins as a purely symbolic variable, and the value of `a` is equal to the symbolic variable `b`. In the assignment `unquote(a):=3`, the left hand side `unquote(a)` is evaluated to `b`, and so `b` is assigned the value `3`. Since `a` evaluates to the same thing as `b`, `a` also evaluated to `3`.

```
> a,b
```

$$3, 3$$

9.1.6 Distribution of multiplication over addition

The `expand` or `fdistrib` command distributes multiplication across addition.

- `expand` takes *expr*, an expression.
- `expand(`*expr*`)` returns the expression *expr* with multiplication distributed with respect to addition.

Example

```
> expand((x+1)*(x-2))
```

or:

```
> fdistrib((x+1)*(x-2))
```

$$x^2 - x - 2$$

9.1.7 Canonical form

The canonical form of a second degree polyomial in a variable x is the form $a(x-c)^2 + b$.

The `canonical_form` command finds the canonical form of a second degree polynomial.

- `canonical_form` takes p, a second degree polynomial.
- `canonical_form(`p`)` returns the canonical form of p.

Examples

```
> canonical_form(x^2-6*x+1)
```

$$(x-3)^2 - 8$$

```
> canonical_form(2*t^2+3*t+8)
```

$$2\left(t + \frac{3}{4}\right)^2 + \frac{55}{8}$$

9.1.8 Multiplication by the conjugate quantity

The `mult_conjugate` tries to remove square roots from the bottom of an expression.

- `mult_conjugate` takes *expr*, an expression. The denominator or numerator is supposed to contain a square root.
- `mult_conjugate(`*expr*`)` returns the following:
  - If *expr* is a fraction and the denominator contains a square root, then this expression is returned with the numerator and denominator multiplied by the conjugate of the denominator.
  - If *expr* is a fraction and the numerator, but not the denominator, contains a square root (if *expr* is not a fraction, it is regarded as a fraction with a denominator of 1), then this expression is returned with the numerator and denominator multiplied by the conjugate of the numerator.

Examples

```
> mult_conjugate((2+sqrt(2))/(2+sqrt(3)))
```

$$\frac{\left(2+\sqrt{2}\right)\left(2-\sqrt{3}\right)}{\left(2+\sqrt{3}\right)\left(2-\sqrt{3}\right)}$$

```
> mult_conjugate((2+sqrt(2))/(sqrt(2)+sqrt(3)))
```

$$\frac{\left(2+\sqrt{2}\right)\left(-\sqrt{2}+\sqrt{3}\right)}{\left(\sqrt{2}+\sqrt{3}\right)\left(-\sqrt{2}+\sqrt{3}\right)}$$

```
> mult_conjugate((2+sqrt(2))/2)
```

$$\frac{\left(2+\sqrt{2}\right)\left(2-\sqrt{2}\right)}{2\left(2-\sqrt{2}\right)}$$

9.1.9 Separation of variables

The `split` command tries to factor an expression involving two variables into the product of two expressions, each of which depends on only one of the variables.

- `split` takes two arguments:
  - *expr*, an expression depending on two variables x and y.
  - $[x, y]$, the list of these two variables.
- `split(`*expr*$,[x,y]$`)` returns a list $[factor_1, factor_2]$, if such a list exists, where *expr*=$factor_1 \cdot factor_2$, $factor_1$ only depends on x and $factor_2$ only depends on y. If such a factorization does not exist, the list `[0]` is returned.

Examples

```
> split((x+1)*(y-2),[x,y])
```

or:

```
> split(x*y-2*x+y-2,[x,y])
```

$$[x+1, y-2]$$

```
> split((x^2*y^2-1,[x,y])
```

$$[0]$$

9.1.10 Factoring

The `factor` and `cfactor` commands factor expressions over their coefficient fields or extensions of their fields. (See also Section 11.1.18, p. 219.)

- `factor` takes one mandatory argument and one optional argument:
  - *expr*, an expression or a list of expressions.
  - Optionally, α, to specify an extension field.

- `factor(`*expr*`)` returns *expr* factored over the field of its coefficients, with the addition of i in complex mode (see Section 2.5.5, p. 14). If sqrt is enabled in the CAS configuration (see Section 2.5.7, p. 15), polynomials of order 2 are factorized in complex mode or in real mode if the discriminant is positive.

  `factor(`*expr*$,\alpha$`)` returns *expr* factored over $F[\alpha]$, where F is the field of coefficients of *expr*.

- `cfactor` factors like `factor`, except the field includes i whether in real or complex mode.

Examples

Factor x^4-1 over $\mathbb{Q}$.

```
> factor(x^4-1)
```

$$(x-1)\,(x+1)\left(x^2+1\right)$$

The coefficients are rationals, hence the factors are polynomials with rationals coefficients.

Factor x^4-1 over $\mathbb{Q}[i]$. This can be done in several ways.

Using `cfactor`:

```
> cfactor(x^4-1
```

or, using `factor` with adding i to the extension field:

```
> factor(x^4-1,i)
```

or, using `factor` in complex mode:

```
> factor(x^4-1)
```

$$(x-1)\,(x+1)\,(x+\mathrm{i})\,(x-\mathrm{i})$$

Factor x^4+1 over $\mathbb{Q}$.

```
> factor(x^4+1)
```

$$x^4+1$$

Indeed, x^4+1 has no factor with rational coefficients.

Factor x^4+1 over $\mathbb{Q}[i]$. Using complex mode:

```
> cfactor(x^4+1)
```

$$\left(x^2+\mathrm{i}\right)\left(x^2-\mathrm{i}\right)$$

Factor x^4+1 over $\mathbb{R}$.

You have to provide the square root required for extending the rationals. In order to do that with the help of Xcas, first check the complex box in the CAS configuration:

```
> solve(x^4+1,x)
```

$$\left[\frac{1}{2}\sqrt{2}\,(1-\mathrm{i})\,,-\frac{1}{2}\sqrt{2}\,(1-\mathrm{i})\,,-\frac{1}{2}\sqrt{2}\,(1-\mathrm{i})\,\mathrm{i},\frac{1}{2}\sqrt{2}\,(1-\mathrm{i})\,\mathrm{i}\right]$$

The roots depend on $\sqrt{2}$, and so will be in $\mathbb{Q}[\sqrt{2}]$. Putting Xcas back in real mode, either check the sqrt box in the CAS configuration or:

```
> factor(x^4+1,sqrt(2))
```

$$\left(x^2-\sqrt{2}x+1\right)\left(x^2+\sqrt{2}x+1\right)$$

To factor over $\mathbb{C}$ input:

```
> cfactor(x^4+1,sqrt(2))
```

or put Xcas back in complex mode.

9.1.11 Zeros of an expression

The `zeros` command finds the zeros of an expression.

- `zeros` takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally, x, a variable name to use (which by default will be `x`).
- `zeros(`*expr*$\langle, x\rangle$`)` returns a list of values of the variable where the expression vanishes. The list may be incomplete in exact mode if the expression is not a polynomial or if intermediate factorizations have irreducible factors of order strictly greater than 2.

  In real mode, (which means the complex box is unchecked in the CAS configuration (see Section 2.5.7, p. 15) or with `complex_mode:=0`), only reals zeros are returned. With (`complex_mode:=1`), real and complex zeros are returned. `cZeros` behaves like `zeros`, except that it returns complex zeros whether in real or complex mode.

Examples

Input in real mode:

```
> zeros(x^2+4)
```

$$[]$$

Input in complex mode:

```
> zeros(x^2+4)
```

$$[-2\mathrm{i}, 2\mathrm{i}]$$

Input in real or complex mode:

```
> cZeros(x^2+4)
```

$$[-2\mathrm{i}, 2\mathrm{i}]$$

Input in real mode:

```
> zeros(ln(x)^2-2)
```

$$\left[\mathrm{e}^{\sqrt{2}}, \mathrm{e}^{-\sqrt{2}}\right]$$

Input in real mode:

```
> zeros(ln(y)^2-2,y)
```

$$\left[\mathrm{e}^{\sqrt{2}}, \mathrm{e}^{-\sqrt{2}}\right]$$

Input in real mode:

```
> zeros(x*(exp(x))^2-2*x-2*(exp(x))^2+4)
```

$$\left[\frac{\ln(2)}{2}, 2\right]$$

9.1.12 Substituting a variable by a value

The | operator is an infixed operator that evaluates an expression after giving values to some variables. It does not evaluate the expression before the variables are replaced by the requested values.

- | is an infixed operator, hence it takes two arguments:
  - *expr*, an expression depending on one or more variables on the left hand side.
  - $x_1 = a_1$, ldots; an equality or sequence of several equalities.
- *expr*|$x_1 = a_1, \ldots$ returns the expression *expr* with x_1 replaced by a_1, etc.

Examples

The command lines below work even if `a` and/or `b` have been assigned.

```
> a^2+1 | a=2
```

5

```
> a^2+b | a=2,b=3
```

7

9.1.13 Substituting a variable by a value

The `subst` command replaces specified variables in an expression by specified values. Unlike the | operator, the `subst` command evaluates the expression before replacing the variables. Since `subst` does not quote its argument, in a normal evaluation process the substitution variable should be purged (see Section 3.3.8, p. 39), otherwise it will be replaced by its assigned value before substitution is done.

The `subst` command can specify the values of variables in two different ways.

- `subst` can take two arguments:
  - *expr*, an expression.
  - *eqs*, an equation of the form $x = a$, or a list of such equalities.
- `subst(`*expr*, *eqs*`)` returns the expression with the variables replaced by their values.
- Alternatively, `subst` can take three arguments:
  - *expr*, an expression.
  - *vars*, a variable or a list of variables.
  - *vals*, a value or a list of values for substitution.
- `subst(`*expr*, *vars*, *vals*`)` returns the expression with the variables replaced by their values.

Examples

Assuming that `a` and `b` are not assigned:

```
> subst(a^2+1,a=2)
```

5

```
> subst(a^2+b,[a=2,b=1])
```

5

```
> subst(a^2+1,a,2)
```

$$5$$

```
> subst(a^2+b,[a,b],[2,1])
```

$$5$$

Changing integration variables. `subst` may also be used to make a change of variable in an integral. In this case the `integrate` command (see Section 13.3.1, p. 283) should be quoted (see Section 9.1.4, p. 170, otherwise, the integral would be computed before substitution) or the inert form `Int` should be used. In both cases, the name of the integration variable must be given as an argument of `Int` or `integrate` even you are integrating with respect to `x`.

Examples

```
> subst('integrate(sin(x^2)*x,x,0,pi/2)',x=sqrt(t))
```

or:

```
> subst(Int(sin(x^2)*x,x,0,pi/2),x=sqrt(t))
```

$$\int_0^{\frac{\pi^2}{4}} \frac{1}{2}\sin t\sqrt{t}\sqrt{t}^{-1}\,\mathrm{d}t$$

```
> subst('integrate(sin(x^2)*x,x)',x=sqrt(t))
```

or:

```
> subst(Int(sin(x^2)*x,x),x=sqrt(t))
```

$$\int \frac{1}{2}\sin t\sqrt{t}\sqrt{t}^{-1}\,\mathrm{d}t$$

9.1.14 Substituting a variable by a value

Another way to substitute a variable by a value, besides with the `|` operator or the `subst` command, is with something akin to functional notation. You can follow an expression or expression name with equalities of the form *variable*=*value*.

Examples

```
> Expr:=x+2*y+3*z
```

then:

```
> subst(Expr,[x=1,y=2])
```

or:

```
> Expr | x=1, y=2
```

or:

```
> Expr(x=1,y=2)
```

$$5+3z$$

```
> (h*k*t^2+h^3*t^3)(t=2)
```

$$4hk+8h^3$$

9.1.15 Substituting a variable by a value (Maple and MuPAD compatibility)

In Maple and in MuPAD, you would use the `subs` command to substitute a variable by a value in an expression. But the order of the arguments differ between Maple and MuPAD. Therefore, to achieve compatibility, in Xcas, the `subs` command arguments order depends on the mode (see Section 2.5.2, p. 14).

- In Maple mode, `subs` takes two arguments:
  - *eq*, an equality or list of equalities of the form *var*=*value*.
  - *expr*, an expression.
- `subs(`*eq*, *expr*`)` returns the expression with the variables replaced by their given values.

Examples

Input in Maple mode (if the variable `a` is purged, otherwise first enter `purge(a)`):

```
> subs(a=2,a^2+1)
```

5

Input in Maple mode (if the variables `a` and `b` are purged, otherwise first enter `purge(a,b)`):

```
> subs([a=2,b=1],a^2+b)
```

5

In MuPAD or Xcas or TI modes, `subs` behaves like `subst` (see Section 9.1.13, p. 175).

- `subst` takes two or three arguments:
  - *expr*, an expression.
  - *eqs*, an equality of the form *var*=*value* or a list of such equalities, or *vars*, *vals*, a variable or list of variables followed by a value or a list of values for substitution.
- `subs(`*expr*, *eqs*`)` or `subs(`*expr*, *vars*, *vals*`)` returns the expression with the variables replaced by their given values.

Examples

Input in MuPAD or Xcas or TI modes (if the variable `a` is purged, otherwise first enter `purge(a)`):

```
> subs(a^2+1,a=2)
```

or:

```
> subs(a^2+1,a,2)
```

5

Input in MuPAD or Xcas or TI modes (if the variables `a` and `b` are purged, otherwise first enter `purge(a,b)` first):

```
> subs(a^2+b,[a=2,b=1])
```

or:

```
> subs(a^2+b,[a,b],[2,1])
```

5

Note that `subs` does not quote its argument, hence in a normal evaluation process, the substitution variable should be purged otherwise it will be replaced by its assigned value before substitution is done.

9.1.16 Substituting a subexpression by another expression

The `algsubs` command replaces subexpressions of an expression, rather than just replace variables.

- `algsubs` takes two arguments:
  - $expr_1$=$expr_2$, an equation between two expressions.
  - *expr*, another expression.
- `algsubs(`$expr_1$=$expr_2$, *expr*`)` returns the last expression *expr* with $expr_1$ replaced by $expr_2$.

Examples

```
> algsubs(x^2=u,1+x^2+x^4)
```

$$u^2+u+1$$

```
> algsubs(a*b/c=d,2*a*b^2/c)
```

$$2*b*d$$

```
> algsubs(2a=p^2-q^2,algsubs(2c=p^2+q^2, c^2-a^2))
```

$$p^2q^2$$

9.1.17 Eliminating one or more variables from a list of equations

The `eliminate` command eliminates variables from a list of equations.

- `eliminate` takes two arguments:
  - *eqns*, a list of equations.
  - *vars*, the variable or list of variables to eliminate. The equations can be given as expressions, in which case they will be assumed to be 0.
- `eliminate(`*eqns*, *vars*`)` returns the equations with the variables *vars* eliminated or an indication that Xcas cannot eliminate them.

Examples

Assuming the variables used haven't been set to any values:

```
> eliminate([x=v0*t,y=y0-g*t^2], t)
```

$$\left[gx^2+yv_0^2-v_0^2y_0\right]$$

```
> eliminate([x+y+z+t-2,x*y*t=1,x^2+t^2=z^2],[x,z])
```

$$\left[2t^2y^2-4t^2y+ty^3-4ty^2+4ty+2t+2y-4\right]$$

If the variable(s) cannot be eliminated, then `eliminate` returns `[1]` or `[-1]`. If `eliminate` returns an empty list, that means the equations determine the values of the variables to be eliminated.

```
> x:=2;y:=-5
  eliminate([x=2*t,y=1-10*t^2],t)
```

$$[1]$$

since `t` cannot be eliminated from both equations.

```
> x:=2;y:=-9
  eliminate([x=2*t,y=1-10*t^2],t)
```

$$[]$$

since the first equation gives $\mathtt{t} = 1$, which satisfies the second equation.

```
> x:=2; y:=-9
  eliminate([x=2*t,y=1-10*t^2,z=x+y-t],t)
```

$$[1, z + 8]$$

since the first equation gives $\mathtt{t} = 1$, which satisfies the second equation, and so that leaves $\mathtt{z} = 2-9-1 = -8$, or $\mathtt{z} + 8 = 0$.

9.1.18 Primitive evaluation at boundaries

The `preval` command evaluates an expression from one value to another, such as in done when evaluating a definite integral using the Fundamental Theorem of Calculus.

- `preval` takes three arguments:
  - F, an expression depending on the variable `x`.
  - a and b, two expressions.
- `preval`(F, a, b) returns $F\big|_{x=b} - F\big|_{x=a}$.

`preval` is used to compute a definite integral when the primitive F of the integrand f is known. Assume, for example, that `F:=int(f,x)`, then `preval(F,a,b)` is equivalent to `int(f,x,a,b)`, but does not require you to recompute `F` from `f` if you change the values of a or b.

Example

```
> preval(x^2+x,2,3)
```

$$6$$

9.1.19 Extracting subexpressions

The `part` command extracts subexpressions from an expression. (See Section 2.9.2, p. 28.)

- `part` takes two arguments:
  - *expr*, an expression.
  - n, an integer.
- `part`$(expr, n)$ evaluates *expr* and then returns the nth sub-expression of *expr*.

Examples

```
> part(x^2+x+1,2)
```

$$x$$

```
> part(x^2+(x+1)*(y-2)+2,2)
```

$$(x + 1)(y - 2)$$

```
> part((x+1)*(y-2)/2,2)
```

$$y-2$$

9.2 Periodic functions

9.2.1 Defining periodic expressions

The `periodic` command creates periodic expressions.

- `periodic` takes two or four arguments:
  - *expr*, an expression $f(x)$ where $x \in [a,b\rangle$ for $a < b$.
  - x, a real variable.
  - a, a real number specifying the lower bound for x.
  - b, a real number specifying the upper bound for x.

  The last three arguments can be passed as a single argument x=$a..b$.
- `periodic(`*expr*$,x,a,b$`)` or `periodic(`*expr*$,x$=$a..b$`)` returns a periodic expression

$$g(x) = f\left(x-(b-a)\left\lfloor\frac{x-a}{b-a}\right\rfloor\right), \quad x \in \mathbb{R},$$

  with period $T = b-a$, satisfying the property $g(x+T) = g(x)$ for all $x \in \mathbb{R}$.

Example

To define the periodic function $f(x) = 1-x^2$ for $-1 \le x < 1$ with period $T = 2$, input:

```
> f:=periodic(1-x^2,x,-1,1)
```

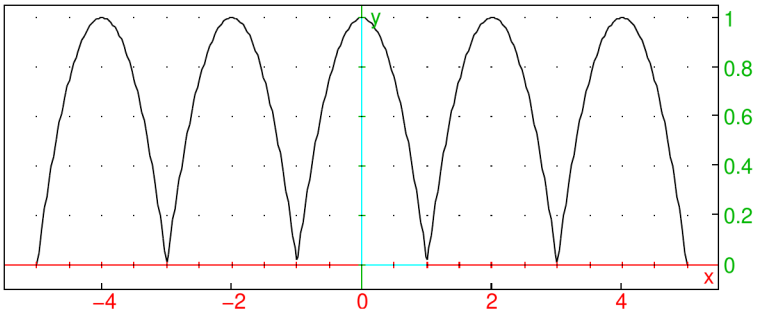
or:

```
> f:=periodic(1-x^2,x=-1..1)
```

$$-\left(-2\left\lfloor\frac{x+1}{2}\right\rfloor+x\right)^2+1$$

Indeed:

```
> plot(f,x=-5..5)
```



9.2.2 Finding a period of an expression

The `period` command finds a period of a given periodic expression.

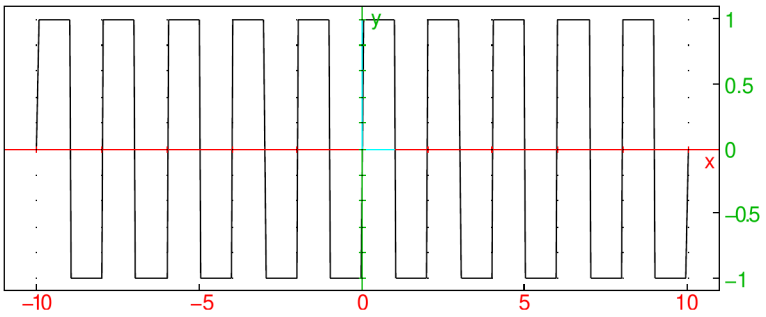
- `period` takes one mandatory argument and one optional argument:
  - *expr*, an expression $f(x)$ where $x \in \mathbb{R}$.
  - Optionally, x, a real variable (by default `x`).
- `period(`*expr*$\langle$`,`*x*$\rangle$`)` returns a (not necessarily the smallest) period of f or $+\infty$ if f is not periodic. Note that $+\infty$ is also returned if `period` is unable to find a period. For periodic functions with arbitrary small periods, such as constant functions, zero is returned.
- `period` is able to detect periodicity of functions built from the basic periodic functions $g_a(x) = \mathrm{e}^{\mathrm{i}ax}$ (which covers for the standard trigonometric functions by Euler's formula) and/or functions returned by the `periodic` command (see Section 9.2.1, p. 180).

Remark. If `periodic` returns zero for an expression $f(x)$, it does not always mean that f is constant. For example, $f(x) = \lfloor x - \lfloor x-1 \rfloor \rfloor$ is always equal to 1 because $1 \leq x - \lfloor x-1 \rfloor < 2$, but $g(x) = \lceil x - \lfloor x-1 \rfloor \rceil$ is not constant by the same argument (for $x \in \mathbb{Z}$, $g(x)$ is equal to 1, and for $x \notin \mathbb{Z}$ its value is 2). However, `periodic` returns zero for both functions, as it is unable to detect changes in isolated points.

Examples

To define and display a square wave, you can enter:

```
> sw:=sign(sin(pi*x)):;
  plot(sw,x)
```



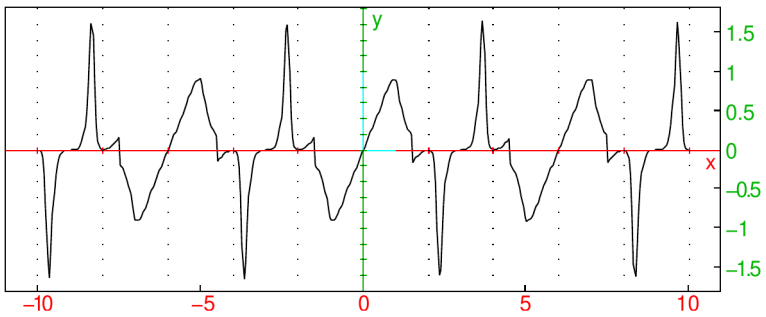
You observe that the period is equal to 2. Indeed:

```
> period(sw)
```

2

Any rational function with periodic variables is periodic if the variable periods are commeasurable (i.e. if the quotient of any two periods is a rational number). Also, if f is periodic, then $g \circ f$ is periodic for any function g. For example, define two periodic functions f and g with periods 2 and 3, respectively, and then the function $h(x) = \frac{f(x)g(x)}{\ln(1+f(x)^2+g(x)^4)}$:

```
> f:=periodic(x^2,x=-1..1):;
  g:=periodic(x^3,x=-3/2..3/2):;
  h:=f*g/ln(1+f^2+g^4):;
  plot(h)
```



The above graph indicates that h is possibly periodic; indeed, its period is equal to 6, which is the least common multiple of the periods of f and g.

```
> period(h)
```

$$6$$

A function built from periodic functions with periods which are not commeasurable is not periodic. For example, $f(x) = \sin(x) + \sin(\pi x)$ is not periodic:

```
> period(sin(x)+sin(pi*x),x)
```

$$+\infty$$

Often (but generally not), the smallest period is returned, such as for the function $f(x) = \sin^2 x$, for instance.

```
> period(sin(x)^2,x)
```

$$\pi$$

9.3 Equations

9.3.1 Defining an equation

The `equal` command creates equations; it is the infixed version of `=`.

- `equal` takes two arguments: *lhs* and *rhs*, the two sides of the equation.
- `equal(`*lhs*, *rhs*`)` returns the equation $lhs = rhs$.

Example

```
> equal(2x-1,3)
```

or:

```
> 2*x-1=3
```

$$2x - 1 = 3$$

9.3.2 Transforming an equation into a difference

The `equal2diff` command turns an equation into the difference of the two sides, resulting in an expression assumed to be equal to 0.

- `equal2diff` takes *lhs*`=`*rhs*, an equation.
- `equal2diff(`*lhs*`=`*rhs*`)` returns the difference $lhs - rhs$.

Example

```
> equal2diff(2x-1=3)
```

$$2x - 1 - 3$$

9.3.3 Transforming an equation into a list

The `equal2list` command separates the two sides of an equation.

- `equal2list` takes *lhs*=*rhs*, an equation.
- `equal2list(`*lhs*=*rhs*`)` returns the sequence *lhs*, *rhs*.

Example

```
> equal2list(2x-1=3)
```

$$2x - 1, 3$$

9.3.4 Left side of an equation

(See Section 5.2.4, p. 56, Section 8.2.3, p. 149, Section 6.5.1, p. 98, Section 6.6.2, p. 100, Section 6.3.6, p. 80, Section 9.3.5, p. 183, Section 28.2.8, p. 826 and Section 28.2.9, p. 827 for other uses of `left`.)
The `left` or `lhs` command finds the left hand side of an equation.

- `left` takes *lhs*=*rhs*, an equation.
- `left(`*lhs*=*rhs*`)` returns *lhs*.

Example

```
> left(2x-1=3)
```

$$2x - 1$$

9.3.5 Right side of an equation

(See Section 5.2.4, p. 56, Section 8.2.3, p. 149, Section 6.5.1, p. 98, Section 6.6.2, p. 100, Section 6.3.6, p. 80, Section 9.3.4, p. 183, Section 28.2.8, p. 826 and Section 28.2.9, p. 827 for other uses of `right`.)
The `right` or `rhs` command finds the right hand side of an equation.

- `right` takes *lhs*=*rhs*, an equation.
- `right(`*lhs*=*rhs*`)` returns *rhs*.

Example

```
> right(2x-1=3)
```

$$3$$

9.3.6 Solving equation(s)

The `solve` command solves an equation or a system of polynomial equations. In real mode, `solve` returns only real solutions; to have `solve` return the complex solutions, switch to complex mode (e.g.by checking the complex box in the CAS configuration, see Section 2.5.5, p. 14).

The `cSolve` command is identical to `solve`, except it returns the complex solutions whether in real mode or complex mode.

With one variable. `solve` can solve equations involving a single unknown.

- `solve` takes one mandatory argument and one optional argument:
  - *eqn*, an equation or expression assumed to be zero.
  - Optionally, x, a variable (by default, x=`x`).
- `solve(`*eqn*$,x$`)` returns the solution to the equation.

For trigonometric equations, `solve` returns by default the principal solutions. To have all the solutions, check the All_trig_sol box in the CAS configuration (see Section 2.5.7, p. 15, item 2.5.7).

Examples

Solve $x^4-1=3$ (in real mode):

```
> solve(x^4-1=3)
```

$$\left[\sqrt{2},-\sqrt{2}\right]$$

In complex mode:

```
> solve(x^4-1=3)
```

$$\left[\sqrt{2},-\sqrt{2},\mathrm{i}\sqrt{2},-\mathrm{i}\sqrt{2}\right]$$

Also (in any mode):

```
> cSolve(x^4-1=3)
```

$$\left[-\sqrt{2},\sqrt{2},-\sqrt{2}\mathrm{i},\sqrt{2}\mathrm{i}\right]$$

Solve $\mathrm{e}^x=2$:

```
> solve(exp(x)=2)
```

$$[\ln(2)]$$

Solve $\cos(2x)=1/2$:

```
> solve(cos(2*x)=1/2)
```

$$\left[-\frac{\pi}{6},\frac{\pi}{6}\right]$$

With the box All_trig_sol checked in CAS configuration:

```
> solve(cos(2*x)=1/2)
```

$$\left[\frac{6\pi n_0+\pi}{6},\frac{6\pi n_0-\pi}{6}\right]$$

With several equations and variables.

- `solve` takes one mandatory argument and one optional argument:
  - *eqns*, a list of polynomial equations.
  - *vars*, a list of variables.
- `solve(`*eqns*, *vars*`)` returns the solutions to the system of equations.

Examples

Find x, y such that $x + y = 1$ and $x - y = 0$:

```
> solve([x+y=1,x-y],[x,y])
```

$$\left[\left[\frac{1}{2}, \frac{1}{2}\right]\right]$$

Find x, y such that $x^2 + y = 2$ and $x + y^2 = 2$:

```
> solve([x^2+y=2,x+y^2=2],[x,y])
```

$$\left[[1,1], [-2,-2], \left[\frac{\sqrt{5}+1}{2}, -\left(\frac{\sqrt{5}+1}{2}\right)^2 + 2\right], \left[\frac{-\sqrt{5}+1}{2}, -\left(\frac{-\sqrt{5}+1}{2}\right)^2 + 2\right]\right]$$

Find x, y, z such that $x^2 - y^2 = 0$ and $x^2 - z^2 = 0$:

```
> solve([x^2-y^2=0,x^2-z^2=0],[x,y,z])
```

$$[[x,x,x], [x,-x,-x], [x,x,-x], [x,-x,x]]$$

Find the intersection of a straight line (given by a list of equations) and a plane.

For example, let D be the straight line with cartesian equations $[y - z = 0, z - x = 0]$ and let P the plane with equation $x - 1 + y + z = 0$. Find the intersection of D and P.

```
> solve([[y-z=0,z-x=0],x-1+y+z=0],[x,y,z])
```

$$\left[\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]\right]$$

Find complex solutions of the system $y - x^2 = 2$, $x^2 y = 0$:

```
> cSolve([-x^2+y=2,x^2+y],[x,y])
```

$$[[-\mathrm{i}, 1], [\mathrm{i}, 1]]$$

9.4 Utility functions

9.4.1 Replacing small values by zero

The `epsilon2zero` command replaces values close enough to zero by 0.

- `epsilon2zero` takes *expr*, an expression in `x`.
- `epsilon2zero(`*expr*`)` returns *expr* where any values of modulus less than `epsilon` (see Section 2.5.7, p. 15, item 2.5.7, by default `epsilon` = 1e−12) are replaced by zero. The expression is not evaluated.

Examples

With epsilon = 1e−12 in the CAS configuration:

```
> epsilon2zero(1e-13+x)
```

$$0 + x$$

With epsilon = 1e−12:

```
> epsilon2zero((1e-13+x)*100000)
```

$$100000\,(0 + x)$$

With epsilon = 0.0001:

```
> epsilon2zero(0.001+x)
```

$$0.001 + x$$

9.4.2 Finding symbolic variables in an expression

The `lname` or `indets` command finds the symbolic variable names used in an expression.

- `lname` takes *expr*, an expression.
- `lname(`*expr*`)` returns the list of the symbolic variable names used in *expr*.

Examples

```
> lname(x*y*sin(x))
```

$$[x, y]$$

```
> a:=2;assume(b>0);assume(c=3);
  lname(a*x^2+b*x+c)
```

$$[x, b, c]$$

9.4.3 List of variables and of expressions

The `lvar` command finds the variables and non-rational that make up an expression.

- `lvar` takes *expr*, an expression.
- `lvar(`*expr*`)` returns a list of variable names and non-rational expressions such that *expr* its argument is a rational function with respect to the variables and expressions of the list.

Examples

```
> lvar(x*y*sin(x)^2)
```

$$[x, y, \sin x]$$

```
> lvar(x*y*sin(x)^2+ln(x)*cos(y))
```

$$[x, y, \sin x, \ln x, \cos y]$$

```
> lvar(y+x*sqrt(z)+y*sin(x))
```

$$[y, x, \sqrt{z}, \sin x]$$

9.4.4 List of variables of an algebraic expressions

The `algvar` command finds the symbolic variable names in an expression and orders them.

- `algvar` takes *expr*, an expression.
- `algvar(`*expr*`)` returns the list of the symbolic variable names used in *expr*, ordered by the algebraic extensions required to build *expr*.

Examples

```
> algvar(y+x*sqrt(z))
```

$$[[y, x], [z]]$$

```
> algvar(y*sqrt(x)*sqrt(z))
```

$$\begin{bmatrix} y \\ z \\ x \end{bmatrix}$$

```
> algvar(y*sqrt(x*z))
```

$$[[y], [x, z]]$$

```
> algvar(y+x*sqrt(z)+y*sin(x))
```

$$[[y, x, \sin x], [z]]$$

9.4.5 Testing if a variable is in an expression

The `has` command determines whether or not an expression contains a given variable.

- `has` takes two arguments:
  - *expr*, an expression.
  - x, the name of a variable.
- `has(`*expr*$, x$`)` returns 0 if *expr* does not contain x and the 1-based index of x in `lname(`*expr*`)` otherwise.

Examples

```
> has(x*y*sin(x),y)
```

$$2$$

```
> has(x*y*sin(x),z)
```

$$0$$

9.4.6 Numeric evaluation

The `evalf` command finds floating point approximations to the numbers in an expression or a matrix (see also Section 7.3.1, p. 128).

- `evalf` takes one mandatory and one optional argument:
  - *expr*, an expression or a matrix.
  - Optionally, n, a positive integer representing the significant digits (by default `DIGITS`, which itself has a default value of 12; see Section 2.5.1, p. 13).
- `evalf(`*expr*$\langle$`,`$n\rangle$`)` returns *expr* with all the numbers replaced by floating point approximations to n digits.

Examples

```
> evalf(sqrt(2))
```

$$1.41421356237$$

```
> evalf(sqrt(2),20)
```

$$1.4142135623730950488$$

```
> evalf([[1,sqrt(2)],[0,1]])
```

$$\begin{bmatrix} 1.0 & 1.41421356237 \\ 0.0 & 1.0 \end{bmatrix}$$

9.4.7 Rational approximation

Floating point numbers are considered approximations, while integers and rational numbers are considered exact. The `float2rational` or `exact` command finds a rational approximation to a floating point number.

- `float2rational` takes *expr*, an expression.
- `float2rational(`*expr*`)` returns *expr* with all the floating point numbers in *expr* replaced by rational numbers; any floating point number x is replaced by a rational r with $|r - x| < \varepsilon$, where ε is given by epsilon in the CAS configuration (see Section 2.5.7, p. 15, item 2.5.7).

Examples

```
> float2rational(1.5)
```

$$\frac{3}{2}$$

```
> float2rational(1.414)
```

$$\frac{707}{500}$$

```
> float2rational(0.156381102937*2)
```

$$\frac{5144}{16447}$$

```
> float2rational(1.41421356237)
```

$$\frac{114243}{80782}$$

```
> float2rational(1.41421356237^2)
```

$$2$$

10 Rewriting algebraic expressions

10.1 General rewriting and simplification routines

10.1.1 Regrouping expressions

The `regroup` command simplifies expressions.

- `regroup` takes *expr*, an expression.
- `regroup(`*expr*`)` returns *expr* with some straightforward simplifications.

Example

```
> regroup(x+3*x+5*4/x)
```

$$4x + \frac{20}{x}$$

10.1.2 Normal form

The `normal` command takes an expression and considers it to be a rational function with respect to generalized identifiers (which are either true identifiers or transcendental functions replaced by temporary identifiers) with coefficients in $\mathbb{Q}$ or $\mathbb{Q}[i]$ or in an algebraic extension (such as $\mathbb{Q}[\sqrt{2}]$ and finds its expanded irreducible representation.

- `normal` takes *expr*, an expression.
- `normal(`*expr*`)` returns the expanded irreducible representation of *expr*. (See also `ratnormal`, Section 10.1.5, p. 192, for pure rational function or `simplify`, Section 10.1.3, p. 191, if the transcendental functions are not algebraically independent.)

Examples

```
> normal((x-1)*(x+1))
```

$$x^2 - 1$$

```
> normal((1-sin(x))*(1+sin(x))
```

$$-\sin^2 x + 1$$

Remarks.

- Unlike `simplify`, `normal` does not try to find algebraic relations between transcendental functions like $\cos(x)^2 + \sin(x)^2 = 1$.
- It is sometimes necessary to run the `normal` command twice to get a fully irreducible representation of an expression containing algebraic extensions.

10.1.3 Simplifying

The `simplify` command simplifies an expression. It behaves like `normal` for rational functions and algebraic extensions. For expressions containing transcendental functions, `simplify` tries first to rewrite them in terms of algebraically independent transcendental functions. For trigonometric expressions, this requires radian mode (check the radian box in the CAS configuration, see Section 2.5.7, p. 15, or input `angle_radian:=1`).

- `simplify` takes *expr*, an expression.
- `simplify(`*expr*`)` returns a simplified version of *expr*.

Examples

```
> simplify((x-1)*(x+1))
```

$$x^2 - 1$$

```
> simplify(3-54*sqrt(1/162))
```

$$-3\sqrt{2} + 3$$

```
> simplify((sin(3*x)+sin(7*x))/sin(5*x))
```

$$2\cos(2x)$$

10.1.4 Automatic simplification

The `autosimplify` command determines how much simplification Xcas will do automatically when you enter an expression. Note that `autosimplify` only works with Xcas, it does not work with `icas` or any other frontend.

By default, Xcas will apply the `regroup` command (see Section 10.1.1, p. 190) to your input, but the `autosimplify` command can change this to applying another rewriting command to your input, such as `simplify` (see Section 10.1.3, p. 191), `factor` (see Section 9.1.10, p. 172), or even `nop` for no simplification. With no arguments, `autosimplify` will return the current rewriting command. Otherwise:

- `autosimplify` command takes *cmd*, a command that will be used to rewrite the results in Xcas.
- `autosimplify(`*cmd*`)` will tell Xcas to apply *cmd* to subsequent inputs. To change the simplification mode during a session, the `autosimplify` command should be on its own line.

Examples

```
> autosimplify(nop)
```

then:

```
> 1+x^2-2
```

$$1 + x^2 - 2$$

```
> autosimplify(simplify)
```

then:

```
> 1+x^2-2
```

$$x^2 - 1$$

```
> autosimplify(factor)
```

then:

```
> 1+x^2-2
```

$$(x-1)\,(x+1)$$

```
> autosimplify(regroup)
```

then:

```
> 1+x^2-2
```

$$x^2-1$$

10.1.5 Normal form for rational functions

The `ratnormal` command rewrites an expression using its irreducible representation. The expression is viewed as a multivariate rational function with coefficients in $\mathbb{Q}$ (or $\mathbb{Q}[i]$). The variables are generalized identifiers which are assumed to be algebraically independent. Unlike with `normal`, an algebraic extension is considered as a generalized identifier. Therefore `ratnormal` is faster but might miss some simplifications if the expression contains radicals or algebraically dependent transcendental functions.

- `ratnormal` takes *expr*, an expression.
- `ratnormal(`*expr*`)` returns the irreducible representation of *expr*.

Examples

```
> ratnormal((x^3-1)/(x^2-1))
```

$$\frac{x^2+x+1}{x+1}$$

```
> ratnormal((-2x^3+3x^2+5x-6)/(x^2-2x+1))
```

$$\frac{-2x^2+x+6}{x-1}$$

10.1.6 Simplification of expressions involving Dirac delta distribution

The `simplifyDirac` command simplifies expressions involving Dirac delta distribution (see Section 7.3.10, p. 134).

- `simplifyDirac` takes one mandatory argument and a sequence of optional arguments:
  - *expr*, an expression.
  - Optionally, x, a variable with respect to which the simplification is done (by default, $x = \mathtt{x}$).
- `simplifyDirac(`*expr* $\langle, x\,\rangle$`)` returns a simplified form of *expr* as a linear combination of Dirac delta and its derivatives, with coefficients being simplified by using `simplify`.

Examples

Since $\delta(x) = -x\,\delta'(x)$:

```
> simplifyDirac(Dirac(x,1)*x)
```

$$-\delta\,(x)$$

$(x-2)^3\delta^{(3)}\,(1-\frac{x}{2})$ simplifies to:

```
> simplifyDirac((x-2)^3*Dirac(1-x/2,3))
```

$$12\,\delta\,(x-2)$$

By using the identity $x^n\delta^{(n)}(x) = (-1)^n n!\,\delta(x)$, $x\,\delta(x) = 0$ and Laurent series expansions, expressions of the form $f(x)\,\delta^{(n)}(x-a)$ can be written as linear combination of delta function and its derivatives with constant coefficients. For example, with $f(x) = \tan\,(x-\frac{\pi}{2})$, $a = 0$ and $n = 2$:

```
> simplifyDirac(sin(1-cos(2x))*Dirac(x,2))
```

$$4\,\delta\,(x)$$

```
> simplifyDirac(tan(x-pi/2)*Dirac(x,2))
```

$$-\frac{2\,\delta\,(x,1)}{3}+\frac{\delta\,(x,3)}{3}$$

If $f(x) = \frac{p(x)}{q(x)}$ is rational with $p(x) = (x-x_1)(x-x_2)\cdots(x-x_n)$, $x_i \neq x_j$ for $i \neq j$, and $q(x) \neq 0\ \forall x \in \mathbb{R}$, then $\delta\,(f(x))$ expands to $\sum_{k=1}^{n}\frac{\delta(x-x_k)}{f'(x_k)}$. For example:

```
> simplifyDirac(Dirac(x^2-x-6))
```

$$\frac{\delta\,(x+2)}{5}+\frac{\delta\,(x-3)}{5}$$

10.1.7 Replacing signum with Heaviside step function and vice versa

The `sign2Heaviside` command rewrites signum function (see Section 8.3.2, p. 155) to Heaviside step function (see Section 8.3.2, p. 155) in an expression by using the identity $\mathrm{sign}(x) = 2\,\theta(x) - 1$.

- `sign2Heaviside` takes *expr*, an expression.
- `sign2Heaviside(`*expr*`)` returns *expr* with all instances of signum function replaced according to the above formula.

The `Heaviside2sign` command has the same syntax but does the opposite of `sign2Heaviside`, i.e. it replaces all instances of Heaviside function by signum according to the identity $\theta(x) = \frac{\mathrm{sign}(x)+1}{2}$.

10.1.8 Rewriting with Heaviside function

The `linstep` command rewrites piecewise defined expressions, as well as expressions involving step functions (signum and Heaviside) and/or absolute values, as a linear combination of Heaviside functions of linear arguments (see Section 8.3.2, p. 155).

- `linstep` takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally, x, a variable (by default, $x =$ `x`).
- `linstep(`*expr*$\langle$`,`$x\rangle$`)` returns *expr* written as a sum of terms in form $f(x)\,\theta(x-a)$, where θ denotes Heaviside function.

Examples

```
> linstep(Heaviside(x-2)*Heaviside(3-x),x)
```

$$\theta(x-2)-\theta(x-3)$$

```
> linstep(abs(x-1),x)
```

$$\theta(x-1)(2x-2)-x+1$$

```
> linstep(piecewise(x<0,piecewise(x<-1,1,-1),x)*(abs(x-1)+sign(x)),x)
```

$$2x\,\theta(x+1)+\theta(x)\left(-x^2+x\right)+\theta(x-1)\left(2x^2-2x\right)-x$$

10.1.9 Rewriting with absolute values

The `linabs` command attempts rewriting piecewise defined expressions, as well as expressions involving step functions (signum and Heaviside) and/or absolute values, as a linear combination of absolute values of linear arguments (see Section 8.3.2, p. 155).

- `linabs` takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally, x, a variable (by default, $x =$ `x`).
- `linabs(`*expr*$\langle$`,`$x\rangle$`)` returns *expr* written as an expression depending on parameters of form $|x-a|$ instead on step/piecewise expressions.
- `linabs` tries to find a simplest representation by nesting the parameters $|x-a|$.

Examples

```
> linabs(sin(x)*Heaviside(x)-sin(x)*Heaviside(-x))
```

$$\sin|x|$$

```
> f:=piecewise(x<-1,x+5/2,x<2,2-x^2/2,x-2):;
  linabs(f,x)
```

$$\frac{(x+4)|x-2|}{4}+\frac{(-x-1)|x+1|}{4}+x+\frac{1}{4}$$

`linabs` is useful for expanding expressions which contain nested absolute values. For example:

```
> linabs(abs(abs(x-2)+x*abs(x+1)+1))
```

$$x^2+x|x+1|+(-x+1)|x+3|+|x-2|+2x-2$$

```
> linabs(abs(1-2*abs(2-3*abs(x-3))))
```

$$-6|x-3|+3|2x-5|+3|2x-7|-2|3x-7|-2|3x-11|+|6x-13|+|6x-23|-5$$

10.1.10 Rewriting an expression with different options

Xcas has many commands to convert expressions into different forms; the `convert` command (or its infixed version `=>`) is a different way to call most of these functions.

- `convert` takes two or more arguments:
  - *expr*, an expression.
  - *option*, an option specifying which rewrite rules to use. A third argument might be necessary for some options. Possible values of *option* are:
    - `sin`, to convert an expression like `trigsin` (see Section 10.2.23, p. 204).
    - `cos`, to convert an expression like `trigcos` (see Section 10.2.24, p. 205).
    - `sincos`, to convert an expression like `sincos` (see Section 10.2.13, p. 201).
    - `trig`, to convert an expression like `sincos` (see Section 10.2.13, p. 201).
    - `tan`, to convert an expression like `halftan` (see Section 10.2.19, p. 203).
    - `exp`, to convert an expression like `trig2exp` (see Section 10.2.21, p. 204).
    - `ln`, to convert an expression like `trig2exp` (see Section 10.2.21, p. 204).
    - `expln`, to convert an expression like `trig2exp` (see Section 10.2.21, p. 204).
    - `string`, to convert a expression into a string.
    - `matrix`, to convert a list of lists into a matrix.
    - `array`, to turn a table into an array (see Section 14.4.2, p. 349).
    - `polynom`, to convert a series (see Section 13.5.2, p. 302) into a polynomial by removing the remainder (see Section 11.1.25, p. 223) or to convert a list representing a polynomial into a polynomial in internal sparse multivariate form (see Section 11.1.2, p. 211 and Section 11.1.6, p. 213).
    - `parfrac` (or `partfrac` or `fullparfrac`), to convert a rational function into its partial fraction decomposition (see Section 11.6.9, p. 251).
    - `interval`, to convert an expression which evaluates to a number into an interval (see Section 6.6.8, p. 103).
    - `list` (or no argument), to convert a polynomial in internal sparse multivariate format (see Section 11.1.2, p. 211) into a list.
    - *unit*, a unit, to convert a unit object to a new compatible unit (see Section 24.1.4, p. 644).

    The values of *option* that require a third argument:
    - `contfrac`, to convert a number into a continued fraction. (See Section 7.2.7, p. 124.) The third argument will be the name of a variable to store the continued fraction into (which must be quoted the variable was assigned).
    - `base`, to convert a number into a different base (beginning with the units digit). If `expr` is a number, then the third argument will be base to convert to (see Section 5.4.2, p. 65), if *expr* is a list of numbers, then the third argument will be the base to convert from (and *expr* will be a list of the digits in this base, starting with the units digit).

    Finally, if *expr* is an expression with units (see Section 24.1.1, p. 642), then *option* can be new units to convert to (see Section 24.1.4, p. 644).
- `convert(`*expr*, *option* ⟨,*extraop*⟩`)` returns the expression with the requested conversions done.

Examples

```
> convert(1.2,confrac,'fc')
```

$$[1, 5]$$

and `fc` contains the continued fraction equal to 1.2.

```
> convert(123,base,10)
```

$$[3, 2, 1]$$

```
> convert([3,2,1],base,10)
```

$$123$$

```
> convert(1000_g,_kg)
```

$$1.0\ \mathrm{kg}$$

10.2 Trigonometry

Xcas can evaluate the trigonometric functions in either radians or degrees (see Section 8.3.2, p. 155). It can also manipulate them algebraically.

10.2.1 Expanding a trigonometric expression

The `trigexpand` command expands sums, differences and products by an integer inside the trigonometric functions.

- `trigexpand` takes *expr*, an expression containing trigonometric functions.
- `trigexpand(`*expr*`)` returns the expression with sums, differences and integer products inside the trigonometric functions expanded.

```
> trigexpand(cos(x+y))
```

$$\cos x \cos y - \sin x \sin y$$

10.2.2 Linearizing a trigonometric expression

The `tlin` command linearizes products and integer powers of the trigonometric functions (e.g. in terms of $\sin(n\,x)$ and $\cos(n\,x)$).

- `tlin` takes *expr*, an expression containing trigonometric functions.
- `tlin(`*expr*`)` returns the expression with the trigonometric functions linearized.

Examples

Linearize $\cos(x)\cos(y)$.

```
> tlin(cos(x)*cos(y))
```

$$\frac{\cos(x-y)}{2}+\frac{\cos(x+y)}{2}$$

Linearize $\cos(x)^3$.

```
> tlin(cos(x)^3)
```

$$\frac{3}{4}\cos x+\frac{\cos(3x)}{4}$$

Linearize $4\cos(x)^2-2$.

```
> tlin(4*cos(x)^2-2)
```

$$2\cos(2x)$$

10.2.3 Increasing the phase by $\pi/2$ in a trigonometric expression

The `shift_phase` command increases the phase of a trigonometric expression by $\pi/2$.

- `shift_phase` takes *expr*, a trigonometric expression.
- `shift_phase(`*expr*`)` returns *expr* with the phase increased by $\pi/2$ (after automatic simplification).

Examples

```
> shift_phase(x+sin(x))
```

$$x-\cos\left(\frac{\pi+2x}{2}\right)$$

```
> shift_phase(x+cos(x))
```

$$x+\sin\left(\frac{\pi+2x}{2}\right)$$

```
> shift_phase(x+tan(x))
```

$$x-\frac{1}{\tan\left(\frac{\pi+2x}{2}\right)}$$

Quoting the argument will prevent the automatic simplification.

```
> shift_phase('sin(x+pi/2)')
```

$$-\cos\left(\frac{\pi+2x+2\frac{\pi}{2}}{2}\right)$$

With an unquoted sine, you get:

```
> shift_phase(sin(x+pi/2))
```

$$\sin\left(\frac{\pi+2x}{2}\right)$$

since `sin(x+pi/2)` is evaluated (in this case simplified) before `shift_phase` is called, and `shift_phase(cos(x))` returns `sin((pi+2*x)/2)`.

10.2.4 Putting together sine and cosine of the same angle

The `tcollect` or `tCollect` command linearizes trigonometric expressions in terms of $\sin(nx)$ and $\cos(nx)$ and combines sines and cosines of the same angle.

- `tcollect` takes *expr*, an expression containing trigonometric functions.
- `tcollect(`*expr*`)` returns *expr* after first linearizing it and then combining sines and cosines of the same angle.

Examples

```
> tcollect(sin(x)+cos(x))
```

$$\sqrt{2}\cos\left(x-\frac{1}{4}\pi\right)$$

```
> tcollect(2*sin(x)*cos(x)+cos(2*x))
```

$$\sqrt{2}\cos\left(2x-\frac{1}{4}\pi\right)$$

10.2.5 Simplifying

The `simplify` command simplifies expressions. As with all automatic simplifications, do not expect miracles; you will have to use specific rewriting rules if it does not work.

- `simplify` takes *expr*, an expression.
- `simplify(`*expr*`)` returns the simplified version of *expr*.

Example

```
> simplify((sin(3*x)+sin(7*x))/sin(5*x))
```

$$2\cos(2x)$$

Remark. `simplify` is more efficient in `radian` mode (which you can turn on, if it is not done already, by checking the radian box in the CAS configuration or inputting `angle_radian:=1`, see Section 2.5.3, p. 14).

10.2.6 Simplifying trigonometric expressions

The `trigsimplify` command simplifies trigonometric expressions by combining `simplify` (see Section 10.1.3, p. 191), `texpand` (see Section 10.4.1, p. 208), `tlin` (see Section 10.2.2, p. 196), `tcollect` (see Section 10.2.4, p. 198), `trigsin` (see Section 10.2.23, p. 204), `trigcos` (see Section 10.2.24, p. 205) and `trigtan` (see Section 10.2.25, p. 205) commands in a certain order.

- `trigsimplify` takes *expr*, an argument containing trigonometric functions.
- `trigsimplify(`*expr*`)` returns the simplified form of *expr*.

Examples

```
> trigsimplify((sin(x+y)-sin(x-y))/(cos(x+y)+cos(x-y)))
```

$$\tan y$$

```
> trigsimplify(1-1/4*sin(2a)^2-sin(b)^2-cos(a)^4)
```

$$\sin^2 a - \sin^2 b$$

10.2.7 Transforming arccos into arcsin

The `acos2asin` command transforms arc cosines in an expression to arc sines by using the identity

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

- `acos2asin` takes `expr`, an expression containing inverse trigonometric functions.
- `acos2asin(`*expr*`)` returns *expr* with all instances of `acos` replaced according to the above formula.

Example

```
> acos2asin(acos(x)+asin(x))
```

Output (after simplification):

$$\frac{\pi}{2}$$

10.2.8 Transforming arccos into arctan

The `acos2atan` command transforms arc cosines in an expression to arc tangents by using the identity

$$\arccos(x) = \frac{\pi}{2} - \arctan\left(\frac{x}{\sqrt{1-x^2}}\right).$$

- `acos2atan` takes `expr`, an expression containing inverse trigonometric functions.
- `acos2atan(`*expr*`)` returns *expr* with all instances of `acos` replaced according to the above formula.

Example

```
> acos2atan(acos(x))
```

$$\frac{\pi}{2} - \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

10.2.9 Transforming arcsin into arccos

The `asin2acos` command transforms any arc sines in an expression to arc cosines by using the identity

$$\arcsin(x) = \frac{\pi}{2} - \arccos(x).$$

- `asin2acos` takes `expr`, an expression containing inverse trigonometric functions.
- `asin2acos(`*expr*`)` returns *expr* with all instances of `asin` replaced according to the above formula.

Example

```
> asin2acos(acos(x)+asin(x))
```

Output (after simplification):

$$\frac{\pi}{2}$$

10.2.10 Transforming arcsin into arctan

The `asin2atan` command transforms arc sines in an expression to arc tangents by using the identity

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right).$$

- `asin2atan` takes `expr`, an expression containing inverse trigonometric functions.
- `asin2atan(`*expr*`)` returns *expr* with all instances of `asin` replaced according to the above formula.

Example

```
> asin2atan(asin(x))
```

$$\arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

10.2.11 Transforming arctan into arcsin

The `atan2asin` command transforms any arc tangents in an expression to arc sines by using the identity

$$\arctan(x) = \arcsin\left(\frac{x}{\sqrt{1+x^2}}\right).$$

- `atan2asin` takes `expr`, an expression containing inverse trigonometric functions.
- `atan2asin(`*expr*`)` returns *expr* with all instances of `atan` replaced according to the above formula.

Example

```
> atan2asin(atan(x))
```

$$\arcsin\left(\frac{x}{\sqrt{1+x^2}}\right)$$

10.2.12 Transforming arctan into arccos

The `atan2acos` command transforms any arc tangents in an expression to arc cosines by using the identity

$$\arctan(x) = \frac{\pi}{2} - \arcsin\left(\frac{x}{\sqrt{1+x^2}}\right).$$

- `atan2acos` takes `expr`, an expression containing inverse trigonometric functions.
- `atan2acos(`*expr*`)` returns *expr* with all instances of `atan` replaced according to the above formula.

Example

```
> atan2acos(atan(x))
```

$$\frac{\pi}{2} - \arccos\left(\frac{x}{\sqrt{1+x^2}}\right)$$

10.2.13 Transforming complex exponentials into sin and cos

The `sincos` or `exp2trig` command uses the identity $e^{ix} = \cos(x) + i\sin(x)$ to rewrite complex exponentials in terms of sine and cosine.

- `sincos` takes *expr*, an expression containing complex exponentials.
- `sincos(`*expr*`)` rewrites *expr* in terms of sin and cos.

Examples

```
> sincos(exp(i*x))
```

$$\cos x + i\sin x$$

```
> exp2trig(exp(-i*x))
```

$$\cos x - i\sin x$$

```
> simplify(sincos(((i)*(exp((i)*x))^2-i)/(2*exp((i)*x))))
```

or:

```
> simplify(exp2trig(((i)*(exp((i)*x))^2-i)/(2*exp((i)*x))))
```

$$-\sin x$$

10.2.14 Transforming $\tan(x)$ into $\sin(x)/\cos(x)$

The `tan2sincos` command replaces $\tan(x)$ by $\frac{\sin(x)}{\cos(x)}$ in an expression.

- `tan2sincos` takes *expr*, an expression containing trigonometric functions.
- `tan2sincos(`*expr*`)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{\sin(x)}{\cos(x)}$.

Example

```
> tan2sincos(tan(2*x))
```

$$\frac{\sin(2x)}{\cos(2x)}$$

10.2.15 Transforming $\sin(x)$ into $\cos(x)\tan(x)$

The `sin2costan` command replaces $\sin(x)$ by $\cos(x)\tan(x)$ in an expression.

- `sin2costan` takes *expr*, an expression containing trigonometric functions.
- `sin2costan(`*expr*`)` returns *expr* with anything of the form $\sin(x)$ replaced by $\cos(x)\tan(x)$.

Example

```
> sin2costan(sin(2*x))
```

$$\tan(2x)\cos(2x)$$

10.2.16 Transforming $\cos(x)$ into $\sin(x)/\tan(x)$

The `cos2sintan` command replaces $\cos(x)$ by $\frac{\sin(x)}{\tan(x)}$ in an expression.

- `cos2sintan` takes *expr*, an expression containing trigonometric functions.
- `cos2sintan(`*expr*`)` returns *expr* with anything of the form $\cos(x)$ replaced by $\frac{\sin(x)}{\tan(x)}$.

Example

```
> cos2sintan(cos(2*x))
```

$$\frac{\sin(2x)}{\tan(2x)}$$

10.2.17 Rewriting $\tan(x)$ in terms of $\sin(2x)$ and $\cos(2x)$

The `tan2sincos2` command replaces $\tan(x)$ by $\frac{\sin(2x)}{1+\cos(2x)}$ in an expression.

- `tan2sincos2` takes *expr*, an expression containing trigonometric functions.
- `tan2sincos2(`*expr*`)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{\sin(2x)}{1+\cos(2x)}$.

Example

```
> tan2sincos2(tan(x))
```

$$\frac{\sin(2x)}{1+\cos(2x)}$$

10.2.18 Rewriting $\tan(x)$ in terms of $\cos(2x)$ and $\sin(2x)$

The `tan2cossin2` command replaces $\tan(x)$ by $\frac{1-\cos(2x)}{\sin(2x)}$ in an expression.

- `tan2cossin2` takes *expr*, an expression containing trigonometric functions.
- `tan2cossin2(`*expr*`)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{1-\cos(2x)}{\sin(2x)}$.

Example

```
> tan2cossin2(tan(x))
```

$$\frac{1-\cos(2x)}{\sin(2x)}$$

10.2.19 Rewriting sin, cos, tan in terms of half tangent

The `halftan` command rewrites the trigonometric functions in terms of $\tan(x/2)$ using the identities:

$$\sin(x) = \frac{2\tan\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right)+1}, \quad \cos(x) = \frac{1-\tan^2\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right)+1}, \quad \tan(x) = \frac{2\tan\left(\frac{x}{2}\right)}{1-\tan^2\left(\frac{x}{2}\right)}.$$

- `halftan` takes *expr*, an expression containing trigonometric functions.
- `halftan(`*expr*`)` returns *expr* with any trigonometric functions replaced by the appropriate expression of `tan(`x`/2)`.

Examples

```
> halftan(sin(2*x)/(1+cos(2*x)))
```

$$\frac{2\tan\left(\frac{2}{2}x\right)}{\left(\tan^2\left(\frac{2}{2}x\right)+1\right)\left(1+\frac{1-\tan^2\left(\frac{2}{2}x\right)}{\tan^2\left(\frac{2}{2}x\right)+1}\right)}$$

```
> normal(ans())
```

$$\tan x$$

```
> halftan(sin(x)^2+cos(x)^2)
```

$$\left(\frac{2\tan\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right)+1}\right)^2 + \left(\frac{1-\tan^2\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right)+1}\right)^2$$

```
> normal(ans())
```

$$1$$

10.2.20 Rewriting trigonometric/hyperbolic functions in terms of half tangent/exponentials

The `halftan_hyp2exp` command rewrites the trigonometric function in terms of $\tan(x/2)$ (like `halftan`, see Section 10.2.19, p. 203) and rewrites the hyperbolic functions in terms of their definitions using exponentials, namely:

$$\sinh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{2}, \quad \cosh(x) = \frac{\mathrm{e}^x + \mathrm{e}^{-x}}{2}, \quad \tanh(x) = \frac{\mathrm{e}^x - \mathrm{e}^{-x}}{\mathrm{e}^x + \mathrm{e}^{-x}} = \frac{\mathrm{e}^{2x}-1}{\mathrm{e}^{2x}+1}.$$

- `halftan_hyp2exp` takes *expr*, a trigonometric and hyperbolic expression.
- `halftan_hyp2exp(`*expr*`)` returns *expr* with any trigonometric functions replaced by the appropriate expression in `tan(`x`/2)` and any hyperbolic functions replaced by the appropriate exponentials.

Examples

```
> halftan_hyp2exp(tan(x)+tanh(x))
```

$$\frac{2\tan\left(\frac{x}{2}\right)}{1-\tan^2\left(\frac{x}{2}\right)} + \frac{\mathrm{e}^{2x}-1}{\mathrm{e}^{2x}+1}$$

```
> normal(halftan_hyp2exp(sin(x)^2+cos(x)^2-sinh(x)^2+cosh(x)^2))
```

$$2$$

10.2.21 Transforming trigonometric functions into complex exponentials

The `trig2exp` command replaces trigonometric functions by their complex exponential form.

- `trig2exp` takes one argument: *expr*, an expression containing trigonometric functions.
- `trig2exp(`*expr*`)` returns *expr* with the trigonometric functions replaced by the appropriate complex exponentials (without linearization).

Examples

```
> trig2exp(tan(x))
```

$$\frac{\left(e^{ix}\right)^2 - 1}{i\left(\left(e^{ix}\right)^2 + 1\right)}$$

```
> trig2exp(sin(x))
```

$$\frac{e^{ix} - \frac{1}{e^{ix}}}{2i}$$

10.2.22 Transforming inverse trigonometric functions into logarithms

Just as the trigonometric functions can be written in terms of complex exponentials, the inverse trigonometric functions can be written in terms of complex logarithms. The `atrig2ln` command does this rewriting.

- `atrig2ln` takes one argument: *expr*, an expression containing inverse trigonometric functions.
- `atrig2ln(`*expr*`)` returns *expr* with any inverse trigonometric functions replaced by the appropriate complex logarithms.

Example

```
> atrig2ln(asin(x))
```

$$i\ln\left(x + \sqrt{x^2 - 1}\right) + \frac{\pi}{2}$$

10.2.23 Simplifying and expressing preferentially with sines

Any trigonometric function can be written in terms of sines and cosines, and even powers of cosines can be turned into powers of sines by using the identity $\sin(x)^2 + \cos(x)^2 = 1$. The `trigsin` command performs these substitutions.

- `trigsin` takes *expr*, an expression containing trigonometric functions.
- `trigsin(`*expr*`)` returns *expr* with the trigonometric functions rewritten in terms of sines and cosines, with as many cosines as possible transformed to sines.

Example

```
> trigsin(sin(x)^4+cos(x)^2+1)
```

$$\sin^4 x - \sin^2 x + 2$$

10.2.24 Simplifying and expressing preferentially with cosines

Any trigonometric function can be written in terms of sines and cosines, and even powers of sines can be turned into powers of cosines by using the identity $\sin(x)^2 + \cos(x)^2 = 1$. The `trigcos` command performs these substitutions.

- `trigcos` takes *expr*, an expression containing trigonometric functions.
- `trigsin(`*expr*`)` returns *expr* with the trigonometric functions rewritten in terms of sines and cosines, with as many sines as possible transformed to cosines.

Example

```
> trigcos(sin(x)^4+cos(x)^2+1)
```

$$\cos^4 x - \cos^2 x + 2$$

10.2.25 Simplifying and expressing preferentially with tangents

The `trigtan` command rewrites trigonometric expressions into expressions where as many trigonometric functions as possible are written in terms of tangents, using the identities $\sin(x)^2 + \cos(x)^2 = 1$, $\tan(x) = \frac{\sin(x)}{\cos(x)}$.

- `trigtan` takes *expr*, an expression containing trigonometric functions.
- `trigtan(`*expr*`)` returns *expr* with the trigonometric functions written as much as possible in terms of tangents.

Example

```
> trigtan(sin(x)^4+cos(x)^2+1)
```

$$\frac{2\tan^4 x + 3\tan^2 x + 2}{\tan^4 x + 2\tan^2 x + 1}$$

10.3 Exponentials and logarithms

10.3.1 Rewriting hyperbolic functions as exponentials

The hyperbolic functions are typically defined in terms of exponential functions; the `hyp2exp` command converts hyperbolic functions into their exponential forms.

- `hyp2exp` takes *expr*, an expression.
- `hyp2exp(`*expr*`)` rewrites each hyperbolic function in *expr* with exponentials (as a rational function of one exponential, i.e. without linearization).

Example

```
> hyp2exp(sinh(x))
```

$$\frac{e^x - \frac{1}{e^x}}{2}$$

10.3.2 Expanding exponentials

The exponential function applied to a sum can be converted into a product of exponentials; namely,

$$e^{x+y} = e^x e^y.$$

The `expexpand` command does this conversion. For expansions in other bases, see Section 10.3.6, p. 207.

- `expexpand` takes *expr*, an expression.
- `expexpand(`*expr*`)` returns the expression *expr* with exponentials (base e) of sums rewritten as products of exponentials.

Example

```
> expexpand(exp(3*x)+exp(2*x+2))
```

$$(e^x)^3 + (e^x)^2 e^2$$

10.3.3 Expanding logarithms

The logarithm applied to a product can be converted into a sum of logarithms; namely,

$$\log(xy) = \log(x) + \log(y).$$

The `lnexpand` command does this expansion.

- `lnexpand` takes *expr*, an expression.
- `lnexpand(`*expr*`)` returns the expression *expr* with logarithms of products rewritten as sums of logarithms.

Example

```
> lnexpand(ln(3*x^2)+ln(2*x+2))
```

$$\ln(3) + 2\ln|x| + \ln(2) + \ln(x+1)$$

10.3.4 Linearizing exponentials

The `lin` command will linearize expressions involving exponentials; namely, it will replace products of exponentials by exponentials of sums. It will first replace any hyperbolic functions by exponentials.

- `lin` takes *expr*, an expression.
- `lin(`*expr*`)` returns the linearized version of *expr*.

Examples

```
> lin(sinh(x)^2)
```

$$\frac{e^{2x}}{4} - \frac{1}{2} + \frac{e^{-2x}}{4}$$

```
> lin((exp(x)+1)^3)
```

$$e^{3x} + 3e^{2x} + 3e^x + 1$$

10.3.5 Collecting logarithms

The `lncollect` command collects the logarithm in an expression; namely, it rewrites sums of logarithms as the logarithm of a product.

- `lncollect` takes *expr*, an expression.
- `lncollect(`*expr*`)` returns *expr* with the logarithms collected.

It may be a good idea to factor the expression with `factor` before collecting by `lncollect`).

Examples

```
> lncollect(ln(x+1)+ln(x-1))
```

$$\ln\left((x+1)\,(x-1)\right)$$

```
> lncollect(exp(ln(x+1)+ln(x-1)))
```

$$(x+1)\,(x-1)$$

Remark. Note that, for Xcas, `log` is the natural logarithm, the same as `ln`; for the base 10 logarithm, use `log10`.

10.3.6 Expanding powers

The `powexpand` command rewrites a power of a sum as a product of powers; it is `expexpand` (see Section 10.3.2, p. 206) with bases other than e.

- `powexpand` takes *expr*, an expression.
- `powexpand(`*expr*`)` returns *expr* with powers of sums replaced by sums of powers.

Example

```
> powexpand(a^(x+y))
```

$$a^x a^y$$

10.3.7 Rewriting a power as an exponential

Powers with arbitrary (positive) bases are often defined in terms of exponentials with base e with

$$a^x = \mathrm{e}^{x\ln(a)}.$$

The `pow2exp` rewrites powers to exponentials.

- `pow2exp` takes *expr*, an exponential.
- `pow2exp(`*expr*`)` returns *expr* with any powers replaced by their corresponding exponential.

Example

```
> pow2exp(a^(x+y))
```

$$\mathrm{e}^{(x+y)\ln a}$$

10.3.8 Rewriting $\exp(n\ln(x))$ as a power

The `exp2pow` command is the inverse of `pow2exp` (see Section 10.3.7, p. 207).

- `exp2pow` takes *expr*, an expression.
- `exp2pow(`*expr*`)` rewrites any subexpressions of `expr` of the form $e^{n\ln(x)}$ as x^n.

Example

```
> exp2pow(exp(n*ln(x)))
```

$$x^n$$

10.3.9 Simplifying complex exponentials

The `tsimplify` command simplifies transcendental expressions by rewriting the expression with complex exponentials. It is a good idea to try other simplification instructions and call `tsimplify` if they do not work.

- `tsimplify` takes *expr*, an expression.
- `tsimplify(`*expr*`)` returns a (possibly) simplified version of *expr*.

Example

```
> tsimplify((sin(7*x)+sin(3*x))/sin(5*x))
```

$$\frac{\left(e^{ix}\right)^4+1}{\left(e^{ix}\right)^2}$$

10.4 Rewriting transcendental expressions

10.4.1 Expanding transcendental expressions

The `texpand` or `tExpand` command expands exponential and trigonometric functions, like simultaneous calling textttexpexpand (see Section 10.3.2, p. 206), which, for example, expands e^{nx} as $(e^x)^n$, `lnexpand` (see Section 10.3.3, p. 206), which, for example, expands $\ln(x^n)$ as $n\ln(x)$, and `trigexpand` (see Section 10.2.1, p. 196), which, for example, expands $\sin(2x)$ as $2\sin(x)\cos(x)$.

- `texpand` takes *expr*, an expression containing transcendental or trigonometric functions.
- `texpand(`*expr*`)` expands these functions.

Examples

Expand $\cos(x+y)$.

```
> texpand(cos(x+y))
```

$$\cos x\cos y-\sin x\sin y$$

Expand $\cos(3x)$.

```
> texpand(cos(3*x))
```

$$4\cos^3 x-3\cos x$$

Expand $\frac{\sin(3x)+\sin(7x)}{\sin(5x)}$.

```
> texpand((sin(3*x)+sin(7*x))/sin(5*x))
```

$$-\frac{2\sin x}{(16\cos^4 x-12\cos^2 x+1)\sin x}+\frac{28\sin x\cos^2 x}{(16\cos^4 x-12\cos^2 x+1)\sin x}-\frac{80\sin x\cos^4 x}{(16\cos^4 x-12\cos^2 x+1)\sin x}+\frac{64\sin x\cos^6 x}{(16\cos^4 x-12\cos^2 x+1)\sin x}$$

```
> normal(ans())
```

$$4\cos^2 x-2$$

Expand e^{x+y}.

```
> texpand(exp(x+y))
```

$$\mathrm{e}^x\mathrm{e}^y$$

Expand $\ln(xy)$.

```
> texpand(log(x*y))
```

$$\ln y+\ln x$$

Expand $\ln(x^n)$.

```
> texpand(ln(x^n))
```

$$n\ln x$$

Expand $\ln\left(\mathrm{e}^2+\mathrm{e}^{2\ln(2)}+\mathrm{e}^{\ln(3)+\ln(2)}\right)$.

```
> texpand(log(e^2)+exp(2*log(2))+exp(log(3)+log(2)))
```

$$6+2\cdot 3$$

or:

```
> texpand(log(e^2)+exp(2*log(2)))+ lncollect(exp(log(3)+log(2)))
```

$$12$$

Expand $\mathrm{e}^{x+y}+\cos(x+y)+\ln(3x^2)$.

```
> texpand(exp(x+y)+cos(x+y)+ln(3*x^2))
```

$$\cos x\cos y-\sin x\sin y+\mathrm{e}^x\mathrm{e}^y+\ln(3)+2\ln|x|$$

10.4.2 Combining terms of the same type

The `combine` command joins subexpressions of various types.

- `combine` takes two arguments:
  - *expr*, an expression.
  - *function*, the name of a function or class of functions. *function* can be one of `exp`, `log`, `ln`, `sin`, `cos`, or `trig`.
- `combine(`*expr*, *function*`)` returns the expression with subexpressions corresponding to the second argument combined.

The `combine` command can duplicate the effect of other commands.

- `combine(`*expr*`,ln)` or `combine(`*expr*`,log)` gives the same result as `lncollect(`*expr*`)` (see Section 10.3.5, p. 207).
- `combine(`*expr*`,trig)` or `combine(`*expr*`,sin)` or `combine(`*expr*`,cos)` gives the same result as `tcollect(`*expr*`)` (see Section 10.2.4, p. 198).

Examples

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),exp)
```

$$\cos x \sin x + \ln x + \ln y + \mathrm{e}^{x+y}$$

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),trig)
```

or:

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),sin)
```

or:

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),cos)
```

$$\mathrm{e}^{x}\mathrm{e}^{y} + \ln x + \ln y + \frac{\sin(2x)}{2}$$

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),ln)
```

or:

```
> combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),log)
```

$$\cos x \sin x + \mathrm{e}^{x}\mathrm{e}^{y} + \ln(xy)$$

11 Polynomials

11.1 Basic functions for polynomials

11.1.1 Polynomials of a single variable

A polynomial of one variable is represented either by a symbolic expression or by the list of its coefficients in decreasing powers order (dense representation). In the latter case, to avoid confusion with other kinds of lists:

- `poly1[...]` is used as delimiters for inputs and for text form output.
- ‖ . . . ‖ is used for XCAS output.

Note that polynomials represented as lists of coefficients are always written in decreasing powers order even if the box Increasing power is checked in CAS configuration (see Section 2.5.7, p. 15).

11.1.2 Polynomials of several variables

A polynomial of several variables can be represented in different ways:

1. by a symbolic expression.
2. by a dense recursive one-dimensional representation like above.
3. by a sum of monomials with non-zero coefficients (distributed sparse representation).

   A monomial with several variables is represented by a coefficient and a list of integers (interpreted as powers of a variable list). The delimiters for monomials are `%%%{` and `%%%}`.

   For example, $3x^2y$ is represented by `%%%{3,[2,1]%%%}` with respect to the variable list `[x,y]`, and $2x^3y^2z - 5xz$ is represented by `%%%{2,[3,2,1]%%%}-%%%{5,[1,0,1]%%%}` with respect to the variable list `[x,y,z]`.

   For a sparse representation, a single variable polynomial can be regarded as a multivariate polynomial with one variable.

11.1.3 Apply a function to the internal sparse format of a polynomial

The `map` command can apply a function to the coefficients of a polynomial written in internal sparse format. (See Section 6.3.28, p. 92 for other uses of `map`.)

- `map` takes two arguments:
  - P, a polynomial of k variables in internal sparse format.
  - f, a function of $k+1$ variables.
- `map(`P, f`)` applies f to the coefficients of P; namely, it returns a polynomial which replaces each term `%%%{`a`,[`$n_1, \ldots, n_k$`]%%%}` in P by `%%%{`$f(a, n_1, \ldots, n_k)$`,[`$n_1, \ldots, n_k$`]%%%}`.

Example

```
> map(%%%{2,[2,1]%%%}+%%%{3,[1,4]%%%},(a,b,c)->a*b*c)
```

$$\%\%\%\{4,[2,1]\%\%\%\}+\%\%\%\{12,[1,4]\%\%\%\}$$

11.1.4 Converting to a symbolic polynomial

The `r2e` or `poly2symb` command converts lists into symbolic polynomials.

- For one-variable polynomials, `r2e` takes one mandatory argument and one optional argument:
  - L, a list of coefficients of a polynomial (in decreasing order),
  - x, a symbolic variable name (by default `x`).
- `r2e(`$L\,\langle,x\rangle$`)` returns the corresponding polynomial with the given variable.
- For sparse multivariate polynomials, `r2e` takes two arguments:
  - S, a sum of monomials of the form `%%%{coeff,[n1,...,nk]%%%}`.
  - *vars*, a vector of symbolic variables.
- `r2e(`$S\,\langle,vars\rangle$`)` returns the corresponding polynomial as an expression with the given variables

.

Examples

```
> r2e([1,0,-1],x)
```

or:

```
> r2e([1,0,-1])
```

or:

```
> poly2symb([1,0,-1],x)
```

$$xx-1$$

```
> poly2symb(%%%{1,[2]%%%}+%%%{-1,[0]%%%},[x])
```

or:

```
> r2e(%%%{1,[2]%%%}+%%%{-1,[0]%%%},[x])
```

$$x^2-1$$

```
> r2e(%%%{1,[2,0]%%%}+%%%{-1,[1,1]%%%}+%%%{2,[0,1]%%%},[x,y])
```

or:

```
> poly2symb(%%%{1,[2,0]%%%}+%%%{-1,[1,1]%%%}+%%%{2,[0,1]%%%},[x,y])
```

$$x^2-xy+2y$$

11.1.5 Converting from a symbolic polynomial

The `e2r` or `symb2poly` command converts a symbolic polynomial into a list (for single variable polynomials) or a sum of monomials.

- `e2r` takes two arguments:
  - P, a symbolic polynomial.
  - *vars*, the variable name (for one variable polynomials) or a list of variable names (for multivariable polynomials).

    For one variable polynomials, this is optional and defaults to `x`.
- `e2r(`P $\langle$`,`*vars*$\rangle$`)` returns the representation of the polynomial as a list of coefficients written in decreasing order, if *vars* is a variable name, or a sum of monomials (sparse representation of multivariate polynomials) if *vars* is a list.

Examples

```
> e2r(x^2-1)
```

or:

```
> symb2poly(x^2-1)
```

or:

```
> symb2poly(y^2-1,y)
```

or:

```
> e2r(y^2-1,y)
```

$$[\![1, 0, -1]\!]$$

```
> e2r(x^2-x*y+y,[x,y])
```

or:

```
> symb2poly(x^2-x*y+2*y,[x,y])
```

$$\%\%\%\{1, [2, 0]\%\%\%\} + \%\%\%\{-1, [1, 1]\%\%\%\} + \%\%\%\{2, [0, 1]\%\%\%\}$$

11.1.6 Transforming a polynomial in internal format into a list and back

The `convert` command does many conversions (see Section 10.1.10, p. 195). Among other things, it can convert between a polynomial in internal sparse multivariate format and a list representing the polynomial.

- To convert from a polynomial in internal sparse multivariate format to a list, `convert` takes one mandatory argument and one optional argument:
  - P, a polynomial written in internal sparse multivariate format (see Section 11.1.2, p. 211).
  - Optionally, `list`.
- `convert(`P $\langle$`, list`$\rangle$`)` returns a list representing the polynomial.
- To convert from a list representing a polynomial to the polynomial in internal sparse multivariate format`convert` takes two arguments:
  - L, a list representing a polynomial.
  - `polynom`.

- `convert(`L`,polynom)` returns the polynomial in internal sparse multivariate format (see Section 11.1.2, p. 211).

Examples

```
> p:=symb2poly(x^2-x*y+2y,[x,y])
```

$$\%\%\%\{1,[2,0]\%\%\%\}+\%\%\%\{-1,[1,1]\%\%\%\}+\%\%\%\{2,[0,1]\%\%\%\}$$

```
> l:=convert(p,list)
```

or:

```
> l:=convert(p)
```

$$\begin{bmatrix} 1 & [2,0] \\ -1 & [1,1] \\ 2 & [0,1] \end{bmatrix}$$

which is a list of the coefficients followed by a list of the variable powers.

```
> convert(l,polynom)
```

$$\%\%\%\{1,[2,0]\%\%\%\}+\%\%\%\{-1,[1,1]\%\%\%\}+\%\%\%\{2,[0,1]\%\%\%\}$$

11.1.7 Coefficients of a polynomial

The `coeff` or `coeffs` command finds the coefficients of a specific degree of a polynomial.

- `coeff` takes two mandatory and one optional argument:
  - P, the polynomial.
  - *vars*, the name of the variable (or the list of the names of variables).
  - Optionally, n, the degree (or the list of the degrees of the variables).
- `coeff(`P*vars*$\langle,n\rangle$`)` returns the nth degree coefficient of P, or if n is not specified, the list of the coefficients of P, including 0 in the univariate dense case and excluding 0 in the multivariate sparse case.

Examples

```
> coeff(-x^4+3*x*y^2+x,x,1)
```

$$3y^2+1$$

```
> coeff(-x^4+3x*y^2+x,y,2)
```

$$3x$$

```
> coeff(-x^4+3x*y^2+x,[x,y],[1,2])
```

$$3$$

11.1.8 Polynomial degree

The `degree` command finds the degree of a polynomial.

- `degree` takes P, a polynomial given by its symbolic representation or by the list of its coefficients.
- `degree(`P`)` returns the degree of P (the highest degree of its non-zero monomials).

Examples

```
> degree(x^3+x)
```

$$3$$

```
> degree([1,0,1,0])
```

$$3$$

11.1.9 Polynomial valuation

The *valuation* of a polynomial is the lowest degree of its non-zero monomials. The `valuation` or `ldegree` command finds the valuation of a polynomal.

- `valuation` takes P, a polynomial given by a symbolic expression or by the list of its coefficients.
- `valuation(`P`)` returns the valuation of P.

Examples

```
> valuation(x^3+x)
```

$$1$$

```
> valuation([1,0,1,0])
```

$$1$$

11.1.10 Leading coefficient of a polynomial

The `lcoeff` command finds the leading coefficient of a polynomial; that is, the coefficient of the monomial of highest degree.

- `lcoeff` takes one mandatory argument and one optional argument:
  - P, a polynomial given by a symbolic expression or by its list of coefficients.
  - Optionally, x, a variable name (by default `x`).
- `lcoeff(`$P\,\langle,x\rangle$`)` returns the leading coefficient of P.

Examples

```
> lcoeff([2,1,-1,0])
```

$$2$$

```
> lcoeff(3*x^2+5*x,x)
```

$$3$$

```
> lcoeff(3*x^2+5*x*y^2,y)
```

$$5x$$

11.1.11 Trailing coefficient degree of a polynomial

The `tcoeff` command finds the trailing coefficient of a polynomial; that is, the coefficient of the monomial of lowest degree.

- `tcoeff` takes one mandatory argument and one optional argument:
  - P, a polynomial given by a symbolic expression or by its list of coefficients.
  - Optionally x, a variable name (by default `x`).
- `tcoeff(`$P\,\langle,x\rangle$`)` returns the trailing coefficient of P.

Examples

```
> tcoeff([2,1,-1,0])
```

$$-1$$

```
> tcoeff(3*x^2+5*x,x)
```

$$5$$

```
> tcoeff(3*x^2+5*x*y^2,y)
```

$$3x^2$$

11.1.12 Polynomial evaluation

The `peval` or `polyEval` command evaluates polynomials.

- `peval` takes two arguments:
  - P, a polynomial given by the list of its coefficients.
  - a, a real number.
- `peval(`P, a`)` returns the exact or numeric value of $P(a)$, calculated using Horner's method.

Examples

```
> peval([1,0,-1],sqrt(2))
```

$$\sqrt{2}\sqrt{2}-1$$

```
> normal(sqrt(2)*sqrt(2)-1)
```

$$1$$

```
> peval([1,0,-1],1.4)
```

$$0.96$$

11.1.13 Polynomial evaluation with Horner algorithm

The `horner` command uses Horner's method to evaluate polynomials.

- `horner` takes two arguments:
  - P, a polynomial given by its symbolic expression or by the list of its coefficients.
  - a, a number.
- `horner(`P, a`)` returns the value $P(a)$, computed using Horner's method.

Example

```
> horner(x^2-2*x+1,2)
```

or:

```
> horner([1,-2,1],2)
```

$$1$$

11.1.14 Rewriting in terms of the powers of $(x-a)$

The `ptayl` command finds the Taylor expansion for a polynomial (which will be finite).

- `ptayl` takes two arguments:
  - P, a polynomial given by a symbolic expression or by the list of its coefficients.
  - a, a number.
- `ptayl(`P, a`)` returns the polynomial T such that $P(x) = T(x-a)$.

Examples

```
> ptayl(x^2+2*x+1,2)
```

Output, the polynomial `T`:

$$x^2+6x+9$$

```
> ptayl([1,2,1],2)
```

$$[1,6,9]$$

i.e. $x^2+2x+1 = (x-2)^2+6(x-2)+9$.

11.1.15 Factoring x^n in a polynomial

The `factor_xn` command factors the largest power of the variable out of a polynomial, writing it as the product of a monomial of largest degree and a rational function having a non-zero finite limit at infinity.

- `factor_xn` takes P, a polynomial.
- `factor_xn(`P`)` returns P written as the product of its monomial of largest degree with a rational function having a non-zero finite limit at infinity.

Example

```
> factor_xn(-x^4+3)
```

$$x^4\left(-1+3x^{-4}\right)$$

11.1.16 GCD of the coefficients of a polynomial

The *content* of a polynomial is the GCD (greatest common divisor) of its coefficients. The `content` command computes the content of a polynomial.

- `content` takes P, a polynomial given by a symbolic expression or by the list of its coefficients.
- `content(`P`)` returns the content of P.

Example

```
> content(6*x^2-3*x+9)
```

or:

```
> content([6,-3,9],x))
```

$$3$$

11.1.17 Primitive part of a polynomial

The primitive part of a polynomial is the polynomial divided by its content (the greatest common divisor of its coefficients). The `primpart` command computes the primitive part of a polynomial.

- `primpart` takes P, a polynomial given by a symbolic expression or by the list of its coefficients.
- `primpart(`P`)` returns the primitive part of P.

Example

```
> primpart(6x^2-3x+9)
```

or:

```
> primpart([6,-3,9],x))
```

$$2x^2-x+3$$

11.1.18 Factoring

The `collect` command factors polynomials over their coefficient fields or extensions of the fields.

- `collect` takes one mandatory and one optional argument:
  - P, a polynomial or a list of polynomials.
  - Optionally, α, a number, such as $\sqrt{n}$, determining an extension field to the field of coefficients of P.
- `collect(`$P\,\langle,\alpha\rangle$`)` returns the factored form of the polynomial (or list of polynomials), where the factorization is done over the field of coefficients (such as $\mathbb{Q}$) or the smallest extension field containing α (e.g. $\mathbb{Q}[\alpha]$). In complex mode (see Section 2.5.7, p. 15), the field is complexified.

The `factor` command (see 9.1.10) will also factor polynomials over their coefficient fields (or extensions of it), but will further factor each factor of degree 2 if the sqrt box is checked in the CAS configuration.

Examples

Factor x^2-4 over the integers.

```
> collect(x^2-4)
```

Output (in real mode):

$$(x-2)\,(x+2)$$

Factor x^2+4 over the integers.

```
> collect(x^2+4)
```

Output (in real mode):

$$x^2+4$$

Output (in complex mode):

$$(x+2\mathrm{i})\,(x-2\mathrm{i})$$

Factor x^2-2 over the rationals.

```
> collect(x^2-2)
```

Output (in real mode):

$$x^2-2$$

But if you input:

```
> collect(sqrt(2)*(x^2-2))
```

you get:

$$\sqrt{2}\left(x-\sqrt{2}\right)\left(x+\sqrt{2}\right)$$

Factor x^3-2x^2+1 and x^2-x over the rationals.

```
> collect([x^3-2*x^2+1,x^2-x])
```

$$\left[(x-1)\left(x^2-x-1\right),x\,(x-1)\right]$$

but:

```
> collect((x^3-2*x^2+1)*sqrt(5))
```

$$\sqrt{5}\left(x+\frac{-\sqrt{5}-1}{2}\right)(x-1)\left(x+\frac{\sqrt{5}-1}{2}\right)$$

or:

```
> collect(x^3-2*x^2+1,sqrt(5))
```

$$\left(x+\frac{-\sqrt{5}-1}{2}\right)(x-1)\left(x+\frac{\sqrt{5}-1}{2}\right)$$

11.1.19 Square-free factorization

The `sqrfree` command provides squarefree factorizations of polynomials; that is, it factors a polynomial as a product of powers of coprime factors, where each factor has roots of multiplicity 1 (in other words, a factor and its derivative are coprime).

- `sqrfree` takes P, a polynomial.
- `sqrfree(`P`)` returns the squarefree factorization of P.

Examples

```
> sqrfree((x^2-1)*(x-1)*(x+2))
```

$$\left(x^2+3x+2\right)(x-1)^2$$

```
> sqrfree((x^2-1)^2*(x-1)*(x+2)^2)
```

$$\left(x^2+3x+2\right)^2(x-1)^3$$

11.1.20 List of factors

The `factors` command provides the factors of a polynomial as a list.

- `factors` takes P, a polynomial or a list of polynomials.
- `factors(`P`)` returns a list containing the factors of P and their exponents, or a list of such lists.

Examples

```
> factors(x^2+2*x+1)
```

$$[x+1,2]$$

```
> factors(x^4-2*x^2+1)
```

$$[x-1,2,x+1,2]$$

```
> factors([x^3-2*x^2+1,x^2-x])
```

$$\begin{bmatrix} x-1 & 1 & x^2-x-1 & 1 \\ x & 1 & x-1 & 1 \end{bmatrix}$$

```
> factors([x^2,x^2-1])
```

$$[[x, 2], [x-1, 1, x+1, 1]]$$

11.1.21 Computing with the exact root of a polynomial

The `rootof` command finds the value of one polynomial at a root of another.

- `rootof` takes two arguments: P and Q, two polynomials given by the lists of their coefficients.
- `rootof(`P, Q`)` gives the value $P(\alpha)$ where α is the root of Q with largest real part (and largest imaginary part in case of equality).

In exact computations, Xcas will rewrite rational evaluations of `rootof` as a unique `rootof` with degree(P) < degree(Q). If the resulting rootof is the solution of a second degree equation, it will be simplified.

Example

Let α be the root with largest imaginary part of $Q(x) = x^4 + 10x^2 + 1$ (all roots of Q have real part equal to 0).

Compute $\frac{1}{\alpha}$.

```
> normal(1/rootof([1,0],[1,0,10,0,1]))
```

$$-\mathrm{i}\left(-\sqrt{2}+\sqrt{3}\right)$$

$P(x) = x$ is represented by $[1, 0]$ and α by `rootof`$([1, 0], [1, 0, 10, 0, 1])$.

Compute α^2.

```
> normal(rootof([1,0],[1,0,10,0,1])^2)
```

or (since $P(x) = x^2$ is represented by $[1, 0, 0]$):

```
> normal(rootof([1,0,0],[1,0,10,0,1]))
```

$$-2\sqrt{6}-5$$

11.1.22 Exact roots of a polynomial

The `roots` command finds roots of polynomials with their multiplicities.

- `roots` takes one mandatory and one optional argument:
  - P, a symbolic polynomial expression.
  - Optionally, x, the name of the variable (the default is `x`).
- `roots(`$P\,\langle, x\rangle$`)` returns a 2 column matrix: each row is the list consisting of a root of P and its multiplicity.

Examples

Find the roots of $P(x) = x^5 - 2x^4 + x^3$.

```
> roots(x^5-16*x^4+x^3)
```

$$\begin{bmatrix} 3\sqrt{7}+8 & 1 \\ -3\sqrt{7}+8 & 1 \\ 0 & 3 \end{bmatrix}$$

Find the roots of $x^{10} - 15x^8 + 90x^6 - 270x^4 + 405x^2 - 243 = (x^2 - 3)^5$.

```
> roots(x^10-15*x^8+90*x^6-270*x^4+405*x^2-243)
```

$$\begin{bmatrix} \sqrt{3} & 5 \\ -\sqrt{3} & 5 \end{bmatrix}$$

Find the roots of $t^3 - 1$.

```
> roots(t^3-1,t)
```

$$\begin{bmatrix} 1 & 1 \\ \frac{i\sqrt{3}-1}{2} & 1 \\ \frac{-i\sqrt{3}-1}{2} & 1 \end{bmatrix}$$

11.1.23 Coefficients of a polynomial defined by its roots

The `pcoeff` or `pcoef` command reconstructs a polynomial from its roots.

- `pcoeff` takes *roots*, a list of the roots of a polynomial P.
- `pcoeff(`*roots*`)` returns the monic polynomial having these roots, represented as the list of its coefficients in decreasing order.

Example

```
> pcoef([1,2,0,0,3])
```

$$[1, -6, 11, -6, 0, 0]$$

i.e. $(x-1)(x-2)(x^2)(x-3) = x^5 - 6x^4 + 11x^3 - 6x^2$.

11.1.24 Truncating to order n

The `truncate` command truncates a polynomial; i.e., it removes higher order terms.

- `truncate` takes two arguments:
  - P, a polynomial.
  - n, an integer.
- `truncate(`P, n`)` returns P truncated to order n; i.e., all terms of order greater or equal to $n+1$ are removed.

`truncate` may be used to transform a series expansion into a polynomial or to compute a series expansion step by step.

Examples

```
> truncate((1+x+x^2/2)^3,4)
```

$$\frac{9x^4 + 16x^3 + 18x^2 + 12x + 4}{4}$$

```
> truncate(series(sin(x)),4)
```

$$\frac{-x^3 + 6x}{6}$$

Note that the returned polynomial is normalized.

11.1.25 Converting a series expansion into a polynomial

The `convert` command (see Section 10.1.10, p. 195), with the option *polynom*, converts a series (see Section 13.5.2, p. 302) into a polynomial. It should be used for operations like drawing the graph of the Taylor series of a function near a point.

- `convert` takes two arguments:
  - *series*, a series.
  - `polynom`, the option.
- `convert(`*series*`,polynom)` returns *series* with the `order_size` term replaced by 0.

Examples

```
> convert(taylor(sin(x)),polynom)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

```
> convert(series(sin(x),x=0,6),polynom)
```

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

11.1.26 Changing the order of variables

The `reorder` command rewrites an expression, based on the priority of variables.

- `reorder` takes two arguments:
  - *expr*, an expression.
  - *vars*, a vector of variable names.
- `reorder(`*expr*, *vars*`)` expands *expr* according to the order of variables given in *vars*.

Example

```
> reorder(x^2+2*x*a+a^2+z^2-x*z,[a,x,z])
```

$$a^2 + 2ax + x^2 - xz + z^2$$

Remark. The variables must be symbolic before calling `reorder`. If they are not, purge them (see Section 3.3.8, p. 39).

11.1.27 Random polynomials

The `randpoly` or `randPoly` command generates random polynomials.

- `randpoly` takes two optional arguments:
  - Optionally x, the name of a variable (by default `x`).
  - Optionally n, an integer (by default 10).

  The order of the arguments is not important.
- `randpoly(`$\langle x, n \rangle$`)` returns a monic polynomial in the variable x of degree n, having as coefficients random integers evenly distributed on $-99, \ldots, +99$.

Examples

```
> randpoly(t,4)
```

$$t^4 + 86t^3 - 97t^2 - 82t + 7$$

```
> randpoly(4)
```

$$x^4 - 27x^3 + 26x^2 - 89x + 63$$

```
> randpoly(4,u)
```

$$u^4 - 49u^3 - 86u^2 - 64u - 30$$

11.1.28 Random lists

The `ranm` command finds lists of random integers.

- `ranm` takes n, an integer.
- `ranm(`n`)` returns a list of n random integers (between -99 and $+99$). This list can be seen as the coefficients of an univariate polynomial of degree $n-1$.

(See also Section 20.3.6, p. 532)

Example

```
> ranm(3)
```

$$[70, 22, 42]$$

11.2 Arithmetic and polynomials

Polynomials are represented by expressions or by lists of coefficients in decreasing power order. In the first case, for instructions requiring a main variable (like extended gcd computations), the variable used by default is `x` if not specified. For coefficients in $\mathbb{Z}/n\mathbb{Z}$, use `% n` for each coefficient of the list or apply it to the entire expression defining the polynomial.

11.2.1 Divisors of a polynomial

The `divis` command finds the divisors of a polynomial.

- `divis` takes P, a polynomial or a list of polynomials.
- `divis(`P`)` and returns the list of the divisors of P.

Examples

```
> divis(x^4-1)
```

$$\left[1, x-1, x+1, (x-1)(x+1), x^2+1, (x-1)\left(x^2+1\right), (x+1)\left(x^2+1\right), (x-1)(x+1)\left(x^2+1\right)\right]$$

```
> divis([x^2,x^2-1])
```

$$\left[\left[1, x, x^2\right], [1, x-1, x+1, (x-1)(x+1)]\right]$$

11.2.2 Euclidean quotient

The `quo` command finds the quotient of the Euclidean division of two polynomials.

- `quo` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `quo(`$P, Q\,\langle, x\rangle$`)` returns the Euclidean quotient of P divided by Q.

`Quo` is the inert form of `quo`; namely, it evaluates to `quo` for later evaluation. It is used when Xcas is in Maple mode (see Section 2.5.2, p. 14) to compute the euclidean quotient of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.

Examples

```
> quo(x^2+2*x+1,x)
```

$$x+2$$

```
> quo(y^2+2*y+1,y,y)
```

$$y+2$$

In list representation, to get the quotient of x^2+2x+4 by x^2+x+2 you can also input:

```
> quo([1,2,4],[1,1,2])
```

$$[1]$$

that is to say, the polynomial `1`.

Input in Xcas mode:

```
> Quo(x^2+2*x+1,x)
```

$$\text{quo}\left(x^2+2x+1, x\right)$$

Input in Maple mode:

```
> Quo(x^3+3*x,2*x^2+6*x+5) mod 5
```

$$-2x+1$$

This division was done using modular arithmetic, unlike with the following command, where the division is done in $\mathbb{Z}[X]$ and reduction afterwards:

```
> quo(x^3+3*x,2*x^2+6*x+5) mod 5
```

$$3x+6$$

If Xcas is not in Maple mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is done e.g. by:

```
> quo((x^3+3*x)%5,(2x^2+6x+5)%5)
```

$$(-2x+1) \bmod 5$$

11.2.3 Euclidean remainder

The `rem` command finds the remainder of the Euclidean division of two polynomials.

- `rem` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `rem`$(P, Q\,\langle, x\rangle)$ returns the Euclidean remainder of P divided by Q.

`Rem` is the inert form of `rem`; namely, it evaluates to `rem` for later evaluation. It is used when XCAS is in MAPLE mode (see Section 2.5.2, p. 14) to compute the euclidean remainder of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using MAPLE-like syntax.

Examples

```
> rem(x^3-1,x^2-1)
```

$$x - 1$$

To have the remainder of $x^2 + 2x + 4$ by $x^2 + x + 2$, you can also do:

```
> rem([1,2,4],[1,1,2])
```

$$[1, 2]$$

i.e. the polynomial $x + 2$.
Input in XCAS mode:

```
> Rem(x^3-1,x^2-1)
```

$$\text{rem}\left(x^3 - 1, x^2 - 1\right)$$

Input in MAPLE mode:

```
> Rem(x^3+3*x,2*x^2+6*x+5) mod 5
```

$$2x$$

This division was done using modular arithmetic, unlike with the following command, where the division is done in $\mathbb{Z}[X]$ and reduction afterwards:

```
> rem(x^3+3*x,2*x^2+6*x+5) mod 5
```

$$12x$$

If XCAS is not in MAPLE mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is entered like this:

```
> rem((x^3+3*x)%5,(2x^2+6x+5)%5)
```

$$x\,(2 \bmod 5)$$

11.2.4 Quotient and remainder

The `quorem` or `divide` command finds the quotient and remainder of the Euclidean division of two polynomials.

- `quorem` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `quorem`$(P, Q\,\langle, x\rangle)$ returns a list consisting of the Euclidean quotient and the Euclidean remainder of P divided by Q.

Examples

```
> quorem([1,2,4],[1,1,2])
```

$$[[1], [1, 2]]$$

```
> quorem(x^3-1,x^2-1,x)
```

$$[x, x-1]$$

11.2.5 GCD of two polynomials with the Euclidean algorithm

The `gcd` command computes the gcd (greatest common divisor) of polynomials. (See also Section 7.1.1, p. 104 for GCD of integers.)

- `gcd` takes an arbitrary number or arguments: *polys*, a sequence or list of polynomials.
- `gcd(`*polys*`)` returns the greatest common divisor of the polynomials in *polys*.

`Gcd` is the inert form of `gcd`; namely, it evaluates to `gcd` for later evaluation. It is used when XCAS is in MAPLE mode (see Section 2.5.2, p. 14) to compute the gcd of polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using MAPLE-like syntax.

Examples

```
> gcd(x^2+2*x+1,x^2-1)
```

$$x+1$$

```
> gcd(x^2-2*x+1,x^3-1,x^2-1,x^2+x-2)
```

or:

```
> gcd([x^2-2*x+1,x^3-1,x^2-1,x^2+x-2])
```

$$x-1$$

For polynomials with modular coefficients:

```
> gcd((x^2+2*x+1) mod 5,(x^2-1) mod 5)
```

$$(1 \,\%\, 5)\, x + 1 \,\%\, 5$$

Input in XCAS mode:

```
> Gcd(x^3-1,x^2-1)
```

$$\gcd\left(x^3-1, x^2-1\right)$$

Input in MAPLE mode:

```
> Gcd(x^2+2*x,x^2+6*x+5) mod 5
```

$$1$$

11.2.6 Choosing the GCD algorithm of two polynomials

The `ezgcd`, `heugcd`, `modgcd` and `psrgcd` commands compute the gcd (greatest common divisor) of two univariate or multivariate polynomials with coefficients in $\mathbb{Z}$ or $\mathbb{Z}[i]$ with different algorithms.

- `ezgcd`, `heugcd`, `modgcd` and `psrgcd` take two arguments: P and Q, two polynomials.
- `ezgcd`(P,Q) returns the gcd of P and Q computed with the ezgcd algorithm.
- `heugcd`(P,Q) returns the gcd of P and Q computed with the heuristic algorithm.
- `modgcd`(P,Q) returns the gcd P and Q computed with the modular algorithm.
- `psrgcd`(P,Q) returns the gcd of P and Q computed with the sub-resultant algorithm.

Examples

```
> ezgcd(x^2-2*x*y+y^2-1,x-y)
```
or:
```
> heugcd(x^2-2*x*y+y^2-1,x-y)
```
or:
```
> modgcd(x^2-2*x*y+y^2-1,x-y)
```
or:
```
> psrgcd(x^2-2*x*y+y^2-1,x-y)
```

$$1$$

```
> ezgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```
or:
```
> heugcd((x+y-1)*(x+y+1),(x+y+1)^2)
```
or:
```
> modgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```

$$x+y+1$$

```
> psrgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```

$$-x-y-1$$

```
> ezgcd((x+1)^4-y^4,(x+1-y)^2)
```

GCD not successful Error: Bad Argument Value

But:
```
> heugcd((x+1)^4-y^4,(x+1-y)^2)
```
or:
```
> modgcd((x+1)^4-y^4,(x+1-y)^2)
```
or:
```
> psrgcd((x+1)^4-y^4,(x+1-y)^2)
```

$$x-y+1$$

11.2.7 LCM of two polynomials

The `lcm` command computes the LCM (Least Common Multiple) of polynomials. (See 7.1.3 for LCM of integers).

- `lcm` takes *polys*, a sequence or list of polynomials.
- `lcm(`*polys*`)` returns the least common multiple of the polynomials in *polys*.

Examples

```
> lcm(x^2+2*x+1,x^2-1)
```

$$(x+1)\left(x^2-1\right)$$

```
> lcm(x,x^2+2*x+1,x^2-1)
```

or:

```
> lcm([x,x^2+2*x+1,x^2-1])
```

$$\left(x^2+x\right)\left(x^2-1\right)$$

11.2.8 Bézout's identity

Bézout's Identity (also known as Extended Greatest Common Divisor) states that for two polynomials $A(x), B(x)$ with greatest common divisor $D(x)$, there exist polynomials $U(x)$ and $V(x)$ such that

$$U(x)\,A(x) + V(x)\,B(x) = D(x).$$

The `egcd` or `gcdex` command computes the greatest common divisor of two polynomials as well as the polynomials $U(x)$ and $V(x)$ in the above identity.

- `egcd` takes two mandatory arguments and one optional argument:
  - A and B, polynomials given as expressions or lists of coefficients in decreasing order.
  - Optionally, if the polynomials are expressions, x, the variable (which defaults to `x`).
- `egcd(`$A, B\,\langle, x\rangle$`)` returns a list $[U, V, D]$, where D is the greatest common divisor of A and B, and U and V are the polynomials from Bézout's identity.

Examples

```
> egcd(x^2+2*x+1,x^2-1)
```

$$[1, -1, 2x+2]$$

```
> egcd([1,2,1],[1,0,-1])
```

$$[[1], [-1], [2, 2]]$$

```
> egcd(y^2-2*y+1,y^2-y+2,y)
```

$$[y-2, -y+3, 4]$$

```
> egcd([1,-2,1],[1,-1,2])
```

$$[[1, -2], [-1, 3], [4]]$$

11.2.9 Solving $au + bv = c$ over polynomials

A consequence of Bézout's identity is that given polynomials $A(x), B(x)$ and $C(x)$, there exist polynomials $U(x)$ and $V(x)$ such that

$$C(x) = U(x)A(x) + V(x)\,B(x)$$

exactly when $C(x)$ is a multiple of the greatest common divisor of $A(x)$ and $B(x)$. The `abcuv` command solves this polynomial equation.

- `abcuv` takes three mandatory and one optional argument:
  - A, B and C, three polynomials given as expressions or lists of coefficients in decreasing order, where C is a multiple of the greatest common divisor of A and B.
  - Optionally if the polynomials are expressions, x, the variable (which defaults to `x`).
- `abcuv(`$A, B, C\,\langle, x\rangle$`)` returns a list of two expressions $[U, V]$ such that $C = UA + VB$.

Examples

```
> abcuv(x^2+2*x+1,x^2-1,x+1)
```

$$\left[\frac{1}{2}, -\frac{1}{2}\right]$$

```
> abcuv(x^2+2*x+1,x^2-1,x^3+1)
```

$$\left[\frac{-x+2}{2}, \frac{3}{2}x\right]$$

```
> abcuv([1,2,1],[1,0,-1],[1,0,0,1])
```

$$[[\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}], [-\frac{1}{2}, \frac{1}{2}, -\frac{1}{2}]]$$

11.2.10 Chinese remainders

Theorem 4 (Chinese remainders)**.** *If $R(x)$ and $Q(x)$ are relatively prime polynomials, then for any polynomials $A(x)$ and $B(x)$, there exists a polynomial $P(x)$ such that:*

$$\begin{aligned} P(x) &\equiv A(x) \pmod{R(x)} \\ P(x) &\equiv B(x) \pmod{Q(x)} \end{aligned}$$

The `chinrem` command finds the polynomial P.

- `chinrem` takes two mandatory arguments and one optional argument:
  - $[A, R]$ and $[B, Q]$, two lists, each consisting of two polynomials given by expressions or lists of coefficients in decreasing order.
  - Optionally, if the polynomials are expressions, x, the main variable (by default `x`).
- `chinrem(`$[A, R], [B, Q]\,\langle, x\rangle$`)` returns the list $[P, S]$, where P and S are polynomials such that:

$$\begin{aligned} S &= RQ \\ P &\equiv A \pmod{R} \\ P &\equiv B \pmod{Q} \end{aligned}$$

If R and Q are coprime, a solution P always exists and all the solutions are congruent modulo $S = RQ$.

Examples

Find P such that

$$\begin{cases} P(x) \equiv x & (\text{mod } x^2+1), \\ P(x) \equiv x-1 & (\text{mod } x^2-1). \end{cases}$$

```
> chinrem([[1,0],[1,0,1]],[[1,-1],[1,0,-1]])
```

$$\left[\left[-\frac{1}{2}, 1, -\frac{1}{2}\right], [1,0,0,0,-1]\right]$$

or:

```
> chinrem([x,x^2+1],[x-1,x^2-1])
```

$$\left[-\frac{x^2}{2}+x-\frac{1}{2}, x^4-1\right]$$

hence $P(x) \equiv -\frac{x^2-2x+1}{2} \pmod{x^4-1}$.

```
> chinrem([[1,2],[1,0,1]],[[1,1],[1,1,1]])
```

$$[[-1,-1,0,1],[1,1,2,1,1]]$$

or:

```
> chinrem([y+2,y^2+1],[y+1,y^2+y+1],y)
```

$$\left[-y^3-y^2+1, y^4+y^3+2y^2+y+1\right]$$

11.2.11 Cyclotomic polynomial

For a positive integer n, *cyclotomic polynomial* of index n is the monic polynomial whose roots are exactly the primitive nth roots of unity (an nth root of unity is primitive if the set of its powers is the set of all the nth roots of unity). Note that this will divide x^n-1, whose roots are all the nth roots of unity.

The `cyclotomic` command computes cyclotomic polynomials.

- `cyclotomic` takes n, an integer.
- `cyclotomic(`n`)` returns the list of the coefficients of the cyclotomic polynomial of index n.

Examples

Let $n=4$; the fourth roots of unity are: $\{1, \mathrm{i}, -1, -\mathrm{i}\}$ and the primitive roots are: $\{\mathrm{i}, -\mathrm{i}\}$. Hence, the cyclotomic polynomial of index 4 is $(x-\mathrm{i})(x+\mathrm{i}) = x^2+1$.

Input for verification:

```
> cyclotomic(4)
```

$$[1,0,1]$$

Obtain cyclotomic poylnomial of index 5:

```
> cyclotomic(5)
```

$$[1,1,1,1,1]$$

Hence, the cyclotomic polynomial of index 5 is $x^4+x^3+x^2+x+1$, which divides x^5-1 since $(x-1)(x^4+x^3+x^2+x+1)=x^5-1$.

Obtain the cyclotomic polynomial of index 10:

```
> cyclotomic(10)
```

$$[1,-1,1,-1,1]$$

Hence, the cyclotomic polynomial of index 10 is $x^4-x^3+x^2-x+1$ and

$$(x^5-1)(x+1)(x^4-x^3+x^2-x+1)=x^{10}-1.$$

Obtain the cyclotomic polynomial of index 20:

```
> cyclotomic(20)
```

$$[1,0,-1,0,1,0,-1,0,1]$$

Hence, the cyclotomic polynomial of index 20 is $x^8-x^6+x^4-x^2+1$ and

$$(x^{10}-1)(x^2+1)(x^8-x^6+x^4-x^2+1)=x^{20}-1.$$

11.2.12 Sturm sequences and number of sign changes of P on $(a,b]$

Given a polynomial or rational expression $P(x)$, the Sturm sequence is the sequence $P_1(x),P_2(x),\ldots$ given by the recurrence relation:

- $P_1(x)$ is the opposite of the euclidean division remainder of $P(x)$ by $P'(x)$.
- $P_2(x)$ is the opposite of the euclidean division remainder of $P'(x)$ by $P_1(x)$.
- ...

If $P(x)$ is a polynomial of degree n, then this sequence has at most n terms.

If $P(x)$ is square-free, then Sturm's Theorem gives a way to use the sequence to determine the number of zeros of $P(x)$ on an interval.

The `sturm` command finds either the Sturm sequence (in which case it can also be called as `sturmseq`) or the number of zeros in an interval (in which case it can also be called as `sturmab`).

- To find the Sturm sequence, `sturm` (or `sturmseq`) takes one mandatory argument and one optional argument:
  - P, a polynomial or rational expression.
  - Optionally, x, a variable name (by default `x`).
- `sturm(`P $\langle,x\rangle$`)` (or `sturmseq(`P $\langle,x\rangle$`)`) returns the list of the Sturm sequences and multiplicities of the square-free factors of P.
- To compute the number of zeros in an interval, `sturm` (or `sturmab`) takes four arguments:
  - P, a polynomial expression.
  - x, a variable name.
  - a and b, two real or complex numbers.
- If a and b are reals, `sturm(`P,x,a,b`)` (or `sturmab(`P,x,a,b`)`) returns the number of sign changes of P on $(a,b]$; In other words, it returns the number of zeros in $[a,b)$ of the polynomial P/G where $G=\gcd(P,P')$.
- if a or b is complex, `sturm(`P,x,a,b`)` (or `sturmab(`P,x,a,b`)`) returns the number of complex roots of P in the rectangle having a and b as opposite vertices.

Examples

```
> sturm(2*x^3+2)
```

or:

```
> sturm(2*y^3+2,y)
```

$$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1]$$

The first term gives the content of the numerator (here 2), then the Sturm sequence (in list representation) is $[x^3 + 1, 3x^2, -9]$.

```
> sturm((2*x^3+2)/(3*x^2+2),x)
```

or:

```
> sturmseq((2*x^3+2)/(3*x^2+2),x)
```

$$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1, [[3, 0, 2], [6, 0], -72]]$$

The first term gives the content of the numerator (here 2), then the Sturm sequence of the numerator ([[1,0,0,1],[3,0,0],-9]), then the content of the denominator (here 1) and the Sturm sequence of the denominator ([[3,0,2],[6,0],-72]). As expressions, $[x^3+1, 3x^2, -9]$ is the Sturm sequence of the numerator and $[3x^2 + 2, 6x, -72]$ is the Sturm sequence of the denominator.

```
> sturm((x^3+1)^2,x)
```

or:

```
> sturmseq((x^3+1)^2,x)
```

$$[1, 1]$$

```
> sturm(3*(3*x^3+1)/(2*x+2),x)
```

$$[3, [[3, 0, 0, 1], [9, 0, 0], -81], 2, [[1, 1], 1]]$$

The first term gives the content of the numerator (here 3), the second term gives the Sturm sequence of the numerator (here $3x^3 + 1, 9x^2, -81$), the third term gives the content of the denominator (here 2), and the fourth term gives the Sturm sequence of the denominator (here $x + 1, 1$).

```
> sturm(2*x^3+2,x)
```

or:

```
> sturmseq(2*x^3+2,x)
```

$$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1]$$

```
> sturm((2*x^3+2)/(x+2),x)
```

or:

```
> sturmseq((2*x^3+2)/(x+2),x)
```

$$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1, [[1, 2], 1]]$$

Examples of computing the number of zeros in an interval:

```
> sturm(x^2*(x^3+2),x,-2,0)
```

or:

```
> sturmab(x^2*(x^3+2),x,-2,0)
```

$$1$$

```
> sturm(x^2*(x^3+2),x,-2,0)
```

or:

```
> sturmab(x^2*(x^3+2),x,-2,0)
```

1

```
> sturm(x^3-1,x,-2-i,5+3i)
```

or:

```
> sturmab(x^3-1,x,-2-i,5+3i)
```

3

```
> sturm(x^3-1,x,-i,5+3i)
```

or:

```
> sturmab(x^3-1,x,-i,5+3i)
```

1

Note that the polynomial must defined by its symbolic expression. For example, these commands produce an error:

```
> sturm([1,0,0,1],x)
```

or:

```
> sturm([1,0,0,2,0,0],x,-2,0)
```

Bad argument type

11.2.13 Sylvester matrix of two polynomials and resultant

Given two polynomials $A(x) = \sum_{i=0}^{i=n} a_i x^i$ and $B(x) = \sum_{i=0}^{i=m} b_i x^i$, their Sylvester matrix is a square matrix of size $m+n$ where $m = \text{degree}(B)$ and $n = \text{degree}(A)$. The m first lines are made with the coefficients of A, so that:

$$\begin{bmatrix} s_{11}=a_n & s_{12}=a_{n-1} & \cdots & s_{1(n+1)}=a_0 & 0 & \cdots & 0 \\ s_{21}=0 & s_{22}=a_n & \cdots & s_{2(n+1)}=a_1 & s_{2(n+2)}=a_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{m1}=0 & s_{m2}=0 & \cdots & s_{m(n+1)}=a_{m-1} & s_{m(n+2)}=a_{m-2} & \cdots & a_0 \end{bmatrix}$$

and the n further lines are made with the coefficients of B, so that:

$$\begin{bmatrix} s_{(m+1)1}=b_m & s_{(m+1)2}=b_{m-1} & \cdots & s_{(m+1)(m+1)}=b_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{(m+n)1}=0 & s_{(m+n)2}=0 & \cdots & s_{(m+n)(m+1)}=b_{n-1} & b_{n-2} & \cdots & b_0 \end{bmatrix}$$

The determinant of a Sylvester polynomial is the resultant of the two polynomials. If A and B have integer coefficients with non-zero resultant r, then the polynomials equation

$$AU + BV = r$$

has a unique solution U, V such that $\text{degree}(U) < \text{degree}(B)$ and $\text{degree}(V) < \text{degree}(A)$, and this solution has integer coefficients.

Remark. The discriminant of a polynomial is the resultant of the polynomial and its derivative. The `sylvester` command computes Sylvester matrices.

- `sylvester` takes two arguments: P and Q, two polynomials.
- `sylvester`(P, Q) returns the Sylvester matrix of P and Q.

The `resultant` command computes the resultant of two polynomials.

- `resultant` takes three arguments:
  - P and Q, two polynomials.
  - x, a variable.
- `resultant`(P, Q, x) returns the resultant of P and Q.

Example

```
> sylvester(x^3-p*x+q,3*x^2-p,x)
```

$$\begin{bmatrix} 1 & 0 & -p & q & 0 \\ 0 & 1 & 0 & -p & q \\ 3 & 0 & -p & 0 & 0 \\ 0 & 3 & 0 & -p & 0 \\ 0 & 0 & 3 & 0 & -p \end{bmatrix}$$

```
> det([[1,0,-p,q,0],[0,1,0,-p,q],[3,0,-p,0,0],[0,3,0,-p,0],[0,0,3,0,-p]])
```

$$-4p^3 + 27q^2$$

```
> resultant(x^3-p*x+q,3*x^2-p,x)
```

$$-4p^3 + 27q^2$$

Examples using the resultant.

1. Let F_1 and F_2 be two fixed points in the plane and A be a variable point on the circle with center F_1 and radius $2a$. Find the cartesian equation of the set of points M, intersection of the line $\overline{F_1A}$ and of the perpendicular bisector of $\overline{F_2A}$.

   Geometric solution. Since $|MF_1| + |MF_2| = |MF_1| + |MA| = |F_1A| = 2a$, the point M is on an ellipse with focus F_1, F_2 and major axis $2a$.

   Analytic solution. In the Cartesian coordinate system with center F_1 and x-axis having the same direction as the vector $\overrightarrow{F_1F_2}$, the coordinates of A are:

   $$A = (2a\cos(\theta), 2a\sin(\theta))$$

   where θ is the (Ox, OA) angle. Now choose $t = \tan(\theta/2)$ as parameter, so that the coordinates of A are rational functions with respect to t. More precisely:

   $$A = (ax, ay) = \left(2a\frac{1-t^2}{1+t^2}, 2a\frac{2t}{1+t^2}\right).$$

If $|F_1F_2| = 2c$ and if I is the midpoint of $\overline{AF_2}$, then since the coordinates of F_2 are $F_2 = (2c, 0)$, the coordinates of I are

$$I = (c + ax/2; ay/2) = \left(c + a\frac{1-t^2}{1+t^2}; a\frac{2t}{1+t^2}\right).$$

IM is orthogonal to $\overline{AF_2}$, hence $M = (x; y)$ satisfies the equation $\mathrm{eq}_1 = 0$ where

$$\mathrm{eq}_1 := (x - ix)(ax - 2c) + (y - iy)ay.$$

But $M = (x, y)$ is also on $\overline{F_1A}$, hence M satisfies the equation $\mathrm{eq}_2 = 0$ where

$$\mathrm{eq}_2 := y/x - ay/ax.$$

The resultant of both equations with respect to t, `resultant(eq1,eq2,t)`, is a polynomial eq_3 depending on the variables x, y, independent of t which is the cartesian equation of the set of points M when t varies.

To obtain the resultant:

```
> ax:=2*a*(1-t^2)/(1+t^2);ay:=2*a*2*t/(1+t^2);
  ix:=(ax+2*c)/2; iy:=(ay/2);
  eq1:=(x-ix)*(ax-2*c)+(y-iy)*ay;
  eq2:=y/x-ay/ax;
  factor(resultant(eq1,eq2,t))
```

$$-64(x^2 + y^2)(x^2a^2 - x^2c^2 + -2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4)$$

The factor $-64(x^2 + y^2)$ is always different from zero, hence the locus equation of M:

$$x^2a^2 - x^2c^2 + -2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4 = 0.$$

If the frame origin is O, the middle point of F_1F_2, then this is the cartesian equation of an ellipse. To make the change of origin $\overrightarrow{F_1M} = \overrightarrow{F_1O} + \overrightarrow{OM}$:

```
> normal(subst(x^2*a^2-x^2*c^2+-2*x*a^2*c+2*x*c^3-a^4+2*a^2*c^2+a^2*y^2-c^4,
               [x,y]=[c+X,Y]))
```

$$X^2a^2 - X^2c^2 + Y^2a^2 - a^4 + a^2c^2$$

or if $b^2 = a^2 - c^2$:

```
> normal(subst(-c^2*X^2+c^2*a^2+X^2*a^2-a^4+a^2*Y^2,c^2=a^2-b^2))
```

$$X^2b^2 + Y^2a^2 - a^2b^2$$

that is to say, after division by a^2b^2, M satisfies the equation

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} = 1.$$

2. Let F_1 and F_2 be fixed points and A a variable point on the circle with center F_1 and radius $2a$. Find the cartesian equation of the hull of D, the segment bisector of $\overline{F_2A}$.

   The segment bisector of $\overline{F_2A}$ is tangent to the ellipse of focus F_1, F_2 and major axis $2a$.

   In the Cartesian coordinate system with center F_1 and x-axis having the same direction as the vector F_1F_2, the coordinates of A are:

$$A = (2a\cos(\theta); 2a\sin(\theta)),$$

where θ is the (Ox, OA) angle. Choose $t = \tan(\theta/2)$ as parameter such that the coordinates of A are rational functions with respect to t. More precisely:

$$A = (ax; ay) = \left(2a\frac{1-t^2}{1+t^2}; 2a\frac{2t}{1+t^2}\right).$$

If $|F_1F_2| = 2c$ and I is the midpoint of AF_2:

$$F_2 = (2c, 0), \quad I = (c + ax/2; ay/2) = \left(c + a\frac{1-t^2}{1+t^2}; a\frac{2t}{1+t^2}\right).$$

Since D is orthogonal to $\overline{AF_2}$, the equation of D is $\mathrm{eq}_1 = 0$ where

$$\mathrm{eq}_1 := (x - ix)(ax - 2c) + (y - iy)ay.$$

So, the hull of D is the locus of M, the intersection point of D and D' where D' has equation $\mathrm{eq}_2 := \frac{\mathrm{d}}{\mathrm{d}t}\mathrm{eq}_1 = 0$.

To obtain the resultant:

```
> ax:=2*a*(1-t^2)/(1+t^2);ay:=2*a*2*t/(1+t^2);
  ix:=(ax+2*c)/2; iy:=(ay/2);
  eq1:=normal((x-ix)*(ax-2*c)+(y-iy)*ay);
  eq2:=normal(diff(eq1,t));
  factor(resultant(eq1,eq2,t))
```

$$-64a^2(x^2+y^2)(x^2a^2 - x^2c^2 + -2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4)$$

The factor $-64a^2(x^2 + y^2)$ is always different from zero, therefore the locus equation is:

$$x^2a^2 - x^2c^2 - 2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4 = 0$$

If O, the midpoint of $\overline{F_1F_2}$, is chosen as origin, you find again the cartesian equation of the ellipse:

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} = 1.$$

11.3 Exact bounds for roots of a polynomial

11.3.1 Exact bounds for real roots of a polynomial

Bounds for the real roots of a polynomial can be found by `realroot` and `VAS` commands.

- `realroot` takes two mandatory arguments and two optional arguments:
  - P, a polynomial.
  - ε, a postive real number.
  - Optionally, a, b, two complex numbers.
- `realroot(`P, ε`)` returns a list of vectors, where the elements of each vector are a list containing one of:
  - an interval of length less than ε containing a real root of the polynomial and the multiplicity of this root.
  - the value of an exact real root of the polynomial and the multiplicity of this root.

- `realroot(`P, ε, a, b`)` returns a list of vectors as above, but only for the roots lying in the interval $[a, b]$.

The `VAS` command uses the Vincent-Akritas-Strzebonski algorithm to find intervals containing the real roots of polynomials.

- `VAS` takes P, a polynomial.
- `VAS(`P`)` returns a list of intervals which contain the real roots of P, where each interval contains exactly one root.

Examples

Find the real roots of $x^3 + 1$.

```
> realroot(x^3+1,0.1)
```

$$\begin{bmatrix} -1 & 1 \end{bmatrix}$$

Find the real roots of $x^3 - x^2 - 2x + 2$.

```
> realroot(x^3-x^2-2*x+2,0.1)
```

$$\begin{bmatrix} -[1.40624999999999..1.50000000000001] & 1 \\ 1 & 1 \\ [1.37499999999999..1.43750000000001] & 1 \end{bmatrix}$$

Find the real roots of $x^3 - x^2 - 2x + 2$ in the interval $[0; 2]$.

```
> realroot(x^3-x^2-2*x+2,0.1,0,2)
```

$$\begin{bmatrix} 1 & 1 \\ [1.37499999999999..1.43750000000001] & 1 \end{bmatrix}$$

```
> VAS(x^3-7*x+7)
```

$$\begin{bmatrix} -4 & 0 \\ 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{bmatrix}$$

```
> VAS(x^5+2*x^4-6*x^3-7*x^2+7*x+7)
```

$$\left[[-5, -1], -1, \left[1, \frac{3}{2}\right], \left[\frac{3}{2}, 2\right] \right]$$

```
> VAS(x^3-x^2-2*x+2)
```

$$[[-3, 0], 1, [1, 3]]$$

11.3.2 Exact bounds for positive real roots of a polynomial

The `VAS_positive` command uses the Vincent-Akritas-Strzebonski algorithm to find intervals containing the positive real roots of polynomials.

- `VAS_positive` takes P, a polynomial.
- `VAS_positive(`P`)` returns a list of intervals which contain the positive real roots of P, where each interval contains exactly one root.

Examples

```
> VAS_positive(x^3-7*x+7)
```

$$\begin{bmatrix} 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{bmatrix}$$

```
> VAS_positive(x^5+2*x^4-6*x^3-7*x^2+7*x+7)
```

$$\begin{bmatrix} 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{bmatrix}$$

```
> VAS_positive(x^3-x^2-2*x+2)
```

$$[1, [1, 3]]$$

11.3.3 An upper bound for the positive real roots of a polynomial

The `posubLMQ` command uses the Local Max Quadratic (LMQ) Akritas-Strzebonski-Vigklas algorithm to find upper bounds for the positive real roots of polynomials.

- `posubLMQ` takes P, a polynomial.
- `posubLMQ(`P`)` returns a (non-optimal) upper bound for the positive real roots of P.

Examples

```
> posubLMQ(x^3-7*x+7)
```

$$4$$

```
> posubLMQ(x^5+2*x^4-6*x^3-7*x^2+7*x+7)
```

$$4$$

```
> posubLMQ(x^3-x^2-2*x+2)
```

$$3$$

11.3.4 A lower bound for the positive real roots of a polynomial

The `poslbdLMQ` command uses the Local Max Quadratic (LMQ) Akritas-Strzebonski-Vigklas algorithm to find lower bounds for the positive real roots of polynomials.

- `poslbdLMQ` takes P, a polynomial.
- `poslbdLMQ(`P`)` returns a (non-optimal) lower bound for the positive real roots of P.

Examples

```
> poslbdLMQ(x^3-7*x+7)
```

$$\frac{1}{2}$$

```
> poslbdLMQ(x^5+2*x^4-6*x^3-7*x^2+7*x+7)
```

$$\frac{1}{2}$$

```
> poslbdLMQ(x^3-x^2-2*x+2)
```

$$\frac{1}{2}$$

11.3.5 Exact values of rational roots of a polynomial

The `rationalroot` command finds rational roots of polynomials.

- `rationalroot` takes one mandatory and two optional arguments:
  - P, a polynomial.
  - Optionally, α and β, two real numbers.
- `rationalroot(`P`)` returns the list of the value of the rational roots of P without multiplicity.
- `rationalroot(`P, α, β`)` returns the list of the rational roots of P which are in the interval $[\alpha, \beta]$.

Examples

Find the rational roots of $2x^3 - 3x^2 - 8x + 12$:

```
> rationalroot(2*x^3-3*x^2-8*x+12)
```

$$\left[2, -2, \frac{3}{2}\right]$$

Find the rational roots of $2x^3 - 3x^2 - 8x + 12$ in $[1, 2]$:

```
> rationalroot(2*x^3-3*x^2-8*x+12,1,2)
```

$$\left[2, \frac{3}{2}\right]$$

Find the rational roots of $2x^3 - 3x^2 + 8x - 12$:

```
> rationalroot(2*x^3-3*x^2+8*x-12)
```

$$\left[\frac{3}{2}\right]$$

Find the rational roots of $2x^3 - 3x^2 + 8x - 12$:

```
> rationalroot(2*x^3-3*x^2+8*x-12)
```

$$\left[\frac{3}{2}\right]$$

Find the rational roots of $(3x - 2)^2(2x + 1) = 18x^3 - 15x^2 - 4x + 4$:

```
> rationalroot(18*x^3-15*x^2-4*x+4)
```

$$\left[-\frac{1}{2}, \frac{2}{3}\right]$$

11.3.6 Exact bounds for complex roots of a polynomial

The `complexroot` command finds bounds for the complex roots of a polynomial.

- `complexroot` takes two mandatory arguments and two optional arguments:
  - P, a polynomial.
  - ε, a postive real number.
  - Optionally, α, β, two complex numbers.
- `complexroot(`P, ε`)` returns a list of vectors, where the elements of each vector are one of:
  - an interval (the boundaries of this interval are the opposite vertices of a rectangle with sides parallel to the axis and containing a complex root of the polynomial) and the multiplicity of this root.

    Suppose the interval is $[a_1 + \mathrm{i}b_1, a_2 + \mathrm{i}b_2]$ then $|a_1 - a_2| < \varepsilon$, $|b_1 - b_2| < \varepsilon$ and the root $a + \mathrm{i}b$ satisfies $a_1 \leq a \leq a_2$ and $b_1 \leq b \leq b_2$.
  - the value of an exact complex root of the polynomial and the multiplicity of this root.
- `complexroot(`$P, \varepsilon, \alpha, \beta$`)` returns a list of vectors as above, but only for the roots lying in the rectangle with sides parallel to the axis having α, β as opposite vertices.

Examples

Find the roots of $x^3 + 1$.

```
> complexroot(x^3+1,0.1)
```

$$\begin{bmatrix} -1 & 1 \\ [0.499999046325680..0.500000953674320] - [0.866024494171135..0.866026401519779]\,\mathrm{i} & 1 \\ [0.499999046325680..0.500000953674320] + [0.866024494171135..0.866026401519779]\,\mathrm{i} & 1 \end{bmatrix}$$

Hence, for x^3+1, -1 is a root of multiplicity 1, $a+ib$ is a root of multiplicity 1 with $0.499999046325680 \leq a \leq 0.500000953674320$ and $-0.866026401519779 \leq b \leq -0.866024494171135$, and $c + \mathrm{i}d$ is a root of multiplicity 1 with $0.499999046325680 \leq c \leq 0.500000953674320$ and $0.866024494171135 \leq d \leq 0.866026401519779$.

Find the roots of $x^3 + 1$ lying inside the rectangle with opposite vertices $-1, 1 + 2\mathrm{i}$.

```
> complexroot(x^3+1,0.1,-1,1+2*i)
```

$$\begin{bmatrix} -1 & 1 \\ [0.499999046325680..0.500000953674320] + [0.866024494171135..0.866026401519779]\,\mathrm{i} & 1 \end{bmatrix}$$

11.3.7 Exact values of the complex rational roots of a polynomial

The `crationalroot` command finds complex rational roots of polynomials.

- `crationalroot` takes one mandatory and two optional arguments:
  - P, a polynomial.
  - Optionally, α and β, two complex numbers.
- `crationalroot(`P`)` returns the list of the value of the rational roots of P without multiplicity.
- `crationalroot(`P, α, β`)` returns the list of the rational roots of P which are in the rectangle with sides parallel to the axis having $[\alpha, \beta]$ as opposite vertices.

Example

Find the rational complex roots of $(x^2+4)(2x-3) = 2x^3-3x^2+8x-12$:

```
> crationalroot(2*x^3-3*x^2+8*x-12)
```

$$\left[2\mathrm{i}, \frac{3}{2}, -2\mathrm{i}\right]$$

11.4 Orthogonal polynomials

11.4.1 Legendre polynomials

The Legendre polynomial $L(n,x)$ of degree n is a polynomial solution of the differential equation

$$(x^2-1)\,y'' - 2\,x\,y' - n(n+1)\,y = 0.$$

The Legendre polynomials satisfy the recurrence relation:

$$\begin{aligned} L(0,x) &= 1 \\ L(1,x) &= x \\ L(n,x) &= \frac{2n-1}{n} x\,L(n-1,x) - \frac{n-1}{n} L(n-2,x) \end{aligned}$$

These polynomials are orthogonal for the scalar product:

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\,\mathrm{d}x.$$

The `legendre` command finds the Legendre polynomials.

- `legendre` takes one mandatory argument and one optional argument:
  - n, an integer.
  - Optionally, x, a variable name (by default `x`).
- `legendre(`$n\,\langle, x\rangle$`)` returns the Legendre polynomial of degree n.

Examples

```
> legendre(4)
```

$$\frac{35}{8}x^4 - \frac{15}{4}x^2 + \frac{3}{8}$$

```
> legendre(4,y)
```

$$\frac{35}{8}y^4 - \frac{15}{4}y^2 + \frac{3}{8}$$

11.4.2 Hermite polynomial

The Hermite polynomials $H(n,x)$ satisfy the recurrence relation:

$$\begin{aligned} H(0,x) &= 1 \\ H(1,x) &= 2x \\ H(n,x) &= 2xH(n-1,x) - 2(n-1)H(n-2,x) \end{aligned}$$

These polynomials are orthogonal for the scalar product:

$$\langle f,g\rangle = \int_{-\infty}^{+\infty} f(x)g(x)\mathrm{e}^{-x^2}\,\mathrm{d}x.$$

The `hermite` command finds the Hermite polynomials.

- `hermite` takes one mandatory argument and one optional argument:
  - n, an integer.
  - Optionally, x, a variable name (by default `x`).
- `hermite(`$n\,\langle,x\rangle$`)` returns the Hermite polynomial of degree n.

Examples

```
> hermite(6)
```

$$64x^6 - 480x^4 + 720x^2 - 120$$

```
> hermite(6,y)
```

$$64y^6 - 480y^4 + 720y^2 - 120$$

11.4.3 Laguerre polynomials

The Laguerre polynomial of degree n and parameter a satisfy the following recurrence relation:

$$\begin{aligned} L(0,a,x) &= 1 \\ L(1,a,x) &= 1 + a - x \\ L(n,a,x) &= \frac{2n+a-1-x}{n}L(n-1,a,x) - \frac{n+a-1}{n}L(n-2,a,x) \end{aligned}$$

These polynomials are orthogonal for the scalar product

$$\langle f,g\rangle = \int_{0}^{+\infty} f(x)g(x)x^a\mathrm{e}^{-x}\,\mathrm{d}x.$$

The `laguerre` command finds the Laguerre polynomials.

- `laguerre` takes one mandatory argument and two optional arguments:
  - n, an integer.
  - Optionally, x, a variable name (by default `x`).
  - Optionally, a, a parameter name (by default `a`).
- `laguerre(`$n\,\langle,x,a\rangle$`)` returns the Laguerre polynomial of degree n and parameter a.

Examples

```
> laguerre(2)
```

$$\frac{1}{2}a^2 - ax + \frac{3}{2}a + \frac{1}{2}x^2 - 2x + 1$$

```
> laguerre(2,y)
```

$$\frac{1}{2}a^2 - ay + \frac{3}{2}a + \frac{1}{2}y^2 - 2y + 1$$

```
> laguerre(2,y,b)
```

$$\frac{1}{2}b^2 - by + \frac{3}{2}b + \frac{1}{2}y^2 - 2y + 1$$

11.4.4 Chebyshev polynomials of the first kind

The Chebyshev polynomial of first kind $T(n, x)$ is defined by

$$T(n, x) = \cos(n \arccos x)$$

and satisfy the recurrence relation:

$$T(0, x) = 1, \quad T(1, x) = x, \quad T(n, x) = 2xT(n-1, x) - T(n-2, x).$$

The polynomials $T(n, x)$ are orthogonal for the scalar product

$$\langle f, g\rangle = \int_{-1}^{1} \frac{f(x)g(x)}{\sqrt{1-x^2}}\,\mathrm{d}x.$$

The `tchebyshev1` command finds the Chebyshev polynomials of the first kind.

- `tchebyshev1` takes one mandatory argument and one optional argument:
  - n, an integer.
  - Optionally x, a variable name (by default `x`).
- `tchebyshev1(`$n\,\langle, x\rangle$`)` returns the Chebyshev polynomial of first kind of degree n.

Examples

```
> tchebyshev1(4)
```

$$8x^4 - 8x^2 + 1$$

```
> tchebyshev1(4,y)
```

$$8y^4 - 8y^2 + 1$$

Indeed, $\cos(4x) = \mathrm{Re}((\cos x + \mathrm{i}\,\sin x)^4) = \cos^4 x - 6\cos^2 x(1-\cos^2 x) + ((1-\cos^2 x))^2 = T(4, \cos(x))$.

11.4.5 Chebyshev polynomial of the second kind

The Chebyshev polynomial of second kind $U(n, x)$ is defined by:

$$U(n, x) = \frac{\sin((n+1)\arccos x)}{\sin(\arccos x)}$$

or equivalently:

$$\sin((n+1)x) = U(n, \cos x)\sin x.$$

These satisfy the recurrence relation:

$$\begin{aligned} U(0, x) &= 1 \\ U(1, x) &= 2x \\ U(n, x) &= 2xU(n-1, x) - U(n-2, x) \end{aligned}$$

The polynomials $U(n, x)$ are orthogonal for the scalar product

$$\langle f, g\rangle = \int_{-1}^{1} f(x)g(x)\sqrt{1-x^2}\,\mathrm{d}x.$$

The `tchebyshev2` command finds the Chebyshev polynomials of the first kind.

- `tchebyshev2` takes one mandatory argument and one optional argument:
  - n, an integer.
  - Optionally x, a variable name (by default `x`).
- `tchebyshev2(`$n\,\langle, x\rangle$`)` returns the Chebyshev polynomial of second kind of degree n.

Examples

```
> tchebyshev2(3)
```

$$8x^3 - 4x$$

```
> tchebyshev2(3,y)
```

$$8y^3 - 4y$$

Indeed, $\sin(4x) = \sin(x)\,(8\,\cos(x)^3 - 4\cos(x)) = \sin(x)\,U(3, \cos(x))$.

11.5 Gröbner basis and Gröbner reduction

11.5.1 Gröbner basis

A set of polynomials $\{F_1, \ldots, F_N\}$ generate an *ideal* I; namely, I is the set of all linear combinations of the F_j. Given such an ideal, a Gröbner basis for I is a subset $G = \{G_1, \ldots, G_n\}$ of I such that for any F in I, there is a G_k in G such that the leading monomial of G_k divides the leading monomial of F. (Note that the leading monomial depends on a fixed ordering of the monomials.)

If G is a Gröbner basis for such an ideal I, then for any nonzero F in I, if you do a Euclidean division of F by the corresponding G_k, take the remainder of this division, do again the same and so on, at some point you get a remainder of zero.

Example

Let I be the ideal generated by $\{x^3 - 2xy, x^2y - 2y^2 + x\}$ with the standard lexicographic order on the monomials. One Gröbner basis for I is

$$G = \{g_1(x, y) = x^2, g_2(x, y) = xy, g_3(x, y) = 2y^2 - x\}.$$

Consider the element $F(x, y) = 2x^2y - 3x^2 + 6xy - 4y^2 + 2x$ of I. The leading monomial x^2y of $F(x, y)$ is divisible by the leading monomial x^2 of $g_1(x, y)$. Dividing $F(x, y)$ by $g_1(x, y)$ leaves a remainder of $R_1(x, y) = 6xy - 4y^2 + 2x$. The leading monomial of $R_1(x, y)$, which is xy, is divisible by the leading monomial of $g_2(x, y)$, which is xy. Dividing $R_1(x, y)$ by $g_2(x, y)$ leaves a remainder of $R_2(x, y) = -4y^2 + 2x$. Finally, the leading monomial of $R_2(x, y)$, which is y^2, is divisible by the leading monomial of $g_3(x, y)$, which is y^2. Dividing $R_2(x, y)$ by $g_3(x, y)$ leaves a remainder of 0.

The `gbasis` command computes Gröbner bases.

- `gbasis` takes two mandatory arguments and three optional arguments:
  - *polys*, a list of polynomials.
  - *vars*, a list of the variable names.
  - Optionally, *order*, which can be one of:
    - `plex`, to order the monomials lexicographically (this is the default).
    - `tdeg`, to order the monomials first by total degree then by lexicographic order.
    - `revlex`, to order the monomials reverse lexicographically.
  - Optionally, `with_cocoa=`*boolean*, where *boolean* can be `true` or `false`. A value of `true` means to use the CoCoA library to compute the Gröbner basis, a value of `false` means not to use it.

    A value of `true` is recommended, but requires that CoCoA support be compiled into Xcas.
  - Optionally, `with_f5=`*boolean*, where *boolean* can be `true` or `false`. A value of `true` means to use the F5 algorithm of the CoCoA library, a value of `false` means not to use it. If this is `true`, then the polynomials are homogenized and so the specified order is not used.
- `gbasis(`*polys*, *vars* ⟨, *order*, `with_cocoa=`*boolean*, `with_f5=`*boolean*⟩`)` returns a Gröbner basis of the ideal spanned by polynomials in *polys*.

Note that the lexicographic order depends on the order the variables are given in *vars*. For example, if *vars*`=[x,y,z]`, then `x^2*y^4*z^3` comes before `x^2*y^3*z^4`, but if *vars*`=[x,z,y]`, then `x^2*y^4*z^3` comes after `x^2*y^3*z^4`.

Examples

```
> gbasis([2*x*y-y^2,x^2-2*x*y],[x,y])
```

$$\left[y^3, x^2 - y^2, 2xy - y^2\right]$$

```
> gbasis([x1+x2+x3,x1*x2+x1*x3+x2*x3,x1*x2*x3-1],[x1,x2,x3],tdeg,with_cocoa=false)
```

$$\left[x_3^3 - 1, -x_2^2 - x_2x_3 - x_3^2, x_1 + x_2 + x_3\right]$$

11.5.2 Gröbner reduction

The `greduce` command finds a polynomial modulo I, where I is an ideal as in Section 11.5.1, p. 245.

- `greduce` takes three arguments mandatory arguments and three optional arguments:
  - P, a multivariate polynomial.
  - *gbasis*, a vector made of polynomials which is supposed to be a Gröbner basis.
  - *vars*, and a vector of variable names.
  - Optionally, the same ordering options and CoCoA options as `gbasis` (see Section 11.5.1, p. 245).
- `greduce(`P, *gbasis*, *vars* ⟨, *options*⟩`)` returns the reduction of P with respect to the Gröbner basis *gbasis*. It is 0 if and only if the polynomial belongs to the ideal.

Examples

```
> greduce(x*y-1,[x^2-y^2,2*x*y-y^2,y^3],[x,y])
```

$$\frac{1}{2}y^2 - 1$$

that is to say $xy - 1 \equiv \frac{1}{2}y^2 - 1 \pmod{I}$ where I is the ideal generated by the Gröbner basis $[x^2 - y^2, 2xy - y^2, y^3]$, because $\frac{1}{2}y^2 - 1$ is the Euclidean division remainder of $xy - 1$ by $G_2 = 2xy - y^2$.

```
> greduce(x1^2*x3^2,[x3^3-1,-x2^2-x2*x3-x3^2,x1+x2+x3],[x1,x2,x3],tdeg)
```

$$x_2$$

11.5.3 Testing if a (list of) polynomial(s) belongs to an ideal given by a Gröbner basis

The `in_ideal` command determines whether or not a polynomial is in an ideal.

- `in_ideal` takes three mandatory arguments and one optional argument:
  - P, a polynomial or a list of polynomials.
  - *gbasis*, a list giving a Gröbner basis.
  - *vars*, the list of polynomial variables.

    If *gbasis* is computed with a different order from the default, then *vars* must use the same order.
  - Optionally, an optional argument from `gbasic` (see Section 11.5.1, p. 245), such as `plex` or `tdeg`. By default it will be `plex`.
- `in_ideal(`P, *gbasis*, *vars* ⟨, *option*⟩`)` returns the value `true` (1) or `false` (0), or a list of `true`s and `false`s, indicating whether or not the polynomial(s) in P are in the ideal generated by *gbasis* using the variables in *vars*.

Examples

```
> in_ideal((x+y)^2,[y^2,x^2+2*x*y],[x,y])
```

$$1$$

```
> in_ideal([(x+y)^2,x+y],[y^2,x^2+2*x*y],[x,y])
```

$$[1, 0]$$

```
> in_ideal(x+y,[y^2,x^2+2*x*y],[x,y])
```

$$0$$

11.5.4 Building a polynomial from its evaluation

The `genpoly` command finds a polynomial which evaluates to a given polynomial.

- `genpoly` takes three arguments:
  - P, a polynomial with $n-1$ variables.
  - b, an integer.
  - x, the name of a variable.
- `genpoly`(P, b, x) returns the polynomial Q with n variables (the $n-1$ variables in P and the variable x) such that the coefficients of Q are in the interval $(-b/2, b/2]$ and $Q|_{x=b} = P$. In other words, P is written in base b but using the convention that the Euclidean remainder belongs to $(-b/2, b/2]$ (this convention is also known as s-mod representation).

Examples

```
> genpoly(61,6,x)
```

$$2x^2 - 2x + 1$$

Indeed 61 divided by 6 is 10 with remainder 1, then 10 divided by 6 is 2 with remainder -2 (instead of the usual quotient 1 and remainder 4 out of bounds), $61 = 2 \cdot 6^2 - 2 \cdot 6 + 1$.

```
> genpoly(5,6,x)
```

$$x - 1$$

Indeed, $5 = 6 - 1$.

```
> genpoly(7,6,x)
```

$$x + 1$$

Indeed, $7 = 6 + 1$.

```
> genpoly(7*y+5,6,x)
```

$$xy + x + y - 1$$

Indeed, $xy + x + y - 1 = y(x+1) + (x-1)$.

```
> genpoly(7*y+5*z^2,6,x)
```

$$xy + xz^2 + y - z^2$$

Indeed, $xy + xz + y - z = y(x+1) + z(x-1)$.

11.6 Rational functions

11.6.1 Numerator

The `getNum` command finds the numerator of an unreduced rational function.

- `getNum` takes *rat*, a rational function.
- `getNum(`*rat*`)` returns the numerator of *rat*.

Unlike `numer` (see Section 11.6.2, p. 249), textttgetNum does not simplify the expression before extracting the numerator.

Examples

```
> getNum((x^2-1)/(x-1))
```

$$x^2 - 1$$

```
> getNum((x^2+2*x+1)/(x^2-1))
```

$$x^2 + 2x + 1$$

11.6.2 Numerator after simplification

The `numer` command finds the numerator of a rational function, after it has been reduced. (See also Section 7.2.3, p. 123.)

- `numer` takes *rat*, a rational function.
- `numer(`*rat*`)` returns the numerator of the irreducible representation of *rat*.

Examples

```
> numer((x^2-1)/(x-1))
```

$$x + 1$$

```
> numer((x^2+2*x+1)/(x^2-1))
```

$$x + 1$$

11.6.3 Denominator

The `getDenom` command finds the denominator of an unreduced rational function.

- `getDenom` takes *rat*, a rational function.
- `getDenom(`*rat*`)` returns the denominator of *rat*.

Unlike `denom` (see Section 11.6.4, p. 250), textttgetDenom does not simplify the expression before extracting the denominator.

Examples

```
> getDenom((x^2-1)/(x-1))
```

$$x - 1$$

```
> getDenom((x^2+2*x+1)/(x^2-1))
```

$$x^2 - 1$$

11.6.4 Denominator after simplification

The `denom` command finds the denominator of a rational function after it has been reduced. (See also Section 7.2.4, p. 123.)

- `denom` takes *rat*, a rational function.
- `denom(`*rat*`)` returns the denominator of the irreducible representation of *rat*.

Examples

```
> denom((x^2-1)/(x-1))
```

$$1$$

```
> denom((x^2+2*x+1)/(x^2-1))
```

$$x - 1$$

11.6.5 Numerator and denominator

The `f2nd` or `fxnd` command finds the numerator and denominator of rational function after simplification.

- `f2nd` takes *rat*, a rational function.
- `f2nd(`*rat*`)` returns the list of the numerator and the denominator of the irreducible representation of *rat*.

Examples

```
> f2nd((x^2-1)/(x-1))
```

$$[x + 1, 1]$$

```
> f2nd((x^2+2*x+1)/(x^2-1))
```

$$[x + 1, x - 1]$$

11.6.6 Simplifying

The `simp2` command removes common factors from a pair of polynomials, as if reducing the numerator and denominator of a rational function. (See also Section 7.2.6, p. 124.)

- `simp2` takes two arguments: P and Q, two polynomials (or two integers, see Section 7.2.6, p. 124).
- `simp2(`P, Q`)` returns a list of two polynomials seen as the numerator and denominator of the irreducible representation of the rational function P/Q.

Example

```
> simp2(x^3-1,x^2-1)
```

$$\left[x^2+x+1, x+1\right]$$

11.6.7 Common denominator

The `comDenom` command finds the common denominator of a sum of rational functions and adds them.

- `comDenom` takes *sum*, a sum of rational functions.
- `comDenom(`*sum*`)` returns *sum* with the terms combined over a common denominator.

Example

```
> comDenom(x-1/(x-1)-1/(x^2-1))
```

$$\frac{x^3-2x-2}{x^2-1}$$

11.6.8 Polynomial and fractional part

The `propfrac` command rewrites a rational function as a polynomial plus a rational function whose numerator has smaller degree than the numberator; namely, it writes $\frac{A(x)}{B(x)}$ (after reduction), as:

$$Q(x)+\frac{R(x)}{B(x)} \quad \text{where } R(x)=0 \text{ or } 0 \le \text{degree}(R(x)) < \text{degree}(B(x)).$$

(See also Section 7.2.2, p. 122.)

- `propfrac` takes *rat*, a rational function.
- `propfrac(`*rat*`)` returns the sum of a polynomial and rational function which add to *rat*, and with the degree of the numerator of the rational function less that the degree of the denominator.

Example

```
> propfrac((5*x+3)*(x-1)/(x+2))
```

$$5x-12+\frac{21}{x+2}$$

11.6.9 Partial fraction expansion

The `partfrac` and `cpartfrac` commands find the partial fraction expansion of a rational function.

- `partfrac` takes *rat*, a rational function.
- `partfrac(`*rat*`)` returns the partial fraction expansion of *rat*.

  The `partfrac` command is equivalent to the `convert` command (see Section 10.1.10, p. 195) with `parfrac` (or `partfrac` or `fullparfrac`) as option.
- `cpartfrac(`*rat*`)` behaves just like `partfrac`, except that it always finds the partial fraction expansion over $\mathbb{C}$.

Example

Find the partial fraction expansion of $\frac{x^5-2x^3+1}{x^4-2x^3+2x^2-2x+1}$ over the real numbers.

Input in real mode:

```
> partfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

$$x+2-\frac{1}{2\,(x-1)}+\frac{x-3}{2\,(x^2+1)}$$

To find the partial fraction decomposition over the complex numbers, you can either put XCAS in complex mode (see Section 2.5.5, p. 14) or use `cpartfrac`.

Input in complex mode:

```
> partfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

or, in real or complex mode:

```
> cpartfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

$$x+2-\frac{1}{2\,(x-1)}+\frac{-1-2\mathrm{i}}{(2-2\mathrm{i})\,(x+\mathrm{i})}+\frac{2+\mathrm{i}}{(2-2\mathrm{i})\,(x-\mathrm{i})}$$

11.7 Exact roots and poles

11.7.1 Roots and poles of a rational function

The `froot` command finds roots and poles of a rational function.

- `froot` takes *rat*, a rational function.
- `froot(`*rat*`)` returns a vector whose components are the roots and the poles of *rat*, each one followed by its multiplicity.

  If XCAS cannot find the exact values of the roots or poles, it tries to find approximate values if *rat* has numeric coefficients.

Examples

```
> froot((x^5-2*x^4+x^3)/(x-2))
```

$$[1,2,0,3,2,-1]$$

Hence, for $F(x)=\frac{x^5-2x^4+x^3}{x-2}$:

- 1 is a root of multiplicity 2,
- 0 is a root of multiplicity 3,
- 2 is a pole of order 1.

```
> froot((x^3-2*x^2+1)/(x-2))
```

$$\left[1,1,\frac{\sqrt{5}+1}{2},1,\frac{-\sqrt{5}+1}{2},1,2,-1\right]$$

Remark. To find the complex roots and poles, put XCAS in complex mode (check the complex box in the CAS configuration, red button giving the state line; see Section 2.5.5, p. 14).

Example

Input in complex mode:

```
> froot((x^2+1)/(x-2))
```

$$[-\mathrm{i}, 1, \mathrm{i}, 1, 2, -1]$$

11.7.2 Rational function given by roots and poles

The `fcoeff` command finds a rational function given its roots and poles.

- `fcoeff` takes *roots*, a list consisting of the roots and poles of a rational function, each one followed by its multiplicity.
- `fcoeff(`*roots*`)` returns the rational function with the given roots and poles.

Example

```
> fcoeff([1,2,0,3,2,-1])
```

$$(x-1)^2\, x^3\, (x-2)^{-1}$$

11.8 Computing in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The way to compute over $\mathbb{Z}/p\mathbb{Z}$ or over $\mathbb{Z}/p\mathbb{Z}[x]$ depends on the syntax mode:

- In Xcas mode, an object n over $\mathbb{Z}/p\mathbb{Z}$ is written as `n%p`.

  The representation is the symmetric representation: e.g. `11%13` returns `-2%13`.
- In Maple mode, integers modulo p are represented like usual integers instead of using specific modular integers. To avoid confusion with standard commands, modular commands are written with a capital letter (inert form) and followed by the `mod` command.

  The Maple commands are discussed in Section 11.9, p. 263.

Examples

An integer `n` in $\mathbb{Z}/13\mathbb{Z}$:

```
> n:=12%13
```

A vector `V` in $\mathbb{Z}/13\mathbb{Z}$:

```
> V:=[1,2,3]%13}
```

or:

```
> V:=[1%13,2%13,3%13]
```

A matrix `A` in $\mathbb{Z}/13\mathbb{Z}$:

```
> A:=[[1,2,3],[2,3,4]]%13
```

or:

```
> A:=[[1%13,2%13,3%13],[[2%13,3%13,4%13]]
```

A polynomial `A` in $\mathbb{Z}/13\mathbb{Z}[x]$ in symbolic representation:

```
> A:=(2*x^2+3*x-1)%13
```

or:

```
> A:=2%13*x^2+3%13*x-1%13
```

A polynomial `A` in $\mathbb{Z}/13\mathbb{Z}[x]$ in list representation:

```
> A:=poly1[1,2,3]%13
```

or:

```
> A:=poly1[1%13,2%13,3%13]
```

To recover an object `obj` with integer coefficients instead of modular coefficients, input `obj%0`. For example:

```
> obj:=4%7;:
  obj%0
```

$$-3$$

Remark. Most Xcas functions that work on integers or polynomials with integer coefficients will often work analogously on $\mathbb{Z}/p\mathbb{Z}$ or $\mathbb{Z}/p\mathbb{Z}[x]$, with the obvious exception that the input and output will be modular. They will be listed in the remaining subsections. For some commands in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$, `p` must be a prime integer.

11.8.1 Expanding and reducing

The `normal` command expands and reduces expressions in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 10.1.2, p. 190.)

- `normal` takes *expr*, a modular expression.
- `normal(`*expr*`)` returns the expanded irreducible representation of *expr*.

Example

```
> normal(((2*x^2+12)*(5*x-4))%13)
```

$$((-3)\,\%\,13)\,x^3+(5\,\%\,13)\,x^2+((-5)\,\%\,13)\,x+4\,\%\,13$$

11.8.2 Addition in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The `+` operator adds two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 7.3.2, p. 129.) For polynomial expressions, use the `normal` command to simplify.

Examples

For integers in $\mathbb{Z}/p\mathbb{Z}$:

```
> 3%13+10%13
```

$$0\,\%\,13$$

For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

```
> normal((11*x+5)%13+(8*x+6)%13)
```

or:

```
> normal((11% 13*x+5%13)+(8% 13*x+6%13))
```

$$(6\,\%\,13)\,x+(-2)\,\%\,13$$

11.8.3 Subtraction in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The - operator subtracts two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 7.3.2, p. 129.) For polynomial expressions, use the `normal` command to simplify.

Examples

For integers in $\mathbb{Z}/p\mathbb{Z}$:

```
> 31%13-10%13
```

$$(-5) \,\%\, 13$$

For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

```
> normal((11*x+5)%13-(8*x+6)%13)
```

or:

```
> normal(11%13*x+5%13-(8%13*x+6%13))
```

$$(3 \,\%\, 13)\, x + (-1) \,\%\, 13$$

11.8.4 Multiplication in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The * operator multiplies two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 7.3.2, p. 129.) For polynomial expressions, use the `normal` command to simplify.

Examples

For integers in $\mathbb{Z}/p\mathbb{Z}$:

```
> 31%13*10%13
```

$$(-2) \,\%\, 13$$

For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

```
> normal((11*x+5)%13*(8*x+6)%13)
```

or:

```
> normal((11%13*x+5%13)*(8%13*x+6%13))
```

$$((-3) \,\%\, 13)\, x^2 + ((-24) \,\%\, 13)\, x + 17 \,\%\, 13$$

11.8.5 Euclidean quotient

The `quo` command finds the quotient of of two polynomials (see also Section 11.2.2, p. 225).

- `quo` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `quo`$(P, Q \,\langle, x\rangle)$ returns the Euclidean quotient of P divided by Q.

Example

```
> quo((x^3+x^2+1)%13,(2*x^2+4)%13)
```

$$((-6) \,\%\, 13)\, x + (-6) \,\%\, 13$$

Indeed, $x^3 + x^2 + 1 = (2x^2 + 4)\frac{x+1}{2} + \frac{5x-4}{4}$ and $-3 \cdot 4 = -6 \cdot 2 \equiv 1 \pmod{13}$.

11.8.6 Euclidean remainder

The `rem` command finds the remainder of the Euclidean division of two polynomials (see also Section 11.2.3, p. 226).

- `rem` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `rem(`$P, Q\,\langle, x\rangle$`)` returns the remainder of the Euclidean division of P divided by Q.

Example

```
> rem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

$$((-2)\,\%\,13)\,x + (-1)\,\%\,13$$

Indeed, $x^3 + x^2 + 1 = (2x^2 + 4) \cdot \frac{x+1}{2} + \frac{5x-4}{4}$ and $-3 \cdot 4 = -6 \cdot 2 \equiv 1 \pmod{13}$.

11.8.7 Euclidean quotient and euclidean remainder

The `quorem` command finds the quotient and remainder of the Euclidean division of two polynomials (see also Section 7.1.10, p. 110 and Section 11.2.4, p. 226).

- `quorem` takes two mandatory arguments and one optional argument:
  - P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
  - Optionally x, the variable (by default `x`), if P and Q are given as expressions.
- `quorem(`$P, Q\,\langle, x\rangle$`)` returns the list of the quotient and remainder of dividing P by Q.

Example

```
> quorem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

$$[((-6)\,\%\,13)\,x + (-6)\,\%\,13, ((-2)\,\%\,13)\,x + (-1)\,\%\,13]$$

Indeed, $x^3 + x^2 + 1 = (2x^2 + 4) \cdot \frac{x+1}{2} + \frac{5x-4}{4}$ and $-3 \cdot 4 = -6 \cdot 2 \equiv 1 \pmod{13}$.

11.8.8 Division in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The `/` operator divides two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials A and B in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 7.3.2, p. 129.) Since $\mathbb{Z}/p\mathbb{Z}$ is only a field if p is prime, the quotient is only guaranteed to exist if p is prime (unless the denominator is $0 \pmod{p}$).

For integers in $\mathbb{Z}/p\mathbb{Z}$. Since 13 is prime:

```
> 5%13/2%13
```

$$(-4)\,\%\,13$$

Since $3 \pmod{14}$ is invertible in $Z/14\mathbb{Z}$:

```
> 5%14/3%14
```

$$(-3)\,\%\,14$$

Since 7 (mod 14) is not invertible in $Z/14\mathbb{Z}$, this results in an error:

```
> 5%14/7%14
```

Not invertible Error: Bad Argument Value

For polynomials. The result of P/Q is its irreducible representation in $\mathbb{Z}/p\mathbb{Z}[x]$:

```
> (2*x^2+5)%13/(5*x^2+2*x-3)%13
```

$$\frac{(6\,\%\,13)\,x+1\,\%\,13}{(2\,\%\,13)\,x+(2\,\%\,13)\,\%\,13}$$

11.8.9 Power in $\mathbb{Z}/p\mathbb{Z}$ and in $\mathbb{Z}/p\mathbb{Z}[x]$

The ^ operator raises modular numbers and polynomials to powers in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 7.3.2, p. 129.) For polynomial expressions, use the `normal` command to simplify. Xcas uses the binary power algorithm to compute this.

Examples

```
> (5%13)^2
```

$$(-1)\,\%\,13$$

```
> normal(((2*x+1)%13)^5)
```

$$(6\,\%\,13)\,x^5+(2\,\%\,13)\,x^4+(2\,\%\,13)\,x^3+(1\,\%\,13)\,x^2+((-3)\,\%\,13)\,x+1\,\%\,13$$

because $10\equiv -3 \pmod{13}$, $40\equiv 1 \pmod{13}$, $80\equiv 2 \pmod{13}$, $32\equiv 6 \pmod{13}$.

11.8.10 Computing $a^n \pmod p$

For integers a, n and p, the `powmod` or `powermod` command finds $a^n \pmod p$.

- `powmod` takes three arguments: a, n and p, integers.
- `powmod`(a,n,p) returns $a^n \pmod p$ in $[0, p-1]$.

Examples

```
> powmod(5,2,13)
```

12

```
> powmod(5,2,12)
```

1

11.8.11 Inverse in $\mathbb{Z}/p\mathbb{Z}$

The `inv` or `inverse` command finds the inverse of an integer in $\mathbb{Z}/p\mathbb{Z}$.

Since $\mathbb{Z}/p\mathbb{Z}$ is only a field if p is prime, the inverse is only guaranteed to exist if p is prime (and the integer is non-zero).

- `inv` takes $n\%p$, an element of $\mathbb{Z}/p\mathbb{Z}$.
- `inv`$(n\%p)$ returns the reciprocal of $n\%p$ in $\mathbb{Z}/p\mathbb{Z}$.

Example

```
> inv(3%13)
```

$$(-4)\,\%\,13$$

Indeed, $3 \cdot (-4) = -12 \equiv 1 \pmod{13}$.

You can also find the reciprocal using division:

```
> 1/(3%13)
```

$$(-4)\,\%\,13$$

11.8.12 Rebuilding a fraction from its value modulo p

Given an integer n and a modulus p, the `fracmod` (or `iratrecon`, for MAPLE compatibility) command finds the rational number equal to $n \pmod{p}$, where both the numerator and denominator are not greater than $\sqrt{p}/2$ in absolute value.

- `fracmod` (or `iratrecon`) takes two arguments:
  - n, an integer (representing a fraction).
  - p, an integer (the modulus).
- `fracmod(`n, p`)` (or `iratrecon(`n, p`)`) returns, if possible, a fraction a/b such that

$$\begin{aligned} a &\equiv n \cdot b \pmod{p} \\ -\frac{\sqrt{p}}{2} < a &\leq \frac{\sqrt{p}}{2} \\ 0 \leq b &< \frac{\sqrt{p}}{2} \end{aligned}$$

  In other words, $n \equiv a/b \pmod{p}$.

Examples

```
> fracmod(3,13)
```

$$-\frac{1}{4}$$

Indeed, $3 \cdot (-4) = -12 \equiv 1 \pmod{13}$, hence $3 = -1/4\%13$.

Note that this means:

```
> -1/4%13
```

$$3\,\%\,13$$

```
> fracmod(13,121)
```

$$-\frac{4}{9}$$

Indeed, $13 \cdot (-9) = -117 \equiv 4 \pmod{121}$ and therefore $13 = -4/9\,\%\,13$.

11.8.13 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$

The `gcd` command finds the greatest common divisor of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (for prime p). (See also Section 7.1.1, p. 104 and Section 11.2.5, p. 227.)

- `gcd` takes two arguments: P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (p must be prime).
- `gcd(`P, Q`)` returns the GCD of P and Q computed in $\mathbb{Z}/p\mathbb{Z}[x]$

Example

```
> gcd((2*x^2+5)%13,(5*x^2+2*x-3)%13)
```

$$(1 \,\%\, 13)\, x + 2 \,\%\, 13$$

11.8.14 Factoring over $\mathbb{Z}/p\mathbb{Z}[x]$

The `factor` command factors polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 9.1.10, p. 172.)

- `factor` takes P, a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (p must be prime).
- `factor(`P`)` returns P in factored form.

Example

```
> factor((-3*x^3+5*x^2-5*x+4)%13)
```

$$((-3) \,\%\, 13)\, ((1 \,\%\, 13)\, x + (-6) \,\%\, 13)\, \left((1 \,\%\, 13)\, x^2 + 6 \,\%\, 13\right)$$

11.8.15 Determinant of a matrix in $\mathbb{Z}/p\mathbb{Z}$

The `det` command finds the determinant of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 15.1.4, p. 356.)

- `det` takes A, a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- `det(`A`)` returns the determinant of A.
  Computations are done in $\mathbb{Z}/p\mathbb{Z}$ by Gaussian reduction.

Example

```
> det([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
> det([[1,2,9],[3,10,0],[3,11,1]]%13)
```

$$5 \,\%\, 13$$

Hence, in $\mathbb{Z}/13\mathbb{Z}$, the determinant of $M = [[1, 2, 9], [3, 10, 0], [3, 11, 1]]$ is `5%13` (in $\mathbb{Z}$, `det(M)=31`).

11.8.16 Inverse of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$

The `inv` or `inverse` command finds the inverse of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 15.1.2, p. 355.)

- `inv` takes A, a matrix in $\mathbb{Z}/p\mathbb{Z}$.
- `inv(`A`)` returns the inverse of the matrix A.

Example

```
> inv([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
> inverse([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
> inv([[1,2,9],[3,10,0],[3,11,1]]%13)
```

or:

```
> inverse([[1,2,9],[3,10,0],[3,11,1]]%13)
```

$$\begin{bmatrix} 2\,\%\,13 & (-4)\,\%\,13 & (-5)\,\%\,13 \\ 2\,\%\,13 & 0\,\%\,13 & (-5)\,\%\,13 \\ (-2)\,\%\,13 & (-1)\,\%\,13 & 6\,\%\,13 \end{bmatrix}$$

11.8.17 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$

The `rref` command finds the reduced row echelon form of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$ (see 15.7.3).

- `rref` takes A, a matrix in $\mathbb{Z}/p\mathbb{Z}$.
- `rref`(A) returns the echelon form of A.

Example

```
> rref([[0,2,9]%15,[1,10,1]%15,[2,3,4]%15])
```

$$\begin{bmatrix} 1\,\%\,15 & 0\,\%\,15 & 0\,\%\,15 \\ 0\,\%\,15 & 1\,\%\,15 & 0\,\%\,15 \\ 0\,\%\,15 & 0\,\%\,15 & 1\,\%\,15 \end{bmatrix}$$

This can be used to solve a linear system of equations with coefficients in $\mathbb{Z}/p\mathbb{Z}$ by rewriting it in matrix form

$$AX = B. \tag{11.1}$$

`rref` can then take as argument the augmented matrix of the system (the matrix obtained by augmenting matrix A to the right with the column vector B).

`rref` returns a matrix $[A_1, B_1]$ where A_1 has ones on its principal diagonal and zeros outside. The solutions in $\mathbb{Z}/p\mathbb{Z}$ of:

$$A_1X = B_1$$

are the same as the solutions of (11.1).

Example

Solve in $\mathbb{Z}/13\mathbb{Z}$:

$$\begin{cases} x + 2y = 9, \\ 3x + 10y = 0. \end{cases}$$

```
> rref([[1,2,9]%13,[3,10,0]%13])
```

or:

```
> rref([[1,2,9],[3,10,0]])%13
```

$$\begin{bmatrix} 1\,\%\,13 & 0\,\%\,13 & 3\,\%\,13 \\ 0\,\%\,13 & 1\,\%\,13 & 3\,\%\,13 \end{bmatrix}$$

hence $x = 3 \,\%\, 13$ and $y = 3 \,\%\, 13$.

11.8.18 Construction of a Galois field

A *Galois field* is a finite field. A Galois field has the *characteristic* p for some prime number p, and its order is p^n for some integer n. Any Galois field of order p^n is isomorphic to $\mathbb{Z}/p\mathbb{Z}[X]/I$, where I is the ideal generated by an irreducible polynomial $P(X)$ in $\mathbb{Z}/p\mathbb{Z}[X]$

The `GF` command creates Galois fields.

- `GF` takes two mandatory arguments and one optional argument:
  - p, a prime number.
  - n, an integer greater than 1 (or an irreducible polynomial over $\mathbb{Z}/p\mathbb{Z}[X]$).

    If n is an integer, the first two arguments can be combined and entered as a prime power p^n.
  - Optionally *vars*, either the name of a variable or a list of two or three variables. These variables must be symbolic, so you should purge them if necessary.
- `GF(`$p, n \,\langle, vars\rangle$`)` returns a Galois field of characteristic p having p^n elements. The output will look like `GF(`$p, P(k), [k, K, g]$`,undef)` where:
  - p is the characteristic.
  - $P(k)$ is an irreducible polynomial generating an ideal I in $\mathbb{Z}/p\mathbb{Z}[X]$, the Galois field being the quotient of $\mathbb{Z}/p\mathbb{Z}[X]$ by I.
  - k is the name of the polynomial variable.
  - K is the name of the Galois field (which will be given to a free variable).
  - g is a generator of the multiplicative group K^*. You can build elements of the field with polynomials in g.

  If the optional argument *vars* is given:
  - *vars* consists of a variable name, then g is that variable name.
  - If *vars* consists of a pair of variable names, then k will be the first variable and K will be the second variable.

    In this case, there is no generator given and the elements of K must be given by $K(P(k))$ for a polynomial $P(k)$.
  - If *vars* consists of three variable names, then k will be the first variable, K will be the second variable and g will be the third variable.
- The elements of the field will be $0, g, g^2, \ldots, g^{p^n-2}$.
- Once a Galois field is created in Xcas, use elements of the field to create polynomials and matrices, and use the usual operators on them, such as `+`, `-`, `*`, `/`, `^`, `inv`, `sqrt`, `quo`, `rem`, `quorem`, `diff`, `factor`, `gcd`, `egcd`, etc.
- Note that there are still some limitations due to an incomplete implementation of some algorithms, such as multivariate factorization when the polynomial is not unitary.

Examples

To create a Galois field with parameters $p = 2$ and $n = 8$, input:

```
> GF(2,8)
```

$$\mathrm{GF}(2, k^8 + k^4 + k^3 + k^2 + 1, [k, K, g], \mathrm{undef})$$

The field K has $2^8 = 256$ elements and g generates the multiplicative group of this field ($\{1,g,g^2,\ldots,g^{254}\}$). The elements of this field can be written as polynomials in g or as K(P(k)), where P(k) is a polynomial in k.

```
> g^9
```

or:

```
> K(k^9)
```

$$(g^5+g^4+g^3+g)$$

Indeed, $g^8 = g^4+g^3+g^2+1$ and therefore $g^9 = g^5+g^4+g^3+g$.

Compute the inverse of a matrix in a Galois field:

```
> GF(3,5,b):; A:=[[1,b],[b,1]]:; inv(A)
```

$$\begin{bmatrix} (b^3+b^2-b) & (-b^4-b^3+b^2) \\ (-b^4-b^3+b^2) & (b^3+b^2-b) \end{bmatrix}$$

Factor a polynomial over a Galois field:

```
> GF(5,3,c):; p:=x^2-c-1:; factor(p)
```

$$\left((1\,\%\,5)\,x+\left(\left(-2c^2+2c\right)\right)\right)\left((1\,\%\,5)\,x+\left(\left(2c^2-2c\right)\right)\right)$$

As one can see in the these examples, the output contains many times the same information that you would prefer not to see if you work many times with the same field. For this reason, the definition of a Galois field may have an optional argument, a variable name which will be used thereafter to represent elements of the field. Since you will also most likely want to modify the name of the indeterminate, the field name is grouped with the variable name in a list passed as third argument to GF. Note that these two variable names must be quoted.

```
> G:=GF(2,2,['w','G']):; G(w^2)
```

$$\text{Done}, G\,(w+1)$$

```
> G(w^3)
```

$$G(1)$$

Hence, the elements of GF(2,2) are G(0), G(1), G(w) and G(w^2)=G(w+1).

We may also impose the irreducible primitive polynomial that we wish to use, by putting it as second argument (instead of n), for example:

```
> G:=GF(2,w^8+w^6+w^3+w^2+1,['w','G'])
```

If the polynomial is not primitive, Xcas will replace it automatically by a primitive polynomial, for example:

```
> G:=GF(2,w^8+w^7+w^5+w+1,['w','G'])
```

$$\text{GF}\left(2, w^8+w^7+w^5+w+1, [w,G], \text{undef}\right)$$

11.8.19 Factoring a polynomial with coefficients in a Galois field

The factor command can factor univariate polynomials with coefficients in a Galois field.

- factor takes one mandatory argument and one optional argument:
  - *expr*, an expression or a list of expressions.

– Optionally, α, to specify an extension field.

- `factor(`*expr*`)` returns *expr* factored over the field of its coefficients, with the addition of i in complex mode (see Section 2.5.5, p. 14). If `sqrt` is enabled in the CAS configuration (see Section 2.5.7, p. 15), polynomials of order 2 are factorized in complex mode or in real mode if the discriminant is positive.

  `factor(`*expr*$,\alpha$`)` returns *expr* factored over $F[\alpha]$, where F is the field of coefficients of *expr*.

- `cfactor` factors like `factor`, except the field includes i whether in real or complex mode.

Example

The command `factor` can factorize a univariate polynomial with coefficients in a Galois field. For example, to have $G = \mathbb{F}_4$, input:

```
> G:=GF(2,2,['w','G'])
```

$$\mathrm{GF}\left(2, w^2+w+1, [w, G], \mathrm{undef}\right)$$

Now input, for example:

```
> a:=G(w):;
  factor(a^2*x^2+1)
```

$$G\,(w)^2\,x^2+1$$

11.9 Computing in $\mathbb{Z}/p\mathbb{Z}[x]$ using Maple syntax

You can set Xcas to work in Maple mode rather than native Xcas mode (see Section 2.5.2, p. 14).

11.9.1 Euclidean quotient

In Xcas mode, `Quo` is simply the inert form of `quo`; namely, it returns the Euclidean quotient of two polynomials without evaluation. (See Section 11.2.2, p. 225.) In Maple mode, the `Quo` command can additionally be used in conjunction with `mod` to compute the Euclidean quotient of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- In Maple mode, `Quo` takes two arguments: P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- `Quo(`P, Q`)` returns the Euclidean quotient of P divided by Q.

Examples

Input in Xcas mode:

```
> Quo((x^3+x^2+1) mod 13,(2*x^2+4) mod 13)
```

$$\mathrm{quo}\left((1\,\%\,13)\,x^3+(1\,\%\,13)\,x^2+1\,\%\,13, (2\,\%\,13)\,x^2+4\,\%\,13\right)$$

To get the result of the division:

```
> eval(ans())
```

$$((-6)\,\%\,13)\,x+(-6)\,\%\,13$$

Input in Maple mode:

```
> Quo(x^3+x^2+1,2*x^2+4) mod 13
```

$$-6x-6$$

Input in MAPLE mode:

```
> Quo(x^2+2*x,x^2+6*x+5) mod 5
```

$$1$$

11.9.2 Euclidean remainder

In XCAS mode, `Rem` is simply the inert form of `rem`; namely, it returns the Euclidean remainder of two polynomials without evaluation. (See Section 11.2.3, p. 226.) In MAPLE mode, the `Rem` command can additionally be used in conjunction with `mod` to compute the Euclidean remainder of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- In MAPLE mode, `Rem` takes two arguments: P and Q, two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- `Rem`(P,Q) returns the Euclidean remainder of P divided by Q.

Examples

Input in XCAS mode:

```
> Rem((x^3+x^2+1) mod 13,(2*x^2+4) mod 13)
```

$$\mathrm{rem}\left((1\,\%\,13)\,x^3+(1\,\%\,13)\,x^2+1\,\%\,13,(2\,\%\,13)\,x^2+4\,\%\,13\right)$$

To get the result of the division:

```
> eval(ans())
```

$$((-2)\,\%\,13)\,x+(-1)\,\%\,13$$

Input in MAPLE mode:

```
> Rem(x^3+x^2+1,2*x^2+4) mod 13
```

$$-2x-1$$

```
> Rem(x^2+2*x,x^2+6*x+5) mod 5
```

$$x$$

11.9.3 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$

In XCAS mode, `Gcd` is simply the inert form of `gcd`; namely, it returns the greatest common divisor of two polynomials without evaluation. (See Section 11.2.5, p. 227.) In MAPLE mode, the `Gcd` command can additionally be used in conjunction with `mod` to compute the greatest common divisor of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- In MAPLE mode, `Gcd` takes *polys*, a sequence or list of polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- `Gcd`(*polys*) returns the greatest common divisor of the polynomials in `polys`.

Examples

Input in Xcas mode:

```
> Gcd(2*x^2+5%13,5*x^2+2*x-3%13)
```

$$\gcd\left(2x^2+5\,\%\,13, 5x^2+2x+(-3)\,\%\,13\right)$$

To get the actual greatest common divisor:

```
> eval(ans())
```

$$(1\,\%\,13)\,x+2\,\%\,13$$

Input in Maple mode:

```
> Gcd(2*x^2+5,5*x^2+2*x-3) mod 13
```

$$x+2$$

```
> Gcd(x^2+2*x,x^2+6*x+5) mod 5
```

$$x$$

11.9.4 Factoring in $\mathbb{Z}/p\mathbb{Z}[x]$

In Xcas mode, `Factor` is simply the inert form of `factor`; namely, it factors a polynomial without evaluation. (See Section 9.1.10, p. 172.) In Maple mode, the `Factor` command can additionally be used in conjunction with `mod` to factor a polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where p must be prime.

- In Maple mode, `Factor` takes P, a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}$ for prime p.
- `Factor(`P`)` returns the factored form of P.

Example

Input in Xcas mode:

```
> Factor((-3*x^3+5*x^2-5*x+4)%13)
```

$$\text{factor}\left(((-3)\,\%\,13)\,x^3+(5\,\%\,13)\,x^2+((-5)\,\%\,13)\,x+4\,\%\,13\right)$$

To get the actual factorization:

```
> eval(ans())
```

$$((-3)\,\%\,13)\,((1\,\%\,13)\,x+(-6)\,\%\,13)\left((1\,\%\,13)\,x^2+6\,\%\,13\right)$$

Input in Maple mode:

```
> Factor(-3*x^3+5*x^2-5*x+4) mod 13
```

$$-3\,(x-6)\left(x^2+6\right)$$

11.9.5 Determinant of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$

In Xcas mode, `Det` is simply the inert form of `det`; namely, it gives the determinant of a matrix without evaluating it. (See Section 15.1.4, p. 356.) In Maple mode, the `Det` command can additionally be used in conjunction with `mod` to find the determinant of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- In Maple mode, `Det` takes A, a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- `Det(`A`)` returns the determinant of A.

Example

Input in Xcas mode:

```
> Det([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1] mod 13])
```

$$\det\left(\begin{bmatrix}1\,\%\,13 & 2\,\%\,13 & (-4)\,\%\,13\\ 3\,\%\,13 & (-3)\,\%\,13 & 0\,\%\,13\\ 3\,\%\,13 & (-2)\,\%\,13 & 1\,\%\,13\end{bmatrix}\right)$$

To find the value of the determinant, enter:

```
> eval(ans())
```

$$5\,\%\,13$$

Hence, in $\mathbb{Z}/13\mathbb{Z}$, the determinant of $A=[[1,2,9],[3,10,0],[3,11,1]]$ is $5\,\%\,13$ (in $\mathbb{Z}$, `det(A)=31`).

Input in Maple mode:

```
> Det([[1,2,9],[3,10,0],[3,11,1]]) mod 13
```

$$5$$

11.9.6 Inverse of a matrix in $\mathbb{Z}/p\mathbb{Z}$

In Xcas mode, `Inverse` is simply the inert form of `inverse`; namely, it gives the inverse of a matrix without evaluating it. (See Section 15.1.2, p. 355.) In Maple mode, the `Inverse` command can additionally be used in conjunction with `mod` to find the inverse of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- In Maple mode, `Inverse` takes A, a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- `Inverse(`A`)` returns the inverse of A.

Example

Input in Xcas mode:

```
> Inverse([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1] mod13])
```

$$\text{inverse}\left(\begin{bmatrix}1\,\%\,13 & 2\,\%\,13 & (-4)\,\%\,13\\ 3\,\%\,13 & (-3)\,\%\,13 & 0\,\%\,13\\ 3\,\%\,13 & (-2)\,\%\,13 & 1\,\%\,13\end{bmatrix}\right)$$

To get the actual inverse, enter:

```
> eval(ans())
```

$$\begin{bmatrix}2\,\%\,13 & (-4)\,\%\,13 & (-5)\,\%\,13\\ 2\,\%\,13 & 0\,\%\,13 & (-5)\,\%\,13\\ (-2)\,\%\,13 & (-1)\,\%\,13 & 6\,\%\,13\end{bmatrix}$$

which is the inverse of $A=[[1,2,9],[3,10,0],[3,11,1]]$ in $\mathbb{Z}/13\mathbb{Z}$.

Input in Maple mode:

```
> Inverse([[1,2,9],[3,10,0],[3,11,1]]) mod 13
```

$$\begin{bmatrix}2 & -4 & -5\\ 2 & 0 & -5\\ -2 & -1 & 6\end{bmatrix}$$

11.9.7 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$

In XCAS mode, `Rref` is simply the inert form of `rref`; namely, it returns `rref` without evaluating it (see Section 15.7.3, p. 392). In MAPLE mode, the `Rref` command can additionally be used in conjunction with `mod` to find the reduced row echelon form of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- In MAPLE mode, `Rref` takes A, a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- `Rref(`A`)` returns the reduced row echelon form of A.

Example

Solve in $\mathbb{Z}/13\mathbb{Z}$:

$$\begin{cases} x + 2y = 9, \\ 3x + 10y = 0. \end{cases}$$

Input in XCAS mode:

```
> Rref([[1,2,9] mod 13,[3,10,0] mod 13])
```

$$\text{rref}\left(\begin{bmatrix} 1 \,\%\, 13 & 2 \,\%\, 13 & (-4) \,\%\, 13 \\ 3 \,\%\, 13 & (-3) \,\%\, 13 & 0 \,\%\, 13 \end{bmatrix}\right)$$

To actually get the reduced echelon form, enter:

```
> eval(ans())
```

$$\begin{bmatrix} 1 \,\%\, 13 & 0 \,\%\, 13 & 3 \,\%\, 13 \\ 0 \,\%\, 13 & 1 \,\%\, 13 & 3 \,\%\, 13 \end{bmatrix}$$

Input in MAPLE mode:

```
> Rref([[1,2,9],[3,10,0]] mod 13)
```

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \end{bmatrix}$$

In both cases you conclude that $x = 3 \,\%\, 13$ and $y = 3 \,\%\, 13$.

12 Combinatorics

12.1 Combinatorial analysis

12.1.1 Factorials

The `factorial` or its postfix operator equivalent `!` command computes the factorial of a number.

- `factorial` takes n, an integer.
- `factorial(`n`)` returns $n!$.
- The Γ function (see Section 7.3.13, p. 136) is used to extend the factorial function to complex numbers. The Γ function is defined for all complex numbers except for zero and the negative integers, and it satisfies $\Gamma(n+1) = n!$ for all non-negative integers n. So the factorial can be extended to all complex numbers except the negative integers by $n! = \Gamma(n+1)$.

Examples

```
> factorial(10)
```

or:

```
> 10!
```

$$3628800$$

```
> factorial(1/2)
```

$$\frac{\sqrt{\pi}}{2}$$

```
> factorial(i)
```

$$0.5 - 0.2\mathrm{i}$$

12.1.2 Binomial coefficients

The `comb` or `nCr` command computes the binomial coefficients.

- `comb` takes two arguments: n and p, integers.
- `comb(n,p)` returns $\binom{n}{p} = C_n^p$.

Example

```
> comb(5,2)
```

$$10$$

Remark. The `binomial` command (see Section 20.4.3, p. 535) can also compute the binomial coefficients, but unlike `comb` and `nCr` it can take an optional third argument, a real number a, to compute the binomial distribution.

12.1.3 Counting permutations

The `perm` or `nPr` command counts permutations.

- `perm` takes two arguments: n and p, integers.
- `perm(`n, p`)` returns P_n^p, the number of permutations of n objects taken p at a time.

Example

```
> perm(5,2)
```

20

12.1.4 Wilf-Zeilberger pairs

The Wilf-Zeilberger certificate $R(n,k)$ is used to prove the identity

$$\sum_k U(n,k) = Cres(n)$$

for some constant C (typically 1) whose value can be determined by evaluating both sides for some value of k. To see how that works, note that the above identity is equivalent to

$$\sum_k F(n,k)$$

being constant, where $F(n,k) = U(n,k)/res(n)$. The Wilf-Zeilberger certificate is a rational function $R(n,k)$ that make $F(n,k)$ and $G(n,k) = R(n,k)F(n,k)$ a Wilf-Zeilberger pair, meaning

- $F(n+1,k) - F(n,k) = G(n,k+1) - G(n,k)$ for integers $n \geq 0$, k.
- $\lim_{k\to\pm\infty} G(n,k) = 0$ for each $n \geq 0$.

To see how this helps, adding the first equation from $k = -M$ to $k = N$ gives you $\sum_{k=-M}^{N}(F(n+1,k) - F(n,k)) = \sum_{k=-M} N(G(n,k+1) - G(n,k))$. The right-hand side is a telescoping series, and so the equality can be written

$$\sum_{k=-M}^{N} F(n+1,k) - \sum_{k=-M}^{N} F(n,k) = G(n,N+1) - G(n,-M).$$

Taking the limit as $N, M \to \infty$ and using the second condition of Wilf-Zeilberger pairs, you get

$$\sum_k F(n+1,k) = \sum_k F(n,k)$$

and so $\sum_k F(n,k)$ does not depend on n, and so is a constant.

The `wz_certificate` command computes Wilf-Zeilberger pairs.

- `wz_certificate` takes four arguments:
  - $U(n,k)$, an expression in two variables.
  - $res(k)$, an expression in one of the variables.
  - n and k, the variables.
- `wz_certificate(`$U(n,k), res(k), n, k$`)` returns the Wilf-Zeilberger certificate $R(n,k)$ for the identity $\sum_{k=-\infty}^{+\infty} U(n,k) = res(n)$.

Example

To show that

$$\sum_k (-1)^k \binom{n}{k}\binom{2k}{k}4^{n-k} = \binom{2n}{n}, \tag{12.1}$$

input:

```
> wz_certificate((-1)^k*comb(n,k)*comb(2k,k)*4^(n-k),comb(2n,n),n,k)
```

$$\frac{2k-1}{2n+1}$$

This means that $R(n,k) = \frac{2k-1}{2n+1}$ is a Wilf-Zeilberger certificate. In other words, $F(n,k) = \frac{(-1)^k\binom{n}{k}\binom{2k}{k}4^{n-k}}{\binom{2n}{n}}$ and $G(n,k) = R(n,k)F(n,k)$ form a Wilf-Zeilberger pair. So $\sum_k F(n,k)$ is a constant. Since $F(0,0) = 1$ and $F(0,k) = 0$ for $k > 0$, we have $\sum_k F(0,k) = 1$ and so $\sum_k F(n,k) = 1$ for all n, implying (12.1).

12.2 Permutations

A *permutation* p of size n is a bijection from $\{0,\ldots,n-1\}$ to itself and is represented by the list:

$$[p(0), p(1), p(2), \ldots, p(n-1)].$$

For example, the permutation p represented by $[1,3,2,0]$ is the function from $[0,1,2,3]$ to $[0,1,2,3]$ defined by $p(0) = 1$, $p(1) = 3$, $p(2) = 2$, $p(3) = 0$.

A *cycle* c of size p, represented by the list $[a_0,\ldots,a_{p-1}]$ $(0 \le a_k \le n-1)$, is the permutation such that

$$c(a_i) = a_{i+1} \text{ for } 0 \le i \le p-2, \quad c(a_{p-1}) = a_0, \quad c(k) = k \text{ otherwise.}$$

For example, the cycle c represented by the list $[3,2,1]$ is the permutation c defined by $c(3) = 2$, $c(2) = 1$, $c(1) = 3$, $c(0) = 0$ (i.e. the permutation represented by the list $[0,3,1,2]$).

12.2.1 Random permutations

The `randperm` or `shuffle` command computes a random permutation.

- `randperm` takes n, an integer.
- `randperm(`n`)` returns a random permutation of $[0,1,\ldots,n-1]$.

Example

```
> randperm(3)
```

$$[2,0,1]$$

12.2.2 Previous and next permutation

The set of n-tuples of an ordered set can be put in *lexicographic order*, where the tuple $(a_1,a_2,\ldots,a_n)$ comes before $(b_1,b_2,\ldots,b_n)$ exactly when for some k (possibly $k = 0$), $a_i = b_i$ for $i = 1,\ldots,k-1$ and $a_k < b_k$. For example, the list of all permutations of size 3 in lexicographic order is: $(0,1,2)$, $(0,2,1)$, $(1,0,2)$, $(1,2,0)$, $(2,0,1)$, $(2,1,0)$.

The `prevperm` and `nextperm` commands find the preceding and succeeding permutation.

- `prevperm` takes p, a permutation.

- `prevperm(`p`)` returns the previous permutation in lexicographic order, or `undef` if there is no previous permutation.
- `nextperm` takes p, a permutation.
- `nextperm(`p`)` returns the next permutation in lexicographic order, or `undef` if there is no next permutation.

Examples

```
> prevperm([0,3,1,2])
```

$$[0, 2, 3, 1]$$

```
> nextperm([0,2,3,1])
```

$$[0, 3, 1, 2]$$

12.2.3 Decomposing a permuation into a product of disjoint cycles

Any permutation can be decomposed as a sequence of cycles which have no elements in common. For example, the permutation $[1, 3, 4, 0, 2]$ can be written as a combination of the cycles $[0, 1, 3]$ and $[2, 4]$.

The `permu2cycles` command decomposes a permutation into a combination of cycles.

- `permu2cycles` takes p, a permutation.
- `permu2cycles(`p`)` returns the decomposition of p as a product of disjoint cycles. A cycle is represented by a list, a cyclic decomposition is represented by a list of lists.

Examples

```
> permu2cycles([1,3,4,5,2,0])
```

$$[[0, 1, 3, 5], [2, 4]]$$

In the answer the cycles of size 1 are omitted, except if $n - 1$ is a fixed point of the permutation (this is required to find the value of n from the cycle decomposition).

```
> permu2cycles([0,1,2,4,3,5])
```

$$[[5], [3, 4]]$$

```
> permu2cycles([0,1,2,3,5,4])
```

$$[[4, 5]]$$

12.2.4 Product of cycles to permutation

The `cycles2permu` command does the opposite of `perm2cycles`; it turns a sequence of cycles into a permutation.

- `cycles2permu` takes c, a list of cycles.
- `cycles2permu(`c`)` returns the permutation (of size n chosen as small as possible) that is the product of the given cycles.

Examples

```
> cycles2permu([[1,3,5],[2,4]])
```

$$[0, 3, 4, 5, 2, 1]$$

```
> cycles2permu([[2,4]])
```

$$[0, 1, 4, 3, 2]$$

```
> cycles2permu([[5],[2,4]])
```

$$[0, 1, 4, 3, 2, 5]$$

12.2.5 Transforming a cycle into a permutation

A cycle is a type of permutation, but has a different representation.

The `cycle2perm` command converts a cycle to the cycle written as a permutation.

- `cycle2perm` takes c, a cycle.
- `cycle2perm`(c) returns the permutation of size n corresponding to the cycle c, where n is chosen as small as possible (see also `permu2cycles` and `cycles2permu`).

Example

```
> cycle2perm([1,3,5])
```

$$[0, 3, 2, 5, 4, 1]$$

12.2.6 Transforming a permutation into a matrix

The matrix of a permutation p of size n is the matrix obtained by permuting the rows of the identity matrix of size n with the permutation p. Multiplying this matrix by a column vector of size n is the same as permuting the elements of the vector with the permutation p.

The `permu2mat` command finds the matrix of a given permutation.

- `permu2mat` takes p, a permutation p.
- `permu2mat`(p) returns the matrix of the permutation p.

Example

```
> permu2mat([2,0,1])
```

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

12.2.7 Checking for a permutation

A permutation can be written as a list, but not every list corresponds to a permutation. The `is_permu` is a boolean function which checks to see if a given list is a permutation.

- `is_permu` takes L, a list.
- `is_permu`(L) returns 1 if L is a permutation and returns 0 if L is not a permutation.

Examples

```
> is_permu([2,1,3])
```

0

```
> is_permu([2,1,3,0])
```

1

12.2.8 Checking for a cycle

The `is_cycle` command is a boolean function which checks to see if a list represents a cycle.

- `is_cycle` takes L, a list.
- `is_cycle(`L`)` returns 1 if L is a cycle and returns 0 if L is not a cycle.

Examples

```
> is_cycle([2,1,3])
```

1

```
> is_cycle([2,1,3,2])
```

0

12.2.9 Product of two permutations

Permutations are functions, and so can be composed. Since cycles can be represented differently than other permutations, there are commands for composing permutations of different types.

The `p1op2` command composes two permutations.

- `p1op2` takes two arguments: p_1 and p_2, permutations.
- `p1op2(`p_1, p_2`)` returns the permutation $p_1 \circ p_2$ obtained by composition. Note that this is the standard mathematical notation; that is, the rightmost permutation is applied first.

The `c1op2` command composes a cycle and a permutation.

- `c1op2` takes two arguments:
  - c_1, a cycle.
  - p_2, a permutation.
- `c1op2(`c_1, p_2`)` returns the permutation $c_1 \circ p_2$ obtained by composition.

The `p1oc2` command composes a permutation and a cycle.

- `p1oc2` takes two arguments:
  - p_1, a permutation.
  - c_2, a cycle.

  `p1oc2(`p_1, c_2`)` returns the permutation $p_1 \circ c_2$ obtained by composition.

The `c1oc2` command composes two cycles.

- `c1oc2` takes two arguments: c_1 and c_2, cycles.
- `c1oc2(`c_1, c_2`)` returns the permutation $c_1 \circ c_2$ obtained by composition.

Examples

```
> p1op2([3,4,5,2,0,1],[2,0,1,4,3,5])
```

$$[5, 3, 4, 0, 2, 1]$$

```
> c1op2([3,4,5],[2,0,1,4,3,5])
```

$$[2, 0, 1, 5, 4, 3]$$

```
> p1oc2([3,4,5,2,0,1],[2,0,1])
```

$$[4, 5, 3, 2, 0, 1]$$

```
> c1oc2([3,4,5],[2,0,1])
```

$$[1, 2, 0, 4, 5, 3]$$

12.2.10 Signature of a permutation

Every permutation can be decomposed into a product of transpositions (cycles with only two elements). The number of transpositions is not unique, but for any permutation the number will be either odd or even. The signature of a permutation is equal to:

- 1 if the permutation is equal to an even product of transpositions,
- -1 if the permutation is equal to an odd product of transpositions.

The signature of a cycle of size k is: $(-1)^{k+1}$.

The `signature` command computes the signature of a permutation.

- `signature` takes p, a permutation.
- `signature(`p`)` returns the signature of the permutation p.

Example

```
> signature([3,4,5,2,0,1])
```

$$-1$$

12.2.11 Inverse of a permutation

Every permutation has an inverse, which is also a permutation.

The `perminv` command computes the inverse of a permutation.

- `perminv` takes p, a permutation.
- `perminv(`p`)` returns the permutation that is the inverse of p.

Example

```
> perminv([1,2,0])
```

$$[2, 0, 1]$$

12.2.12 Inverse of a cycle

The inverse of a cycle will be another cycle.

The `cycleinv` command computes the inverse of a cycle.

- `cycleinv` takes c, a cycle.
- `cycleinv`(c) returns the cycle that is the inverse of c.

Example

```
> cycleinv([2,0,1])
```

$$[1, 0, 2]$$

12.2.13 Order of a permutation

If any permutation p on a finite set $[0, \ldots, n-1]$ is repeated often enough, it reach be the identity permutation. The smallest m such that p^m is the identity is called the *order* of p.

The `permuorder` command computes the order of a permutation.

- `permuorder` takes p, a permutation.
- `permuorder`(p) returns the order of the permutation p.

Examples

```
> permuorder([0,2,1])
```

$$2$$

```
> permuorder([3,2,1,4,0])
```

$$6$$

12.2.14 Group generated by two permutations

Given permutations a and b, the group they generate is the set of all possible compositions of any number of as and any number of bs.

The `groupermu` command computes the group generated by two permutations.

- `groupermu` takes two arguments: a and b, permutations.
- `groupermu`(a, b) returns the group of the permutations generated by a and b.

Example

```
> groupermu([0,2,1,3],[3,1,2,0])
```

$$\begin{bmatrix} 0 & 2 & 1 & 3 \\ 3 & 1 & 2 & 0 \\ 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

13 Calculus

13.1 Limits

13.1.1 Univariate function limits

The `limit` command computes limits, both at numbers and infinities, and in the real case it can compute one-sided limits.

- `limit` takes three mandatory and one optional argument.
  - *expr*, an expression.
  - x, the name of a variable.
  - a, the limit point.
  - Optionally, *side* (either 0, −1 or 1), to specify which side to take a one-sided limit (by default $side = 0$).

  Alternatively, one-sided limits may be specified by using symbols `left` and `right` instead of −1 and 1.
- `limit(`*expr*, x, a ⟨, *side*⟩`)` returns the limit of *expr* as x approaches a.
  - If $side = 0$ (the default), then the ordinary limit is returned.
  - If $side = -1$ or $side =$ `left`, then the limit from the left ($x < a$) is returned.
  - If $side = 1$ or $side =$ `right`, then the limit from the right ($x > a$) is returned.
- It is also possible to put `x=a` as argument instead of `x,a`; `limit(`*expr*, *var*`=`*pt* ⟨, *side*⟩`)` is equivalent to `limit(`*expr*, *var*, *pt* ⟨, *side*⟩`)`.

Examples

To find $\lim_{x\to 0^-} \frac{1}{x}$, input:

```
> limit(1/x,x,0,-1)
```

or:

```
> limit(1/x,x=0,-1)
```

$$-\infty$$

To find $\lim_{x\to 0^+} \frac{1}{x}$, input:

```
> limit(1/x,x,0,1)
```

or:

```
> limit(1/x,x=0,1)
```

$$+\infty$$

To find $\lim_{x\to 0} \frac{1}{x}$, input:

```
> limit(1/x,x,0,0)
```

or:

```
> limit(1/x,x,0)
```

or:

```
> limit(1/x,x=0)
```

$$\infty$$

Note that ∞ or *infinity* without an explicit + or - represents unsigned infinity.

Find, for $n > 2$, the limit of $\frac{n\tan(x)-\tan(nx)}{\sin(nx)-n\sin(x)}$ as x approaches 0.

```
> limit((n*tan(x)-tan(n*x))/(sin(n*x)-n*sin(x)),x=0)
```

$$2$$

Note that Xcas does not complain about a possibility of n being equal to 1.

Find the limit of $\sqrt{x+\sqrt{x+\sqrt{x}}}-\sqrt{x}$ as x approaches $+\infty$.

```
> limit(sqrt(x+sqrt(x+sqrt(x)))-sqrt(x),x=+infinity)
```

$$\frac{1}{2}$$

Find the limit of $\frac{\sqrt{1+x+x^2/2}-e^{x/2}}{(1-\cos(x))\sin(x)}$ as x approaches 0.

```
> limit((sqrt(1+x+x^2/2)-exp(x/2))/((1-cos(x))*sin(x)),x,0)
```

$$-\frac{1}{6}$$

13.2 Derivative

13.2.1 Derivatives and partial derivatives

The `diff` or `derive` or `deriver` command computes derivatives and partial derivatives of expressions.

- To compute first-order derivatives, `diff` takes one mandatory argument and one optional argument:
  - *expr*, an expression or a list of expressions.
  - Optionally, x, a variable (resp. a list of variable names, see several variable functions in Section 13.7, p. 308). If the only variable is `x`, this second argument can be omitted.
- `diff(`*expr*$\langle, x\rangle$`)` returns the derivative (resp. a vector of derivatives) of the expression *expr* (or list of expressions) with respect to the variable x (resp. with respect to each variable in the list x).
- To compute higher order derivatives, `diff` takes more than two arguments:
  - *expr*, an expression.
  - $x_1, x_2 \ldots$, the names of the derivation variables. Note that for repeated variables, you can use the `$` operator (see Section 6.1.2, p. 67) followed by the number of derivations with respect to the variable; for example, instead of writing `x,x,x` you could write `x$3`.
- `diff(`*expr*$, x_1, x_2, \ldots$`)` returns the partial derivative of *expr* with respect to the variables $x_1, x_2, \ldots$

Examples

Compute $\frac{\partial}{\partial z}(xy^2z^3+xyz)$.

```
> diff(x*y^2*z^3+x*y*z,z)
```

$$3xy^2z^2+xy$$

Compute the first order partial derivatives of xy^2z^3+xyz.

```
> diff(x*y^2*z^3+x*y,[x,y,z])
```

$$\left[y^2z^3+y, 2xyz^3+x, 3xy^2z^2\right]$$

Compute $\frac{\partial^3}{\partial y\,\partial^2 z}(xy^2z^3+xyz)$.

```
> diff(x*y ^2*z^3+x*y*z,y,z$2)
```

$$12xyz$$

Compute $\frac{\partial^2}{\partial x\partial z}(xy^2z^3+xyz)$.

```
> diff(x*y^2*z^3+x*y*z,x,z)
```

$$3y^2z^2+y$$

Compute $\frac{\partial^3}{\partial x\partial^2 z}(xy^2z^3+xyz)$.

```
> diff(x*y^2*z^3+x*y*z,x,z,z)
```

or:

```
> diff(x*y^2*z^3+x*y*z,x,z$2)
```

$$6y^2z$$

Compute the third derivative of $\frac{1}{x^2+2}$.

```
> normal(diff((1)/(x^2+2),x,x,x))
```

or:

```
> normal(diff((1)/(x^2+2),x$3))
```

$$\frac{-24x^3+48x}{x^8+8x^6+24x^4+32x^2+16}$$

Remarks.

- Note the difference between `diff(`f,x,y`)` and `diff(`f`,[`x,y`])`: the first returns $\frac{\partial^2 f}{\partial x\,\partial y}$ while the second returns $\left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$.
- Never define a derivative function with `f1(x):=diff(f(x),x)`. Indeed, `x` would mean two different things Xcas is unable to deal with: on the left hand side, `x` is the variable name to define the f_1 function, and on the right hand side, `x` is the differentiation variable. The right way to define a derivative is either with `function_diff` (see Section 13.2.2, p. 278) or with `f1:=unapply(diff(f(x),x),x)`.

13.2.2 Functional derivative

The `function_diff` command finds the derivatives of functions (as opposed to expressions, see Section 8.2.1, p. 148).

- `function_diff` takes f, a function.
- `function_diff(`f`)` returns the derivative f' of f.

Examples

```
> function_diff(sin)
```

$$x \mapsto \cos x$$

```
> function_diff(sin)(x)
```

$$\cos x$$

```
> f(x):=x^2+x*cos(x):;
  function_diff(f)
```

$$x \mapsto \cos x - x \sin x + 2x$$

```
> function_diff(f)(x)
```

$$\cos x - x \sin x + 2x$$

To define the function g as f':

```
> g:=function_diff(f)
```

The `function_diff` instruction has the same effect as using the expression derivative `diff` (see Section 13.2.1, p. 277) in conjunction with `unapply` (see Section 8.2.2, p. 148). For example:

```
> g:=unapply(diff(f(x),x),x):;
  g(x)
```

$$\cos x - x \sin x + 2x$$

Remark. In MAPLE mode (see Section 2.5.2, p. 14), for compatibility, `D` may be used in place of `function_diff`. For this reason, it is impossible to assign a variable named `D` in MAPLE mode (hence you cannot name a geometric object `D`).

13.2.3 Implicit differentiation

The `implicitdiff` command can differentiate implicitly defined functions or expressions containing implicitly defined functions. It has three different calling sequences.

- To implicitly differentiate dependent variables, `implicitdiff` takes four arguments:
  - *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form

    $$g_i(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$$

    for $i = 1, 2, \ldots, m$, where $x_1, \ldots, x_n$ are the independent variables and $y_1, \ldots, y_m$ are the dependent variables.
  - *depvars*, the list of dependent variables, where each dependent variable can optionally be written as a function of the x_i or the name written as a function of the independent variables $y_i(x_1, \ldots, x_n)$. If there is only one dependent variable, this can be omitted.
  - y, a dependent variable or a list of dependent variables to be differentiated.
  - *diffvars*, a sequence of independent variables $x_{i_1}, \ldots, x_{i_k}$ with respect to differentiate.
- `implicitdiff(`*constraints* ⟨, *depvars*⟩, *y*, *diffvars*`)` returns the derivative (or list of derivatives) of y with respect to *diffvars*.

- To find a specified derivative of an expression containing implicitly defined functions, `implicitdiff` takes four arguments:
  - *expr*, a differentiable expression involving independent variables $x_1, x_2, \ldots, x_n$ and dependent variables $y_1, y_2, \ldots, y_m$.
  - *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form
    $$g_i(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$$
    for $i = 1, 2, \ldots, m$.
  - *depvars*, the dependent variable or list of dependent variables, where each dependent variable can either be the variable name y_i or the name written as a function of the independent variables $y_i(x_1, \ldots, x_n)$).
  - *diffvars*, a sequence of independent variables $x_{i_1}, \ldots, x_{i_k}$ with respect to which *expr* is differentiated.
- `implicitdiff(`*expr*, *implicitdef*, *depvars*, *diffvars*`)` returns the expression *expr* differentiated with respect to *diffvars*.
- To find all kth order derivatives of an expression involving implicitly defined functions, `implicitdiff` takes four mandatory arguments and one optional argument:
  - *expr*, a differentiable expression involving independent variables $x_1, x_2, \ldots, x_n$ and dependent variables $y_1, y_2, \ldots, y_m$.
  - *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form
    $$g_i(x_1, \ldots, x_n, y_1, \ldots, y_m) = 0$$
    for $i = 1, 2, \ldots, m$.
  - `vars`, a list $[x_1, \ldots, x_n, y_1, \ldots, y_m]$ of the independent and dependent variables entered as symbols in single list such that dependent variables come last.
  - `order=`k, where k is the order of the derivatives to be taken.
  - Optionally, a, a point where the partial derivatives should be evaluated at.
- `implicitdiff(`*expr*, *implicitdef*, *vars*, `order=`k $\langle$, a $\rangle$`)` returns all partial derivatives of order k. If $k = 1$ they are returned in a single list, which represents the gradient of `expr` with respect to independent variables. If $k = 2$ the corresponding Hessian matrix is returned (see Section 13.7.3, p. 310). If $k > 2$, a table with keys in form $[k_1, k_2, \ldots, k_n]$, where $\sum_{i=1}^{n} k_i = k$, is returned. Such a key corresponds to
  $$\frac{\partial^k f}{\partial var_1^{k_1}\, \partial var_2^{k_2} \cdots \partial var_n^{k_n}}.$$

Examples

```
> implicitdiff(x^2*y+y^2=1,y,x)
```

$$-\frac{2xy}{x^2+2y}$$

```
> implicitdiff([x^2+y=z,x+y*z=1],[y(x),z(x)],y,x)
```

$$\frac{-2xy-1}{y+z}$$

```
> implicitdiff(x*y,-2x^3+15x^2*y+11y^3-24y=0,y(x),x)
```

$$\frac{2x^3 - 5x^2y + 11y^3 - 8y}{5x^2 + 11y^2 - 8}$$

```
> f:=x*y*z:;
  g:=-2x^3+15x^2*y+11y^3-24y=0:;
  implicitdiff(f,g,[x,z,y],order=1)
```

$$\left[\frac{2x^3z - 5x^2yz + 11y^3z - 8yz}{5x^2 + 11y^2 - 8}, xy\right]$$

```
> implicitdiff(f,g,[x,z,y],order=2,[1,-1,0])
```

$$\begin{bmatrix} \frac{64}{9} & -\frac{2}{3} \\ -\frac{2}{3} & 0 \end{bmatrix}$$

In the following example, the value of $\frac{\partial^4 f}{\partial x^4}$ is computed at the point $(x = 0, y = 0, z)$.

```
> pd:=implicitdiff(f,g,[x,z,y],order=4,[0,z,0]):;
  pd[4,0]
```

$$-2z$$

13.2.4 Approximating derivatives of discrete functions

The `numdiff` command finds numerical approximations to derivatives.

- `numdiff` takes three mandatory arguments and one optional argument.
  - $X = [\alpha_0, \alpha_1, \ldots, \alpha_n]$, $Y = [\beta_0, \beta_1, \ldots, \beta_n]$, two lists of real numbers, where $n \geq 1$.
  - x_0, a real number.
  - Optionally, m, an integer or a sequence of integers (by default 1).

  Note that X, Y and $x0$ can also be symbolic expressions.
- `numdiff`$(X, Y, x_0 \langle, m\rangle)$ returns an approximation of the mth derivative of a function f at x_0, or a sequence of derivatives of order given by the sequence m, where f has values given by $f(\alpha_k) = \beta_k$, $k = 0, 1, \ldots, n$.
- `numdiff` uses Fornberg's algorithm with complexity $O(n^2 m)$ in both time and space.
- Note that $\alpha_0, \alpha_1, \ldots, \alpha_n$ do not have to be equally spaced, but they must be mutually different and input in ascending order. There are no restrictions on the choice of x_0.
- The algorithm performs reasonably fast on inexact input data, with rounding errors usually not damaging for $m \leq 4$. For higher m, consider providing input data in an exact form. This will make the algorithm run considerably slower, but will also avoid numerical instabilities due to the unlimited precision arithmetic being used.

In case you need only the first or second derivative at a sequence of points, you can use the `interp` command (see Section 17.1.2, p. 438) which uses cubic spline interpolation. Note that this method does not accept symbolic input.

Examples

Let $f(x) = \sin(x)\,\mathrm{e}^{-x}$, $x \in [0, 1]$. Sample this function at the points in

$$X = [0, 0.1, 0.2, 0.4, 0.5, 0.7, 0.8, 1]$$

to approximate $f''(1/\pi)$.

```
> f:=unapply(sin(x)*exp(-x),x):;
  X:=[0,0.1,0.2,0.4,0.5,0.7,0.8,1]:;
  Y:=apply(f,X):;
```

Now you can approximate the second derivative of f at the point $x_0 = \frac{1}{\pi}$.

```
> x0:=1/pi:;
  d:=numdiff(X,Y,x0,2)
```

$$-1.38167652799$$

Finally, compute the relative error of the obtained approximation.

```
> abs(d-f''(x0))/abs(f''(x0))*100
```

$$2.82975186496 \times 10^{-5}$$

The result is expressed in percentages.

Use a sequence of values for the parameter m to find a list of approximations of the respective derivatives at x_0. This is faster than calling `numdiff` to approximate one derivative at a time. For example, approximate the first, second and third derivative of the function

$$f(x) = 1 - \frac{1}{1 + x^2}, \quad x \in [0, 1],$$

at the point $x_0 = 0.57$ by sampling f at 21 equidistant points in the segment $[0, 1]$.

```
> f:=unapply(1-1/(1+x^2),x):;
  X:=[(0.05*k)$(k=0..20)]:;
  Y:=apply(f,X):;
  numdiff(X,Y,0.57,1,2,3)
```

$$[0.649439427528, 0.0217571104587, -2.99724196738]$$

Actually, $f'(x_0) = 0.649439427528$, $f''(x_0) = 0.0217571104558$ and $f'''(x_0) = -2.99724196764$.

`numdiff` can be used for generating custom finite-difference stencils for approximation of derivatives. For example, let $X = [-1, 0, 2, 4]$, $Y = [a, b, c, d]$ and $x_0 = 1$. To obtain an approximation formula for the second derivative:

```
> numdiff([-1,0,2,4],[a,b,c,d],1,2)
```

$$\frac{2}{5}a - \frac{b}{2} + \frac{d}{10}$$

The approximation is always a linear combination of elements in Y, regardless of X, x_0 and m.

Given the lists $X = [\alpha_0, \alpha_1, \ldots, \alpha_n]$ and $Y = [\beta_0, \beta_1, \ldots, \beta_n]$, the Lagrange polynomial passing through points (α_k, β_k) where $k = 0, 1, \ldots, n$ can be obtained by setting $m = 0$ and entering a symbol for x_0. Let $X = [-2, 0, 1]$ and $Y = [2, 4, 1]$:

```
> expand(numdiff([-2,0,1],[2,4,1],x,0))
```

$$-\frac{4}{3}x^2 - \frac{5}{3}x + 4$$

Note that by entering

```
> lagrange([-2,0,1],[2,4,1],x)
```

we obtain the same result.

13.3 Integration

13.3.1 Antiderivatives and definite integrals

The `int` and `integrate` commands compute a primitive or a definite integral. A difference between the two commands is that if you input `quest()` just after the evaluation of `integrate`, the answer is written with the $\int$ symbol.

`Int` is the inert form of *integrate*; namely, it evaluates to *integrate* for later evaluation.

- To find a primitive (an antiderivative), `int` (or `integrate`) takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally, x, the name of a variable (by default the value is `x`, so if the variable is `x` the second argument is unnecessary).
- `int(`*expr*$\langle, x\rangle$`)` or `integrate(`*expr*$\langle, x\rangle$`)` returns a primitive of *expr* with respect to x.
- To evaluate a definite integral, `int` (or `integrate`) takes four arguments:
  - *expr*, an expression.
  - x, the variable.
  - a and b, the bounds of the definite integral.
- `int(`*expr*$, x, a, b$`)` or `integrate(`*expr*$, x, a, b$`)` returns the exact value of the definite integral if the computation was successful or an unevaluated integral otherwise.

Examples

```
> integrate(x^2)
```

$$\frac{x^3}{3}$$

```
> integrate(t^2,t)
```

$$\frac{t^3}{3}$$

```
> integrate(x^2,x,1,2)
```

$$\frac{7}{3}$$

```
> integrate(1/(sin(x)+2),x,0,2*pi)
```

$$\frac{2}{3}\pi\sqrt{3}$$

`Int` is the inert form of `integrate`, it prevents evaluation, for example to avoid a symbolic computation that might not be successful if you just want a numeric evaluation, like for example:

```
> evalf(Int(exp(x^2),x,0,1))
```

or:

```
> evalf(int(exp(x^2),x,0,1))
```

$$1.46265174591$$

Let $f(x) = \frac{x}{x^2-1} + \ln\left(\frac{x+1}{x-1}\right)$. Find a primitive of f.

```
> int(x/(x^2-1)+ln((x+1)/(x-1)))
```

or:

```
> f(x):=x/(x^2-1)+ln((x+1)/(x-1)):;
  int(f(x))
```

$$x \ln\left(\frac{x+1}{x-1}\right) + \frac{2}{2} \ln\left|x^2-1\right| + \frac{\ln\left|x^2-1\right|}{2}$$

Compute $\int \frac{2}{x^6+2x^4+x^2}\,\mathrm{d}x$.

```
> int(2/(x^6+2*x^4+x^2))
```

$$2\left(\frac{-3x^2-2}{2\left(x^3+x\right)} - \frac{3}{2}\arctan x\right)$$

Compute $\int \frac{1}{\sin(x)+\sin(2x)}\,\mathrm{d}x$.

```
> integrate(1/(sin(x)+sin(2*x)))
```

$$2\left(\frac{\ln\left(\frac{1-\cos x}{1+\cos x}\right)}{12} - \frac{\ln\left|\frac{1-\cos x}{1+\cos x} - 3\right|}{3}\right)$$

13.3.2 Primitives and definite integrals

The Risch algorithm is a powerful algorithm for finding an elementary primitive of an elementary function or concluding that one does not exist. The `risch` command finds primitives and can use them to evaluate definite integrals.

- To find a primitive, `risch` takes one mandatory argument and one optional argument:
  - *expr*, an expression.
  - Optionally x, the name of a variable (by default the variable is `x`).
- `risch(`*expr*$\langle, x\rangle$`)` returns a primitive of *expr* with respect to x.
- To evaluate a definite integral, `risch` takes four arguments:
  - *expr*, an expression *expr*.
  - x, the variable.
  - a and b, the bounds of the definite integral.
- `int(`*expr*$, x, a, b$`)` returns the exact value of the definite integral if the computation was successful or an unevaluated integral otherwise.

Examples

```
> risch(x^2)
```

$$\frac{x^3}{3}$$

```
> risch(t^2,t)
```

$$\frac{t^3}{3}$$

```
> risch(exp(-x^2))
```

$$\int \mathrm{e}^{-x^2}\,\mathrm{d}x$$

meaning that e^{-x^2} has no primitive expressed with the elementary functions.

```
> risch(x^2,x,0,1)
```

$$\frac{1}{3}$$

13.3.3 Discrete summation

The `sum` command can evaluate sums, series, and find discrete antiderivatives. A discrete antiderivative of a sum $\sum_n f(n)$ is an expression G such that $G\big|_{x=n+1} - G\big|_{x=n} = f(n)$, which means that $\sum_{n=M}^{N} f(n) = G\big|_{x=N+1} - G\big|_{M}$.

- To evaluate a sum or series, `sum` takes four arguments:
  - *expr*, an expression.
  - k, the name of the variable.
  - n_0 and n_1, integers (the bounds of the sum).
- `sum(`*expr*$, k, n_0, n_1$`)` returns the sum $\sum_{k=n_0}^{n_1} expr$.
- To find a discrete antiderivative, `sum` takes two arguments:
  - *expr*, an expression.
  - x, the name of the variable.
- `sum(`*expr*$, x$`)` returns a discrete antiderivative.

Examples

```
> sum(1,k,-2,n)
```

$$n + 1 + 2$$

```
> normal(sum(2*k-1,k,1,n))
```

$$n^2$$

```
> sum(1/(n^2),n,1,10)
```

$$\frac{1968329}{1270080}$$

```
> sum(1/(n^2),n,1,+(infinity))
```

$$\frac{1}{6}\pi^2$$

```
> sum(1/(n^3-n),n,2,10)
```

$$\frac{27}{110}$$

```
> sum(1/(n^3-n),n,2,+(infinity))
```

$$\frac{1}{4}$$

This result comes from the decomposition of `1/(n^3-n)` (see Section 11.6.9, p. 251).

```
> partfrac(1/(n^3-n))
```

$$-\frac{1}{n}+\frac{1}{2\,(n-1)}+\frac{1}{2\,(n+1)}$$

Hence:

$$\sum_{n=2}^{N}-\frac{1}{n}=-\sum_{n=1}^{N-1}\frac{1}{n+1}=-\frac{1}{2}-\sum_{n=2}^{N-2}\frac{1}{n+1}-\frac{1}{N}$$

$$\frac{1}{2}\sum_{n=2}^{N}\frac{1}{n-1}=\frac{1}{2}\left(\sum_{n=0}^{N-2}\frac{1}{n+1}\right)=\frac{1}{2}\left(1+\frac{1}{2}+\sum_{n=2}^{N-2}\frac{1}{n+1}\right)$$

$$\frac{1}{2}\sum_{n=2}^{N}\frac{1}{n+1}=\frac{1}{2}\left(\sum_{n=2}^{N-2}\frac{1}{n+1}+\frac{1}{N}+\frac{1}{N+1}\right)$$

After simplification by $\sum_{n=2}^{N-2}$, it remains:

$$-\frac{1}{2}+\frac{1}{2}\left(1+\frac{1}{2}\right)-\frac{1}{N}+\frac{1}{2}\left(\frac{1}{N}+\frac{1}{N+1}\right)=\frac{1}{4}-\frac{1}{2N(N+1)}$$

Therefore:

- for $N=10$ the sum is equal to $1/4-1/220=27/110$,
- for $N=+\infty$ the sum is equal to $1/4$ because $\frac{1}{2N(N+1)}$ approaches zero when N approaches infinity.

```
> sum(1/(x*(x+1)),x)
```

$$-\frac{1}{x}$$

13.3.4 Riemann sums

Given a function f on $[0,1]$, the Riemann sum corresponding to dividing the interval into n equal parts and using the right endpoints is

$$\sum_{k=1}^{n}f\left(\frac{x}{n}\right)\frac{1}{n}.$$

The `sum_riemann` command determines if a sum is such a Riemann sum, and if it is, evaluates the integral.

- `sum_riemann` takes two arguments:
  - *expr*, an expression depending on two variables.
  - $[n,k]$, the list of those two variables.
- `sum_riemann(`*expr*`,[`n,k`])` returns

$$\lim_{n\to+\infty}\sum_{k=1}^{n}expr$$

(which, viewing the sum as a Riemann sum of a continuous function on $[0,1]$, is the definite integral) or returns `"it is probably not a Riemann sum"` when the no result is found.

Examples

Let $S_n = \sum_{k=1}^{n} \frac{k^2}{n^3}$. Compute the limit of $(S_n)_{n\in\mathbb{N}}$.

```
> sum_riemann(k^2/n^3,[n,k])
```

$$\frac{1}{3}$$

Let $S_n = \sum_{k=1}^{n} \frac{k^3}{n^4}$. Compute the limit of $(S_n)_{n\in\mathbb{N}}$.

```
> sum_riemann(k^3/n^4,[n,k])
```

$$\frac{1}{4}$$

Compute $\lim_{n\to+\infty} \left(\frac{1}{n+1} + \frac{1}{n+2} + \cdots + \frac{1}{2n}\right)$.

```
> sum_riemann(1/(n+k),[n,k])
```

$$\ln(2)$$

Let $S_n = \sum_{k=1}^{n} \frac{32n^3}{16n^4-k^4}$. Compute the limit of $(S_n)_{n\in\mathbb{N}}$.

```
> sum_riemann(32*n^3/(16*n^4-k^4),[n,k])
```

$$2\arctan\left(\frac{1}{2}\right) + \ln(3)$$

13.3.5 Integration by parts

Recall the integration by parts formula:

$$\int u(x)v'(x)\,\mathrm{d}x = u(x)v(x) - \int v(x)u'(x)\,\mathrm{d}x.$$

If you want to integrate a function $f(x)$ by parts, you need to specify how to write $f(x)$ as $u(x)v'(x)$, which you can do by either specifying $u(x)$ or $v(x)$. The result will be in the form $F(x) + \int g(x)\,\mathrm{d}x$, where $F(x) = u(x)v(x)$ and $g(x) = -v(x)u'(x)$.

In some cases, to finish an integral you need to integrate by parts more than once. After one integrating by parts once and getting $F(x) + \int g(x)\,\mathrm{d}x$, you may have to integrate $\int g(x)\,\mathrm{d}x$ by parts and add $F(x)$ to the result.

Xcas has two commands for integrating by parts: `ibpdv` (where you specify $v(x)$) and `ibpu` (where you specify $u(x)$), both of which return the result as a list $[F(x), g(x)]$. Both of these commands allow you to keep track of the function $F(x)$ you may need to add to the result of a subsequent integration by parts.

The `ibpdv` command finds the primitive of an expression written as $u(x)v'(x)$ by specifying $v(x)$.

- `ibpdv` takes two arguments:
  - *uvprime*, an expression which you can think of as $u(x)v'(x)$, or [*Fexpr*, *uvprime*], a list of two expressions, where again you can think of *uvprime* as $u(x)v'(x)$, and *Fexpr* represents the function $F(x)$ that you can add to the result of integrating by parts.
  - *vexpr*, an expression you can think of as $v(x)$. If *vexpr* is `0`, then instead of integrating by parts, the expression *uvprime* is integrated as a whole (this can be useful for finishing a multi-step integration by parts problem).
- `ibpdv(`*uvprime*, *vexpr*`)` or `ibpdv(`[*Fexpr*, *uvprime*], *vexpr*`)` returns

  - if $vexpr \neq 0$:

    $[u(x)v(x), -v(x)u'(x)]$ (or $[F(x) + u(x)v(x), -v(x)u'(x)]$ if the first argument is a list).
  - if $vexpr = 0$:

    $G(x)$ (or $F(x) + G(x)$, if the first argument is a list), where $G(x)$ is a primitive of *uvprime*.

Hence, `ibpdv` returns the terms computed in an integration by parts, with the possibility of doing several `ibpdv`s successively.

When the answer of

```
> ibpdv(u(x)*v'(x),v(x))
```

is computed, to obtain a primitive of $u(x)v'(x)$, it remains to compute the integral of the second term of this answer and then to sum this integral with the first term of this answer: to do this, just use `ibpdv` command with the answer as first argument and a new $v(x)$ (or 0 to terminate the integration) as second argument.

The `ibpu` command finds the primitive of an expression written as $u(x)v'(x)$ by specifying $u(x)$.

- `ibpu` takes two arguments:
  - *uvprime*, an expression which you can think of as $u(x)v'(x)$, or $[Fexpr, uvprime]$, a list of two expressions, where again you can think of *uvprime* as $u(x)v'(x)$, and *Fexpr* represents the function $F(x)$ that you can add to the result of integrating by parts.
  - *uexpr*, an expression you can think of as $u(x)$. If *uexpr* is `0`, then instead of integrating by parts, the expression *uvprime* is integrated as a whole (this can be useful for finishing a multi-step integration by parts problem).
- `ibpu(`*uvprime*, *uexpr*`)` or `ibpu(`$[Fexpr, uvprime]$, *uexpr*`)` returns
  - if $uexpr \neq 0$:

    $[u(x)v(x), -v(x)u'(x)]$ (or $[F(x) + u(x)v(x), -v(x)u'(x)]$ if the first argument is a list).
  - if $uexpr = 0$:

    $G(x)$ (or $F(x) + G(x)$, if the first argument is a list), where $G(x)$ is a primitive of *uvprime*.

Hence, `ibpu` returns the terms computed in an integration by parts, with the possibility of doing several `ibpu`s successively.

When the answer of

```
> ibpu(u(x)*v'(x),u(x))
```

is computed, to obtain a primitive of $u(x)v'(x)$, it remains to compute the integral of the second term of this answer and then to sum this integral with the first term of this answer: to do this, just use the `ibpu` command with the answer as first argument and a new $u(x)$ (or 0 to terminate the integration) as second argument.

Example

```
> ibpdv(ln(x),x)
```

$$[x \ln x, -1]$$

then:

```
> ibpdv([x*ln(x),-1],0)
```

or:

```
> ibpdv(ans(),0)
```

$$-x + x \ln x$$

When the first argument of `ibpdv` is a list of two elements, `ibpdv` works only on the last element of this list and adds the integrated term to the first element of this list. (therefore it is possible to do several `ibpdv`s successively).

For example, to evaluate $\int \ln(x)^2\,dx$, input:

```
> ibpdv((ln(x))^2,x)
```

$$\left[x\ln^2 x, -2\ln x\right]$$

It remains to integrate $-2\ln x$:

```
> ibpdv([x*(ln(x))^2,-2*log(x)],x)
```

or:

```
> ibpdv(ans(),x)
```

$$\left[x\ln^2 x - 2x\ln x, 2\right]$$

And now it remains to integrate 2:

```
> ibpdv([x*(ln(x))^2+x*(-2*log(x)),2],0)
```

or:

```
> ibpdv(ans(),0)
```

$$x\ln^2 x - 2x\ln x + 2x$$

```
> ibpu(ln(x),ln(x))
```

$$[x\ln x, -1]$$

then:

```
> ibpu([x*ln(x),-1],0)
```

or:

```
> ibpu(ans(),0)
```

$$-x + x\ln x$$

When the first argument of `ibpu` is a list of two elements, `ibpu` works only on the last element of this list and adds the integrated term to the first element of this list. Therefore it is possible to do several `ibpu`s successively, similarly to how you can do several `ibpdv`s successively.

For example, to evaluate $\int \ln(x)^2\,dx$, input:

```
> ibpu((ln(x))^2,(ln(x))^2)
```

$$\left[x\ln^2 x, -2\ln x\right]$$

It remains to integrate $-2\ln x$:

```
> ibpu([x*(ln(x))^2,-2*ln(x)],ln(x))
```

or:

```
> ibpu(ans(),ln(x))
```

$$\left[x\ln^2 x - 2x\ln x, 2\right]$$

Finally, it remains to integrate 2:

```
> ibpu([x*(ln(x))^2+x*(-2*ln(x)),2],0)
```

or:

```
> ibpu(ans(),0)
```

$$x\ln^2 x - 2x\ln x + 2x$$

13.3.6 Change of variables

See the `subst` command in Section 9.1.13, p. 175.

13.3.7 Integrals and limits

The `limit` command (see Section 13.1.1, p. 276) can compute limits involving integrals.

Examples

Find the limit of $\int_2^a x^{-2}\,\mathrm{d}x$ as $a \to +\infty$.

Input (if `a` is assigned, first input `purge(a)`):

```
> limit(integrate(1/(x^2),x,2,a),a,+(infinity))
```

$$\frac{1}{2}$$

Indeed, since $\int_2^a x^{-2}\,\mathrm{d}x = \frac{1}{2} - \frac{1}{a}$, the integral $\int_2^a x^{-2}\,\mathrm{d}x$ tends to $\frac{1}{2}$ as a goes to infinity.

Find the limit of $\int_2^a \left(\frac{x}{x^2-1} + \ln\left(\frac{x+1}{x-1}\right)\right)\mathrm{d}x$ as $a \to +\infty$. (If `a` is assigned, first input `purge(a)`.)

```
> limit(integrate(x/(x^2-1)+log((x+1)/(x-1)),x,2,a),a,+infinity)
```

$$+\infty$$

Indeed, since $\int_2^a x/(x^2-1)\,\mathrm{d}x = (1/2)(\ln(a^2-1) - \ln(3))$ and $\int_2^a \ln((x+1)/(x-1))\,\mathrm{d}x = \ln(a+1) + \ln(a-1) + a\ln((a+1)/(a-1)) - 3\ln(3)$, the integral $\int_2^a x/(x^2-1) + \ln((x+1)/(x-1))\,\mathrm{d}x$ goes to infinity as a goes to infinity.

For an example when the integral cannot be simply evaluated, find the limit of $\int_a^{3a} \frac{\cos(x)}{x}\,\mathrm{d}x$ as $a \to 0$.

```
> limit(int(cos(x)/x,x,a,3a),a,0)
```

$$\ln(3)$$

To find this limit yourself, you cannote that $1 - x^2/2 \le \cos(x) \le 1$, and so $1/x - x/2 \le \cos(x)/x \le 1/x$, and so $\int_a^{3a}(1/x - x/2)\,\mathrm{d}x \le \int_a^{3a}\cos(x)/x\,\mathrm{d}x \le \int_a^{3a} 1/x\,\mathrm{d}x$, which gives you $\ln(3) - 2a^2 \le \int_a^{3a}\cos(x)/x\,\mathrm{d}x \le \ln(3)$, and so as a approaches 0, $\int_a^{3a}\cos(x)/x\,\mathrm{d}x$ will approach $\ln(3)$.

13.3.8 Length of an arc

The `arcLen` command finds the lengths of curves in the plane, which can either be given by an equation or a curve object.

- To find the length of a curve given by an equation, `arcLen` takes four arguments:
  - *expr*, an expression (resp. a list of two expressions [*expr*$_1$,`expr`$_2$]) involving a variable x.
  - x, the name of the variable.
  - a and b, two values for the bounds of this variable.

- `arcLen(`*expr*, *x*, *a*, *b*`)` resp. `arcLen([`*expr*$_1$,*expr*$_2$`],`*x*, *a*, *b*`)` returns the length of the curve defined by $y = f(x) =$ *expr* resp. by $x_1 =$ *expr*$_1$, $x_2 =$ *expr*$_2$ as x varies from a to b, using the formula

$$\operatorname{arcLen}(f(x), x, a, b) = \int_a^b \sqrt{f'(x)^2 + 1}\,\mathrm{d}x$$

or

$$\operatorname{arcLen}(f(x), x, a, b) = \int_a^b \sqrt{x'(t)^2 + y'(t)^2}\,\mathrm{d}t.$$

- To find the length of a curve given by a curve object, `arcLen` takes a single argument: *curve*, a geometric curve defined in one of the graphics chapters (chapters 26 and 27).
- `arcLen(`*curve*`)` returns the length of the curve.

Examples

Compute the length of the parabola $y = x^2$ from $x = 0$ to $x = 1$:

```
> arcLen(x^2,x,0,1)
```

or:

```
> arcLen([t,t^2],t,0,1)
```

$$\frac{2\sqrt{5} - \ln\left(\sqrt{5} - 2\right)}{4}$$

Compute the length of the curve $y = \cosh(x)$ from $x = 0$ to $x = \ln(2)$:

```
> arcLen(cosh(x),x,0,log(2))
```

$$\frac{3}{4}$$

Compute the length of the circle $x = \cos(t), y = \sin(t)$ from $t = 0$ to $t = 2\pi$:

```
> arcLen([cos(t),sin(t)],t,0,2*pi)
```

$$2\pi$$

Compute the length of the unit circle segment in the first quadrant:

```
> arcLen(circle(0,1,0,pi/2))
```

$$\frac{1}{2}\pi$$

Compute the length of an arc:

```
> arcLen(arc(0,1,pi/2))
```

$$\frac{1}{4}\pi\sqrt{2}$$

13.4 Differential equations

This section is limited to symbolic (or exact) solutions of differential equations. For numeric solutions of differential equations, see `odesolve` (Section 23.4.1, p. 637). For graphic representation of solutions of differential equations, see `plotfield` (Section 19.9.1, p. 488), `plotode` (Section 19.8.3, p. 486) and `interactive_plotode` (Section 19.9.2, p. 489).

13.4.1 Solving differential equations

The `desolve` (or `dsolve` or `deSolve`) command can solve:

- linear differential equations with constant coefficients,
- first order linear differential equations,
- first order differential equations without y,
- first order differential equations without x,
- first order differential equations with separable variables,
- first order homogeneous differential equations: $y' = F(y/x)$,
- first order differential equations with integrating factor,
- first order Bernoulli differential equations: $a(x)y' + b(x)y = c(x)y^n$,
- first order Clairaut differential equations: $y = xy' + f(y')$.
- `desolve` takes one mandatory arguments and two optional arguments:
  - *de*, a differential equation or list of differential equations, including any initial conditions.
  - Optionally, x, the variable (by default `x`).
  - Optionally, y, the unknown function (by default `y`). The unknown function can be given in variable form (such as `y`) or function form (such as `y(x)`), in which case the variable does not have to be given as a separate argument.
- `desolve(`*de*$\langle, x, y\rangle$`)` returns the solution of the differential equation.

In the differential equations, the function y can be denoted by y or $y(x)$, the derivative by y', $y'(x)$ or `diff(`$y(x), x$`)`, etc.

Examples

Solve $y'' + 2y' + y = 0$.

```
> desolve(y''+2*y'+y,y)
```

$$\mathrm{e}^{-x}\,(c_0 x + c_1)$$

Find the solution which satisfies the initial conditions $y(0) = 1$ and $y'(0) = 0$:

```
> desolve([y''+2*y'+y,y(0)=1,y'(0)=0],y)
```

or:

```
> desolve(y''+2*y'+y and y(0)=1 and y'(0)=0,y)
```

$$\mathrm{e}^{-x}\,(x + 1)$$

With t as the independent variable:

```
> desolve(diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(t))
```

or:

```
> desolve(diff(y(t),t$2)+2*diff(y(t),t)+y(t),t,y)
```

$$\mathrm{e}^{-t}\,(c_0 t + c_1)$$

With the initial conditions:

```
> desolve([diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(0)=1,y'(0)=0],y(t))
```

or:

```
> desolve([diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(0)=1,y'(0)=0],t,y)
```

$$e^{-t}\,(t+1)$$

Solve $y''+y=\cos(x)$.

Input (typing twice prime for `y''`):

```
> desolve(y''+y=cos(x),y)
```

or:

```
> desolve((diff(diff(y))+y)=(cos(x)),y)
```

$$c_0\cos x + c_1\sin x + \frac{2x\sin x+\cos x}{4}$$

c_0, c_1 are the constants of integration: $y(0)=c_0$ $y'(0)=c_1$.

If the variable is not `x` but `t`:

```
> desolve(derive(derive(y(t),t),t)+y(t)=cos(t),t,y)
```

$$c_0\cos t + c_1\sin t + \frac{2t\sin t+\cos t}{4}$$

c_0, c_1 are the constants of integration: $y(0)=c_0$ and $y'(0)=c_1$.

Solve $y''+y=\cos(x)$, $y(0)=1$.

```
> desolve([y''+y=cos(x),y(0)=1],y)
```

$$\frac{3}{4}\cos x + c_1\sin x + \frac{2x\sin x+\cos x}{4}$$

Solve $y''+y=\cos(x)$, $y(0)^2=1$.

```
> desolve([y''+y=cos(x),y(0)^2=1],y)
```

$$\left[\frac{3\cos x}{4} + c_1\sin x + \frac{2x\sin x+\cos x}{4}, -\frac{5\cos x}{4} + c_1\sin x + \frac{2x\sin x+\cos x}{4}\right]$$

each component of this list is a solution, you have two solutions depending on the constant c_1 ($y'(0)=c_1$) and corresponding to $y(0)=1$ and to $y(0)=-1$.

Solve $y''+y=\cos(x)$, $y(0)^2=1$, $y'(0)=1$.

```
> desolve([y''+y=cos(x),y(0)^2=1,y'(0)=1],y)
```

$$\left[\frac{3\cos x}{4} + \sin x + \frac{2x\sin x+\cos x}{4}, -\frac{5\cos x}{4} + \sin x + \frac{2x\sin x+\cos x}{4}\right]$$

each component of this list is a solution (you have two solutions).

Solve $y''+2y'+y=0$.

```
> desolve(y''+2*y'+y=0,y)
```

$$e^{-x}\,(c_0x+c_1)$$

the solution depends on two constants of integration: c_0 and c_1 ($y(0)=c_0$ and $y'(0)=c_1$).

Solve $y''-6y'+9y=xe^{3x}$.

```
> desolve(y''-6*y'+9*y=x*exp(3*x),y)
```

$$e^{3x}\,(c_0x+c_1) + \frac{1}{6}x^3e^{3x}$$

The solution depends on 2 constants of integration: c_0, c_1 ($y(0)=c_0$ and $y'(0)=c_1$).

Examples of first order linear equations

Solve $xy' + y - 3x^2 = 0$.

```
> desolve(x*y'+y-3*x^2,y)
```

$$\frac{c_0 + x^3}{x}$$

Solve $y' + xy = 0$, $y(0) = 1$.

```
> desolve([y'+x*y=0, y(0)=1]),y)
```

or:

```
> desolve((y'+x*y=0) && (y(0)=1),y)
```

$$e^{-\frac{x^2}{2}}$$

Solve $x(x^2 - 1)y' + 2y = 0$.

```
> desolve(x*(x^2-1)*y'+2*y=0,y)
```

$$\frac{c_0 x^2}{x^2 - 1}$$

Solve $x(x^2 - 1)y' + 2y = x^2$.

```
> desolve(x*(x^2-1)*y'+2*y=x^2,y)
```

$$\frac{c_0 x^2 + x^2 \ln x}{x^2 - 1}$$

If the variable is t instead of x, for example, solve:

$$t(t^2 - 1)\, y'(t) + 2y(t) = t^2.$$

```
> desolve(t*(t^2-1)*diff(y(t),t)+2*y(t)=(t^2),y(t))
```

$$\frac{c_0 t^2 + t^2 \ln t}{t^2 - 1}$$

Solve $x(x^2 - 1)\, y' + 2y = x^2$, $y(2) = 0$.

```
> desolve([x*(x^2-1)*y+2*y=x^2,y(2)=0],y)
```

$$\frac{-\ln(2)\, x^2 + x^2 \ln x}{x^2 - 1}$$

Solve $\sqrt{1 + x^2}\, y' - x - y = \sqrt{1 + x^2}$.

```
> desolve(y'*sqrt(1+x^2)-x-y-sqrt(1+x^2),y)
```

$$\frac{-c_0 + \ln\left(\sqrt{x^2 + 1} - x\right)}{x - \sqrt{x^2 + 1}}$$

Examples of first differential equations with separable variables

Solve $y' = 2\sqrt{y}$.

```
> desolve(y'=2*sqrt(y),y)
```

$$\left[\left(-\frac{1}{2}c_0 + x\right)^2\right]$$

Solve $xy'\ln(x) - y(3\ln(x)+1) = 0$.

```
> desolve(x*y'*ln(x)-(3*ln(x)+1)*y,y)
```

$$c_0 x^3 \ln x$$

Examples of Bernoulli differential equations $a(x)y' + b(x)y = c(x)y^n$ where n is a real constant

The method used is to divide the equation by y^n, so that it becomes a first order linear differential equation in $u = y^{1-n}$.

Solve $x\,y' + 2y + x\,y^2 = 0$.

```
> desolve(x*y'+2*y+x*y^2,y)
```

$$\left[0, -\frac{1}{c_1 x^2 + x}\right]$$

Solve $x\,y' - 2y = x\,y^3$.

```
> desolve(x*y'-2*y-x*y^3,y)
```

$$\left[\left(\left(-\frac{1}{5}\cdot 2x^5 + c_0\right)\mathrm{e}^{-4\ln x}\right)^{-\frac{1}{2}}, -\left(\left(-\frac{1}{5}\cdot 2x^5 + c_0\right)\mathrm{e}^{-4\ln x}\right)^{-\frac{1}{2}}\right]$$

Solve $x^2\,y' - 2y = x\mathrm{e}^{(4/x)}\,y^3$.

```
> desolve(x*y'-2*y-x*exp(4/x)*y^3,y)
```

$$\left[\left(\left(-\int 2x^4\left(\mathrm{e}^{\frac{1}{x}}\right)^4 \mathrm{d}x + c_0\right)\mathrm{e}^{-4\ln x}\right)^{-\frac{1}{2}}, -\left(\left(-\int 2x^4\left(\mathrm{e}^{\frac{1}{x}}\right)^4 \mathrm{d}x + c_0\right)\mathrm{e}^{-4\ln x}\right)^{-\frac{1}{2}}\right]$$

Examples of first order homogeneous differential equations $y' = F(y/x)$

The method of integration is to search for $t = y/x$ instead of y). Solve $3x^3\,y' = y\,(3x^2 - y^2)$.

```
> desolve(3*x^3*diff(y)=((3*x^2-y^2)*y),y)
```

$$\left[0, -\frac{x\sqrt{6}\sqrt{\ln\left(\frac{x}{c_0}\right)}}{2\ln\left(\frac{x}{c_0}\right)}, \frac{x\sqrt{6}\sqrt{\ln\left(\frac{x}{c_0}\right)}}{2\ln\left(\frac{x}{c_0}\right)}\right]$$

Hence the solutions are $y = 0$ and the familiy of curves with parametric equations $x = c_0\,\mathrm{e}^{3/(2t^2)}, y = t\,c_0\,\mathrm{e}^{3/(2t^2)}$ (the parameter is denoted by `t` in the answer).

Examples of first order differential equations with an integrating factor

By multiplying the equation by a function of x, y it becomes a closed differential form.

Solve $y\,y' + x = 0$.

```
> desolve(y*y'+x,y)
```

$$\left[\sqrt{-x^2 - 2c_0}, -\sqrt{-x^2 - 2c_0}\right]$$

In this example, $x\,\mathrm{d}x + y\,\mathrm{d}y$ is closed, the integrating factor was 1.

Solve $2x\,y\,y' + x^2 - y^2 + a^2 = 0$.

```
> desolve(2*x*y*y'+x^2-y^2+a^2,y)
```

$$\left[\sqrt{a^2 - x^2 - c_1 x}, -\sqrt{a^2 - x^2 - c_1 x}\right]$$

In this example, the integrating factor was $1/x^2$.

Example of first order differential equations without x

Solve $(y+y')^4+y'+3y=0$.

This kind of equation cannot be solved directly by XCAS, you can use the following steps on solve it with the help of XCAS. The idea is to find a parametric representation of $F(u,v)=0$ where the equation is $F(y,y')=0$. Let $u=f(t), v=g(t)$ be such a parametrization of $F=0$, then $y=f(t)$ and $\mathrm{d}y/\mathrm{d}x=y'=g(t)$. Hence

$$\mathrm{d}y/\mathrm{d}t=f'(t)=y'\,\mathrm{d}x/\mathrm{d}t=g(t)\,\mathrm{d}x/\mathrm{d}t.$$

The solution is the curve of parametric equations $x(t), y(t)=f(t)$, where $x(t)$ is solution of the differential equation $g(t)\,\mathrm{d}x=f'(t)dt$.

Back to the example, you can let $y+y'=t$, hence:

$$y=-t-8t^4,\quad y'=\mathrm{d}y/\mathrm{d}x=3t+8t^4,\quad \mathrm{d}y/\mathrm{d}t=-1-32t^3.$$

Therefore

$$(3t+8t^4)\,\mathrm{d}x=(-1-32t^3)\,\mathrm{d}t.$$

```
> desolve((3*t+8*t^4)*diff(x(t),t)=(-1-32*t^3),x(t))
```

$$\frac{9c_0-11\ln\left(8t^3+3\right)-\ln\left(t^3\right)}{9}$$

The solution is the curve of parametric equation:

$$x(t)=\frac{9c_0-11\ln\left(8t^3+3\right)-\ln\left(t^3\right)}{9},\quad y(t)=-t-8t^4.$$

Examples of first order Clairaut differential equations $y=x\,y'+f(y')$

The solutions are the lines D_m of equation $y=mx+f(m)$ where m is a real constant.

Solve $x\,y'+(y')^3-y=0$.

```
> desolve(x*y'+y'^3-y,y)
```

$$\left[c_0x+c_0^3\right]$$

Solve $y-x\,y'-\sqrt{a^2+b^2\,y'^2}=0$.

```
> desolve((y-x*y'-sqrt(a^2+b^2*y'^2),y)
```

$$\left[c_0x+\sqrt{a^2+b^2c_0^2}\right]$$

13.4.2 Laplace transform

Denoting by $\mathcal{L}$ the Laplace transform, you get the following:

$$\mathcal{L}(y)(x)=\int_0^{+\infty}\mathrm{e}^{-xu}y(u)du$$
$$\mathcal{L}^{-1}(g)(x)=\frac{1}{2\mathrm{i}\pi}\int_C\mathrm{e}^{zx}g(z)\,\mathrm{d}z$$

where C is a closed contour enclosing the poles of `g`.

The `laplace` command finds the Laplace transform of a function.

- `laplace` takes one mandatory argument and two optional arguments:
  - *expr*, an expression involving a variable.

  - Optionally, x, the variable name (by default `x`).
  - Optionally, s, a variable for the output (by default x).

- `laplace(`*expr* $\langle$`,` *x* $\rangle$`)` returns the Laplace transform of *expr*.

The `ilaplace` or `invlaplace` command finds the inverse Laplace transform of a function.

- `ilaplace` takes one mandatory argument and two optional arguments:
  - *expr*, an expression involving a variable.
  - Optionally, x, the variable name (by default `x`).
  - Optionally, s, a variable for the output (by default x).
- `ilaplace(`*expr* $\langle$`,` *x* $\rangle$`)` returns the inverse Laplace transform of *expr*.

You also can use the `addtable` command Laplacians of unspecified functions (see Section 21.4.3, p. 584). The Laplace transform has the following properties:

$$\begin{aligned}\mathcal{L}(y')(x) &= -y(0) + x\mathcal{L}(y)(x)\\ \mathcal{L}(y'')(x) &= -y'(0) + x\mathcal{L}(y')(x)\\ &= -y'(0) - xy(0) + x^2\mathcal{L}(y)(x)\end{aligned}$$

These properties make the Laplace transform and inverse Laplace transform useful for solving linear differential equations with constant coefficients. For example, suppose you have

$$\begin{aligned}y'' + py' + qy &= f(x)\\ y(0) = a,\ y'(0) &= b\end{aligned}$$

then

$$\begin{aligned}\mathcal{L}(f)(x) &= \mathcal{L}(y'' + py' + qy)(x)\\ &= -y'(0) - xy(0) + x^2\mathcal{L}(y)(x) - py(0) + px\mathcal{L}(y)(x)) + q\mathcal{L}(y)(x)\\ &= (x^2 + px + q)\mathcal{L}(y)(x) - y'(0) - (x + p)y(0)\end{aligned}$$

Therefore, if $a = y(0)$ and $b = y'(0)$, you get

$$\mathcal{L}(f)(x) = (x^2 + px + q)\mathcal{L}(y)(x) - (x + p)a - b$$

and the solution of the differential equation is:

$$y(x) = \mathcal{L}^{-1}\left(\frac{\mathcal{L}(f)(x) + (x + p)a + b}{x^2 + px + q}\right).$$

Examples

```
> laplace(sin(x))
```

$$\frac{1}{x^2 + 1}$$

With t as the original variable:

```
> laplace(sin(t),t)
```

$$\frac{1}{t^2 + 1}$$

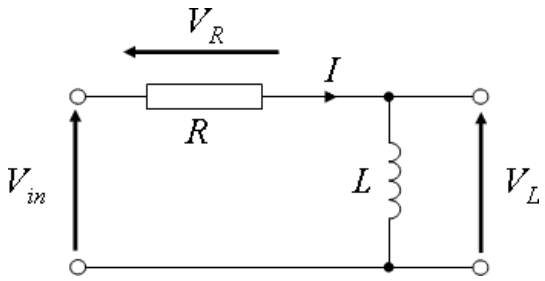


Figure 13.1: RL circuit (source)

With t as the original variable and s as the transform variable:

```
> L:=laplace(sin(t),t,s)
```

$$\frac{1}{s^2+1}$$

Apply the inverse transform to obtain the original function:

```
> ilaplace(L,s,t)
```

$$\sin t$$

Solve $y''-6y'+9y=xe^{3x}$, $y(0)=c_0$, $y'(0)=c_1$. Here, $p=-6$ and $q=9$.

```
> laplace(x*exp(3*x))
```

$$\frac{1}{x^2-6x+9}$$

```
> ilaplace((1/(x^2-6*x+9)+(x-6)*c_0+c_1)/(x^2-6*x+9))
```

$$\frac{1}{6}\left(x^3-18xc_0+6xc_1+6c_0\right)\mathrm{e}^{3x}$$

Note that this equation could be solved directly:

```
> desolve(y''-6*y'+9*y=x*exp(3*x),y)
```

$$\mathrm{e}^{3x}\left(c_0x+c_1\right)+\frac{1}{6}x^3\mathrm{e}^{3x}$$

Applications

Laplace transform is useful in analysis of eletric circuits. A simple RL circuit is shown in Figure 13.1. Given the input voltage V_{in}, the problem is to find the current $I(t)$. From Kirchhoff laws:

$$V_L+V_R=LI'(t)+RI(t)=V_{in}.$$

By transforming both sides of the above equation and assuming $I(0)=0$, you get

$$Ls\mathcal{L}(I)(s)+R\mathcal{L}(I)(s)=\mathcal{L}(V_{in})(s),$$

from which follows

$$I(t)=\mathcal{L}^{-1}\left(\frac{\mathcal{L}(V_{in})(s)}{Ls+R}\right).$$

Let $V_{in}(t)=V_0\sin(\omega t)$. You find $I(t)$ by entering:

```
> assume(omega>0):;
  I:=ilaplace(laplace(V0*sin(omega*t),t,s)/(L*s+R),s,t):;
  factor(I)
```

$$\frac{\left(-\omega L\cos\left(\omega t\right)+\omega L\mathrm{e}^{-\frac{Rt}{L}}+R\sin\left(\omega t\right)\right)V_0}{R^2+L^2\omega^2}$$

Let now V_{in} be a single rectangular pulse $V_{in}=\begin{cases}V_0, & 0\le t\le 1,\\ 0, & t>1.\end{cases}$. It is easily defined by using the `boxcar` command (see Section 21.1.1, p. 567):

```
> Vin:=V0*boxcar(0,1,t):;
  I:=ilaplace(laplace(Vin,t,s)/(L*s+R),s,t):;
  normal(piecewise(I))
```

$$\begin{cases}\frac{-V_0\mathrm{e}^{-\frac{Rt}{L}}+V_0}{R}, & 1>t\\ \frac{-V_0\mathrm{e}^{-\frac{Rt}{L}}+V_0\mathrm{e}^{-\frac{Rt-R}{L}}}{R}, & \text{otherwise}\end{cases}$$

Laplace transform can be used with arbitrary periodic functions. For example, Let $V_{in}=V_0\left(1-(t-1)^2\right)$ for $t\in[0,2]$ define a periodic function with period $T=2$.

```
> Vin:=V0*periodic(1-(t-1)^2,t=0..2):;
  I:=ilaplace(laplace(Vin,t,s)/(L*s+R),s,t)
```

$$\frac{\left(2L\left(R+L\right)\mathrm{e}^{-\frac{R\left(t-2\left\lfloor\frac{t}{2}\right\rfloor\right)}{L}}+4R^2\left\lfloor\frac{t}{2}\right\rfloor\left(t-\left\lfloor\frac{t}{2}\right\rfloor-1\right)-t\left(t-2\right)R^2-4R\left\lfloor\frac{t}{2}\right\rfloor L+2L\left(Rt-R-L\right)\right)V_0}{R^3}$$

Subexpressions in the above result are factored manually for a more readable presentation. Xcas computes this result by using symbolic periodic summation.

Applying the above technique to more complex circuits involving resistors, capacitors and inductors is straightforward; contour analysis by Kirchhoff laws produces a system of linear differential equations which is transformed to a system of linear equations by Laplace transform. From there, it is easy to find transforms of contour currents.

13.4.3 Solving linear homogeneous second-order ODE with rational coefficients

The `kovacicsols` command finds Louivillian solutions of ordinary linear homogeneous second-order differential equations of the form

$$a\,y''+b\,y'+c\,y=0, \tag{13.1}$$

where a, b and c are rational functions of the independent variable. `kovacicsols` uses Kovacic's algorithm.

- `kovacicsols` takes one mandatory argument and two optional arguments:
  - *kode*, an equality of the form of equation (13.1), an expression for the left-hand side, or a list of the coefficients $[a,b,c]$.
  - Optionally, x the independent variable (by default `x`).
  - Optionally, y, the dependent variable (by default `y`). This option should not be used if the first argument is a list of coefficients.
- `kovacicsols(`*kode*$\langle,x,y\rangle$`)` returns a Liouvillian solution of equation (13.1). This can be a list or an expression. An empty list means that there are no Liouvillian solutions to the equation. A

non-empty list will contain one or two independent solutions to the differential equation. If the list contains two solutions y_1 and y_2, the general solution will be

$$y = C_1\, y_1 + C_2\, y_2$$

where $C_1, C_2 \in \mathbb{R}$ are arbitrary constants. However, for some equations only one solution y_1 is returned, in which case the other solution can be obtained as (using reduction of order):

$$y_2 = y_1 \int y_1^{-2}. \tag{13.2}$$

If `kovacicsols` returns an expression, it will give the solution of the differential equation implicitly. In that case the return value is a polynomial P of order $n \in \{4, 6, 12\}$ in the variable `omega_` (denoted here by ω) with rational coefficients r_k, $k = 0, 1, 2, \ldots, n$. If $P(\omega_0) = 0$ for some ω_0, then $y = \exp\left(\int \omega_0\right)$ is a solution to the differential equation.

Examples

Find the general solution to $y'' = \left(\frac{1}{x} - \frac{3}{16\,x^2}\right) y$.

```
> kovacicsols(y''=y*(1/x-3/16x^2))
```

$$\left[x^{\frac{1}{4}} \mathrm{e}^{2\sqrt{x}},\, x^{\frac{1}{4}} \mathrm{e}^{-2\sqrt{x}}\right]$$

Therefore, the general solution is $y = C_1\, x^{1/4}\, \mathrm{e}^{2\,\sqrt{x}} + C_2\, x^{1/4}\, \mathrm{e}^{-2\,\sqrt{x}}$.

Solve $x'(t) + \frac{3\,(t^2-t+1)}{16\,(t-1)^2\,t^2}\, x(t) = 0$.

```
> kovacicsols(x''+3*(t^2-t+1)/(16*(t-1)^2*t^2)*x,t,x)
```

$$\left[\left(-t\,(t-1)\left(2\sqrt{t^2-t}+2t-1\right)\right)^{\frac{1}{4}},\, \left(t\,(t-1)\left(2\sqrt{t^2-t}-2t+1\right)\right)^{\frac{1}{4}}\right]$$

so the general solution is, for $C_1, C_2 \in \mathbb{R}$,

$$x(t) = C_1 \sqrt[4]{t\,(t-1)\,(1-2\,t-2\,\sqrt{t^2-t})} + C_2 \sqrt[4]{t\,(t-1)\,(1-2\,t+2\,\sqrt{t^2-t})}.$$

Find a particular solution to $y'' = \frac{4\,x^6-8\,x^5+12\,x^4+4\,x^3+7\,x^2-20\,x+4}{4\,x^4}\, y$.

```
> r:=(4x^6-8x^5+12x^4+4x^3+7x^2-20x+4)/(4x^4);
  kovacicsols(y''=r*y)
```

$$\left[\frac{(-1+x^2)\,\mathrm{e}^{\frac{\frac{1}{2}\left(x^3-2x^2-2\right)}{x}}}{x\sqrt{x}}\right]$$

Hence $y = (x^2-1)\,x^{-3/2}\,\mathrm{e}^{\frac{x^3-2\,x^2-2}{2\,x}}$ is a solution to the given equation.

Solve $y'' + y' = \frac{6\,y}{x^2}$.

```
> kovacicsols(y''+y'=6y/x^2)
```

$$\left[\frac{(12+6x+x^2)\,\mathrm{e}^{-x}}{x^2}\right]$$

Solve the Titchmarsh equation $y'' + (19 - x^2)\, y = 0$.

```
> kovacicsols(y''+(19-x^2)*y=0,x,y)
```

$$\left[\left(\frac{945}{16}x - \frac{315}{2}x^3 + \frac{189}{2}x^5 - 18x^7 + x^9\right)\mathrm{e}^{-\frac{x^2}{2}}\right]$$

This is only a single, particular solution.

Find the general solution of Halm's equation $(1+x^2)^2\,y''(x)+3\,y(x)=0$.

```
> sol:=kovacicsols((1+x^2)^2*y''+3y=0,x,y)
```

$$\left[\frac{-1+x^2}{\sqrt{x^2+1}}\right]$$

The other basic solution is obtained by using (13.2).

```
> y1:=sol[0]; y2:=normal(y1*int(y1^-2,x))
```

$$\frac{-1+x^2}{\sqrt{x^2+1}},\ -\frac{x\sqrt{x^2+1}}{x^2+1}$$

Therefore, $y=C_1\,\frac{x^2-1}{\sqrt{x^2+1}}+C_2\,\frac{x}{\sqrt{x^2+1}}$, where $C_1,C_2\in\mathbb{R}$.

Find the general solution of the non-homogeneous equation $y''-\frac{27\,y}{36\,(x-1)^2}=x+4$. First you need to find the general solution to the corresponding homogeneous equation $y_h''-\frac{27\,y_h}{36\,(x-1)^2}=0$.

```
> sols:=kovacicsols(y''-y*27/(36*(x-1)^2),x,y)
```

$$\left[\frac{-2x+x^2}{\sqrt{x-1}}\right]$$

Call this solution y_1 and find the other basic independent solution by using (13.2).

```
> y1:=sols[0]:;
  y2:=y1*int(1/y1^2,x)
```

$$-\frac{\sqrt{x-1}}{2x-2}$$

So the general solution of the homogeneous equation is

$$y_h=C_1\,y_1+C_2\,y_2=\frac{C_1\,(x^2-2\,x)+C_2}{\sqrt{x-1}},\quad C_1,C_2\in\mathbb{R}.$$

A particular solution y_p of the non-homogeneous equation can be obtained by variation of parameters:

$$y_p=-y_1\int\frac{y_2\,f(x)}{W}\,\mathrm{d}x+y_2\int\frac{y_1\,f(x)}{W}\,\mathrm{d}x,$$

where $f(x)=x+4$ and W is the Wronskian of y_1 and y_2, i.e.

$$W=y_1\,y_2'-y_2\,y_1'\neq 0.$$

```
> W:=y1*y2'-y2*y1':; f:=x+4:;
  yp:=normal(-y1*int(y2*f/W,x)+y2*int(y1*f/W,x))
```

$$\frac{4x^3+72x^2-156x+80}{21}$$

Hence $y_p=\frac{1}{21}\,(4\,x^3+72\,x^2-156\,x+80)$. Now $y=y_p+y_h$. You can checking that it is indeed the general solution of the given equation.

```
> purge(C1,C2):; ysol:=yp+C1*y1+C2*y2:;
  normal(diff(ysol,x,2)-27/(36*(x-1)^2)*ysol)==f
```

true

Solve the equation $y'' = \left(\frac{3}{16\,x\,(x-1)} - \frac{2}{9\,(x-1)^2} - \frac{3}{16\,x^2}\right) y$ from the original Kovacic's paper.

```
> r:=-3/(16x^2)-2/(9*(x-1)^2)+3/(16x*(x-1)):;
  kovacicsols(y''=r*y)
```

$$-\omega\_^4 x^4 (x-1)^4 + \frac{\omega\_^3 x^3 (x-1)^3 (7x-3)}{3} - \frac{\omega\_^2 x^2 (x-1)^2 (48x^2-41x+9)}{24} + \frac{\omega\_\, x\,(x-1)\,(320x^3-409x^2+180x-27)}{432} + \frac{-2048x^4+3484x^3-2313x^2+702x-81}{20736}$$

The solution is $y = \exp\left(\int \omega_0\right)$, where ω_0 is a zero of the above expression, thus being a root of a fourth-order polynomial in ω. In similar cases you can try the Ferrari method to obtain ω_0.

Solve the equation $48\,t\,(t+1)\,(5\,t-4)\,y'' + 8\,(25\,t+16)\,(t-2)\,y' - (5\,t+68)\,y = 0$.

```
> kovacicsols([48t*(t+1)*(5t-4),8*(25t+16)*(t-2),-(5t+68)],t)
```

$$\frac{1}{20736}\omega\_^4\left(135t^4-616t^3-144t^2+3072t-4096\right) - \frac{1}{54}\omega\_^3 t\,(t+1)\left(23t^2-92t+128\right) - \frac{1}{24}\omega\_^2 t^2\,(t+1)\left(15t^3-80t^2+80t+256\right) + \frac{2}{3}\omega\_ t^3\,(t-4)\,(t+1)^2\,(5t+8) - t^4\,(t+1)^2\,(t+4)\,(5t+4)$$

13.5 Taylor series and asymptotic expansions

13.5.1 Dividing by increasing power order

The `divpc` command finds the truncated Taylor expansion of a quotient of polynomials.

- `divpc` takes three mandatory arguments and one optional argument:
  - P and Q, two polynomial expressions such that Q has a nonzero constant term/
  - n, an integer.
  - Optionally, x, the variable name (by default `x`).
- `divpc`$(P, Q, n\,\langle, x\rangle)$ returns the Taylor expansion of P/Q of order n about $x = 0$.

Note that this command does not work on polynomials written as a list of coefficients.

Example

```
> divpc(1+x^2+x^3,1+x^2,5)
```

$$-x^5 + x^3 + 1$$

13.5.2 Series expansion

The `taylor` or `series` command finds Taylor expansions.

- `taylor` takes one mandatory and four optional arguments:
  - *expr*, an expression depending on a variable.
  - Optionally, x, the variable (by default `x`).
  - Optionally n, an integer, the order of the series expansion (by default `5`).

  - Optionally, a, the center of the Taylor expansion (by default 0). This can be combined with the optional x by replacing x by $x = a$.
  - *dir*, a direction, which can be `-1` or `1`, for unidirectional series expansion, or `0` (for bidirectional series expansion) (by default `0`).

- `taylor(`*expr*, x $\langle, a, n,$ *dir*$\rangle$`)` returns the Taylor expansion of *expr* about a or order n; consisting of a polynomial in $x - a$ plus a remainder of the form of the form:

$$(x-a)^n \texttt{ order\_size}(x-a),$$

  where `order_size` is a function such that

$$\forall r > 0, \quad \lim_{x \to 0} x^r \texttt{ order\_size}(x) = 0$$

For regular series expansion, `order_size` is a bounded function, but for non regular series expansion, it might tend slowly to infinity, for example like a power of $\ln(x)$.

Example

```
> taylor(sin(x),x=1,2)
```

or:

```
> series(sin(x),x=1,2)
```

or (be careful with the order of the arguments):

```
> taylor(sin(x),x,2,1)
```

or:

```
> series(sin(x),x,2,1)
```

$$\sin(1) + \cos(1)(x-1) - \frac{1}{2}\sin(1)(x-1)^2 + (x-1)^3 \,\mathrm{order\_size}(x-1)$$

Remark. The order returned by `taylor` may be smaller than n if cancellations between numerator and denominator occur, for example consider

$$\frac{x^3 + \sin(x)^3}{x - \sin(x)}.$$

```
> taylor(x^3+sin(x)^3/(x-sin(x)),x=0,5)
```

$$6 - \frac{27}{10}x^2 + x^3 + \frac{711}{1400}x^4 + x^6 \mathrm{order\_size}(x)$$

which is only a 2^{nd} degree expansion. Indeed the numerator and denominator valuation is 3, hence you lose 3 orders. To get order 4, you should use $n = 7$.

```
> taylor(x^3+sin(x)^3/(x-sin(x)),x=0,7)
```

$$6 - \frac{27}{10}x^2 + x^3 + \frac{711}{1400}x^4 - \frac{737}{14000}x^6 + x^8 \mathrm{order\_size}(x)$$

a fourth degree expansion.

Examples

Find a 4th-order expansion of $\cos(2x)^2$ in the vicinity of $x = \frac{\pi}{6}$.

```
> taylor(cos(2*x)^2,x=pi/6, 4)
```

$$\frac{1}{4} - \sqrt{3}\left(x - \frac{\pi}{6}\right) + 2\left(x - \frac{\pi}{6}\right)^2 + \frac{8}{3}\sqrt{3}\left(x - \frac{\pi}{6}\right)^3 - \frac{8}{3}\left(x - \frac{\pi}{6}\right)^4 + \left(x - \frac{\pi}{6}\right)^5 \text{order\_size}\left(x - \frac{\pi}{6}\right)$$

Find a 5th-order series expansion of $\arctan(x)$ in the vicinity of $x = +\infty$.

```
> series(atan(x),x=+infinity,5)
```

$$\frac{\pi}{2} - \frac{1}{x} + \frac{\left(\frac{1}{x}\right)^3}{3} - \frac{\left(\frac{1}{x}\right)^5}{5} + \left(\frac{1}{x}\right)^6 \text{order\_size}\left(\frac{1}{x}\right)$$

Note that the expansion variable and the argument of the `order_size` function is $h = \frac{1}{x} \to 0$ as $x \to +\infty$.

Find a 2nd-order expansion of $(2x-1)\mathrm{e}^{\frac{1}{x-1}}$ in the vicinity of $x = +\infty$.

```
> series((2*x-1)*exp(1/(x-1)),x=+infinity,3)
```

Output (only a 1st-order series expansion):

$$2\left(\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6}\left(\frac{1}{x}\right)^2 + \left(\frac{1}{x}\right)^3 \text{order\_size}\left(\frac{1}{x}\right)$$

Note that this is only a 1st-order expansion. To get a 2nd-order series expansion in $1/x$:

```
> series((2*x-1)*exp(1/(x-1)),x=+infinity,4)
```

$$2\left(\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6}\left(\frac{1}{x}\right)^2 + \frac{47}{12}\left(\frac{1}{x}\right)^3 + \left(\frac{1}{x}\right)^4 \text{order\_size}\left(\frac{1}{x}\right)$$

Find a 2nd-order series expansion of $(2x-1)\mathrm{e}^{\frac{1}{x-1}}$ in the vicinity of `x=-`∞.

```
> series((2*x-1)*exp(1/(x-1)),x=-infinity,4)
```

$$-2\left(-\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6}\left(-\frac{1}{x}\right)^2 - \frac{47}{12}\left(-\frac{1}{x}\right)^3 + \left(-\frac{1}{x}\right)^4 \text{order\_size}\left(-\frac{1}{x}\right)$$

Find a 2nd-order series expansion of $\frac{(1+x)^{\frac{1}{x}}}{x^3}$ in the vicinity of $x = 0^+$.

```
> series((1+x)^(1/x)/x^3,x=0,2,1)
```

(Note that this is a one-sided series expansion, since *dir* = `1`.)

$$\mathrm{e}\,x^{-3} - \frac{\mathrm{e}}{2}x^{-2} + x^{-1}\text{order\_size}\,(x)$$

13.5.3 Inverse of a series

The `revert` command finds the beginning of the power series of the inverse function given the beginning of the series of the original function.

- `revert` takes one mandatory and one optional argument:
  - *series*, the beginning of a power series centered at 0 for a function f.
  - Optionally x, the name of the variable (by default x).
- `revert(`*series*$\langle, x\rangle$`)` returns the beginning of the power series for the inverse of f; namely the beginning of the power series for $g(f(0)+x)$ where the function g satisfies $g(f(x)) = x$.

Examples

Find the series expansion of $f^{-1}(x)$ where $f(x) = x + x^2 + x^4 + \cdots$

```
> revert(x+x^2+x^4)
```

$$x - x^2 + 2x^3 - 6x^4$$

Note that if the power series of a function f begins with $x + x^2 + x^4$, then $f(0) = 0$, $f'(0) = 1$, $f''(0) = 2$, $f'''(0) = 0$ and $f^{(4)}(0) = 24$. The function g with $g(f(x)) = x$ will then satisfy $g(0) = 0$, $g'(0) = 1/f'(0) = 1$, $g''(0) = -2$, $g'''(0) = 12$ and $g^{(4)}(0) = -144$. The power series for g will then begin with $x - x^2 + 2x^3 - 6x^4$.

If the argument is the beginning of the power series for e^x, then the output will be the beginning of the power series for $\ln(1 + x)$.

```
> revert(1+x+x^2/2+x^3/6+x^4/24)
```

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}$$

13.5.4 Residue of an expression at a point

The `residue` command finds the residue of an expression at a point.

- `residue` takes three arguments:
  - *expr*, an expression depending on a variable.
  - x, a variable name.
  - a, a complex number. This can be combined with the previous argument in an equality $x = a$.
- `residue(`*expr*$, x, a$`)` returns the residue of *expr* at the point a.

Example

```
> residue(cos(x)/x^3,x,0)
```

or:

```
> residue(cos(x)/x^3,x=0)
```

$$-\frac{1}{2}$$

13.5.5 Padé expansion

The `pade` command finds a rational expression which agrees with a function up to a given order.

- `pade` takes 4 arguments
  - *expr*, an expression.
  - x, the variable name.
  - n, an integer or R, a polynomial.
  - p, an integer.
- `pade(`*expr*$, x, n, p$`)` or `pade(`*expr*$, x, P, p$`)` returns a rational function P/Q such that degree$(P) < p$ and $P/Q \equiv$ *expr* $\pmod{x^{n+1}}$ (meaning that P/Q and f have the same Taylor expansion at 0 up to order n) or $P/Q \equiv$ *expr* $\cdot f \pmod{R}$, respecively.

Examples

```
> pade(exp(x),x,5,3)
```

or:

```
> pade(exp(x),x,x^6,3)
```

$$\frac{-3x^2 - 24x - 60}{x^3 - 9x^2 + 36x - 60}$$

To verify:

```
> taylor((3*x^2+24*x+60)/(-x^3+9*x^2-36*x+60))
```

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + x^6 \text{order\_size}\,(x)$$

which is the 5th-order series expansion of `exp(x)` at $x = 0$.

```
> pade((x^15+x+1)/(x^12+1),x,12,3)
```

or:

```
> pade((x^15+x+1)/(x^12+1),x,x^13,3)
```

$$x + 1$$

```
> pade((x^15+x+1)/(x^12+1),x,14,4)
```

or:

```
> pade((x^15+x+1)/(x^12+1),x,x^15,4)
```

$$\frac{2x^3 + 1}{x^{11} - x^{10} + x^9 - x^8 + x^7 - x^6 + x^5 - x^4 + x^3 + x^2 - x + 1}$$

To verify:

```
> series(ans(),x=0,15)
```

$$1 + x - x^{12} - x^{13} + 2x^{15} + x^{16} \text{order\_size}\,(x)$$

Then:

```
> series((x^15+x+1)/(x^12+1),x=0,15)
```

$$1 + x - x^{12} - x^{13} + x^{15} + x^{16} \text{order\_size}\,(x)$$

These two expressions have the same 14th-order series expansion at $x = 0$.

13.6 Z-transform

13.6.1 Z-transform of a sequence

The Z-transform of a sequence $a_0, a_1, \ldots, a_n, \ldots$ is the function

$$f(z) = \sum_{n=0}^{\infty} \frac{a_n}{z^n}.$$

For example, the Z-transform of the sequence $0, 1, 2, 3, \ldots$ is

$$f(z) = 0 + \frac{1}{z} + \frac{2}{z^2} + \frac{3}{z^3} + \cdots$$

which has closed form

$$f(z) = \frac{z}{(z-1)^2}.$$

The `ztrans` command finds the Z-transform of a sequence.

- `ztrans` takes one mandatory and two optional arguments:
  - a_x, a formula with a variable for the general term of a sequence.
  - Optionally, x, the variable (by default `x`).
  - Optionally, z, a variable to be used by the resulting function.

  `ztrans(`a_x $\langle, x, z\rangle$`)` returns the Z-transform of the input sequence.

Examples

To find the Z-transform of the identity function:

```
> ztrans(x)
```

$$\frac{x}{x^2 - 2x + 1}$$

With n as the original variable and z as the transform variable:

```
> ztrans(n,n,z)
```

$$\frac{z}{z^2 - 2z + 1}$$

Find the Z-transform of the constant function $f(x) = 1$:

```
> ztrans(1)
```

$$\frac{x}{x-1}$$

Indeed:

$$\sum_{n=0}^{\infty} \frac{1}{x^n} = \frac{1}{1 - \frac{1}{x}} = \frac{x}{x-1}.$$

You also have

```
> ztrans(1,n,z)
```

$$\frac{z}{z-1}$$

Note that differentiating both sides of

$$\sum_{n=0}^{\infty} \frac{1}{z^n} = \frac{z}{z-1}$$

gives you

$$\sum_{n=0}^{\infty} \frac{n}{z^{n-1}} = \frac{1}{(z-1)^2},$$

and so, multiplying both sides by z,

$$\sum_{n=0}^{\infty} \frac{n}{z^n} = \frac{z}{(z-1)^2} = \frac{z}{z^2 - 2z + 1},$$

as indicated in the previous example.

13.6.2 Inverse Z-transform of a rational function

The inverse Z-transform of a rational expression is a formula for the general term of a sequence with the given rational expression as its Z-transform. The `invztrans` command finds the inverse Z-transform of a rational expression.

- `invztrans` command takes one mandatory and two optional arguments:
  - *rat*, a rational expression.
  - Optionally, x the variable (by default `x`).
  - Optionally, n, a variable to be used by the result (by default x).
- `ztrans(rat` $\langle, x, n\rangle$`)` returns the inverse Z-transform of *rat*.

Examples

```
> invztrans(x/(x-1))
```

$$1$$

since `ztrans(1)` yields $\frac{x}{x-1}$.

```
> invztrans(z/(z-1),z,n)
```

$$1$$

```
> invztrans(x/(x-1)^2)
```

$$x$$

```
> invztrans(z/(z-1)^2,z,n)
```

$$n$$

13.7 Multivariate calculus

13.7.1 Gradient

The `derive` command finds partial derivatives of a multivariable expression. `diff` and `grad` can be used synonymously for `derive` here.

- `derive` takes two arguments:
  - *expr*, an expression involving n real variables.
  - $[x_1, \ldots, x_n]$, a vector of the variable names.
- `derive(`*expr*, `[`$x_1, \ldots, x_n$`])` returns the gradient of *expr*; namely, the vector of partial derivatives of *expr* with respect to $x_1, \ldots, x_n$.

  For instance, in dimension $n = 3$, with variables $[x, y, z]$:

$$\overrightarrow{\text{grad}}(F) = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right].$$

Example

Find the gradient of $F(x, y, z) = 2x^2y - xz^3$.

```
> derive(2*x^2*y-x*z^3,[x,y,z])
```

or:

```
> diff(2*x^2*y-x*z^3,[x,y,z])
```

or:

```
> grad(2*x^2*y-x*z^3,[x,y,z])
```

$$\left[2 \cdot 2xy - z^3, 2x^2, -3xz^2\right]$$

```
> normal(ans())
```

$$\left[4xy - z^3, 2x^2, -3xz^2\right]$$

To find the critical points of $F(x, y, z) = 2x^2y - xz^3$:

```
> solve(derive(2*x^2*y-x*z^3,[x,y,z]),[x,y,z])
```

$$[[0, y, 0]]$$

13.7.2 Laplacian

Recall, the Laplacian of a function F of n variables $x_1, \ldots, x_n$ is

$$\nabla^2(F) = \frac{\partial^2 F}{\partial x_1^2} + \frac{\partial^2 F}{\partial x_2^2} + \cdots + \frac{\partial^2 F}{\partial x_n^2}.$$

Also, the $n \times n$ discrete Laplacian matrix (also called the second difference matrix) is the $n \times n$ tridiagonal matrix with 2s on the main diagonal, -1s just above and below the main diagonal;

$$\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix}$$

If L is the $n \times n$ discrete Laplacian matrix and Y is an $n \times 1$ column vector whose kth coordinate is $y_i = y(a + k\Delta x)$ for a twice differential function y, then the kth coordinate of LY will be (implicitly assuming that $y(a) = y(a + (N + 1)\Delta x) = 0$):

$$-y(a + (k - 1)\Delta x) + 2y(a + k\Delta x) - y(a + (k - 1)\Delta x),$$

which approximates $y''(a + k\Delta x)$. So LY is approximately $-\Delta x^2 Y''$, where Y'' is the $n \times 1$ column vector whose kth coordinate is $y''(a + k\delta x)$.

The `laplacian` command can compute the Laplacian operator or the discrete Laplacian matrix.

- To compute the Laplacian operator, `laplacian` takes two arguments:
  - *expr*, an expression involving several variables.
  - *vars*, a list of the variable names.
- `laplacian(`*expr*, *vars*`)` returns the Laplacian of the expression.
- To compute the discrete Laplacian matrix, `laplacian` takes n, an integer or floating-point integer.
- `laplacian(`n`)` returns the $n \times n$ discrete Laplacian matrix.

Examples

Find the Laplacian of $F(x,y,z) = 2x^2y - xz^3$.

```
> laplacian(2*x^2*y-x*z^3,[x,y,z])
```

$$-6xz + 4y$$

```
> laplacian(3)
```

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

```
> laplacian(2.0)
```

$$\begin{bmatrix} 2.0 & -1.0 \\ -1.0 & 2.0 \end{bmatrix}$$

13.7.3 Hessian matrix

Recall, the Hessian of a function F of n variables $x_1, \ldots, x_n$ is the matrix of second order derivatives:

$$\begin{bmatrix} \frac{\partial^2 F}{\partial x_1^2} & \cdots & \frac{\partial^2 F}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial^2 F}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 F}{\partial x_n^2} \end{bmatrix}$$

The `hessian` command computes the Hessian of a function.

- `hessian` takes two arguments:
  - *expr*, an expression involving several variables.
  - *vars*, a list of the variable names.
- `hessian(`*expr*, *vars*`)` returns the Hessian of the expression.

Examples

Find the Hessian matrix of $F(x,y,z) = 2x^2y - xz^3$.

```
> hessian(2*x^2*y-x*z^3,[x,y,z])
```

$$\begin{bmatrix} 4y & 4x & -3z^2 \\ 2 \cdot 2x & 0 & 0 \\ -3z^2 & 0 & -2 \cdot 3xz \end{bmatrix}$$

To get the Hessian matrix at the critical points:

```
> solve(derive(2*x^2*y-x*z^3,[x,y,z]),[x,y,z])
```

Output (the critical points):

$$[[0, y, 0]]$$

Input to evaluate the Hessian at these points:

```
> subst([[4*y,4*x,-3*z^2],[2*2*x,0,0],[-3*z^2,0,6*x*z]],[x,y,z],[0,y,0])
```

$$\begin{bmatrix} 4y & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

13.7.4 Divergence

Recall that the divergence of a vector field $\mathbf{F} = [F_1, \ldots, F_n]$ with variables $[x_1, \ldots, x_n]$ is

$$\operatorname{div} \mathbf{F} = \frac{\partial F_1}{x_1} + \cdots + \frac{\partial F_n}{x_n}.$$

The `divergence` command computes the divergence of a vector field.

- `divergence` takes two arguments:
  - F, a vector field given as a list $[F_1, \ldots, F_n]$ of expressions.
  - *vars*, a list of the variable names.
- `divergence(`F, *vars*`)` returns the divergence of the vector field F.

Example

```
> divergence([x*z,-y^2,2*x^y],[x,y,z])
```

$$-2y + z$$

13.7.5 Rotational

The curl of a 3D vector field $\mathbf{F} = [F_1, F_2, F_3]$ with variables $[x_1, x_2, x_3]$ is

$$\operatorname{curl} \mathbf{F} = \left[\frac{\partial F_3}{\partial x_2} - \frac{\partial F_2}{\partial x_3}, \frac{\partial F_1}{\partial x_3} - \frac{\partial F_3}{\partial x_1}, \frac{\partial F_2}{\partial x_1} - \frac{\partial F_1}{\partial x_2}\right].$$

The `curl` command computes the curl of a three dimensional vector field (note that it must be three dimensional).

- `curl` takes two arguments:
  - F, a 3D vector field, given as a list of three expressions depending on three variables.
  - *vars*, a list of the three variable names.
- `curl(`F, *vars*`)` returns the curl of the vector field.

Example

```
> curl([x*z,-y^2,2*x^y],[x,y,z])
```

$$\left[2 \ln x \cdot x^y, x - 2yx^{y-1}, 0\right]$$

13.7.6 Potential

Recall that a vector field $\mathbf{F}$ is conservative if there is a scalar-valued function f such that $\operatorname{grad} f = \mathbf{F}$. In this case, f is called a potential of $\mathbf{F}$, and is only determined up to a constant.

The `potential` command computes the potential of a vector field, or signals an error if the vector field is not conservative.

- `potential` takes two arguments:
  - $\mathbf{F}$, a vector field given as a list of n expressions involving n variables.
  - *vars*, a list of the variable names.

- `potential(`$\mathbf{F}$, *vars*`)` returns a potential function for $\mathbf{F}$ if $\mathbf{F}$ is conservative, and raises an error otherwise.

Note that `potential` is the reciprocal function of `derive`.

Also note that in $\mathbb{R}^3$, a vector field $\mathbf{F}$ is conservative if and only if its curl is zero; i.e., if $\operatorname{curl}\mathbf{F} = \mathbf{0}$. In time-independent electro-magnetism, $\mathbf{F} = \mathbf{E}$ is the electric field and f is the electric potential.

Example

```
> potential([2*x*y+3,x^2-4*z,-4*y],[x,y,z])
```

$$x^2y + 3x - 4yz$$

13.7.7 Conservative flux field

A vector field $\mathbf{F}$ in $\mathbb{R}^3$ is a conservative flux field, or a solenoidal field, if there is a vector field $\mathbf{G}$ such that $\operatorname{curl}\mathbf{G} = \mathbf{F}$. Given a conservative flux vector field $\mathbf{F}$, the general solution of $\operatorname{curl}\mathbf{G} = \mathbf{F}$ is the sum of a particular solution and the gradient of an arbitrary functions.

The `vpotential` command finds a particular vector field $\mathbf{G}$ such that $\operatorname{curl}\mathbf{G} = \mathbf{F}$ if $\mathbf{F}$ is a conservative flux field, and signals an error otherwise. Specifically, `vpotential` returns the solution $\mathbf{G}$ with zero as the first component.

- `vpotential` takes two arguments:
  - F, a vector field in $\mathbb{R}^3$, given as a list of three expressions depending on three variables.
  - *vars*, a list of the variable names.
- `vpotential(`F*vars*`)` returns a solution of $\operatorname{curl}\mathbf{G} = \mathbf{F}$ whose first coordinate is zero if $\mathbf{F}$ is a conservative vector field, and signals an error otherwise.

`vpotential` is the reciprocal function of `curl`.

In $\mathbb{R}^3$, a vector field $\mathbf{F}$ is a curl if and only if its divergence is zero. In time-independent electro-magnetism, $\mathbf{F} = \mathbf{B}$ is the magnetic field and $\mathbf{G} = \mathbf{A}$ is the potential vector.

Example

```
> vpotential([2*x*y+3,x^2-4*z,-2*y*z],[x,y,z])
```

$$\left[0, -2xyz, -\frac{x^3}{3} + 4xz + 3y\right]$$

13.7.8 Determining where a function is convex

The `convex` command determines where a function is convex.

- `convex` takes two mandatory arguments and one optional argument:
  - *expr*, an expression which is at least twice differentiable, which specifies a function f.
  - *vars*, the variable or list of variables in the expression. Some variables may depend on a common independent parameter, say t, when entered as e.g. $x(t)$ instead of x. The first derivatives of such variables, when encountered in f, are treated as independent parameters of the function.

  - Optionally, `simplify`, which enables advanced simplification of the intermediate symbolic expressions and the final result.

- `convex(`*expr*, *vars* ⟨, `simplify=`*bool*⟩`)` returns:
  - `true`, if the function is convex on the entire domain.
  - `false`, if the function is nowhere convex.
  - otherwise, the list of inequalities specifying the domain on which the function is convex.
- The `convex` command operates by computing the Hessian H_f of f (see Section 13.7.3, p. 310) and its LDL factorization. If the resulting block-diagonal matrix is positive semidefinite, then H_f is positive semidefinite and f is hence convex.
- The algorithm respects the assumptions that may be set upon variables. Therefore, the convexity of a given function can be checked on a particular domain.
- Advanced simplification can be applied when generating convexity conditions by passing the `simplify` argument. By default, only basic simplification is applied by using the `ratnormal` command.
- The function f concave if and only if the function $g = -f$ is convex.

Examples

Verify that $f(x) = 3\,\mathrm{e}^x + 5x^4 - \ln(x)$ is convex on $\mathbb{R}$:

```
> convex(3*exp(x)+5x^4-ln(x),x)
```

true

Verify that $f(x, y, z) = x^2 + y^2 + 3z^2 - xy + 2xz + yz$ is convex on $\mathbb{R}^3$:

```
> convex(x^2+y^2+3z^2-x*y+2x*z+y*z,[x,y,z])
```

true

The function $f(x_1, x_2) = x_1^3 + 2x_1^2 + 2x_1x_2 + \frac{x_2^2}{2} - 8x_1 - 2x_2 - 8$ is convex on $[0, +\infty\rangle \times \mathbb{R}$:

```
> convex(x1^3+2x1^2+2*x1*x2+x2^2/2-8x1-2x2-8,[x1,x2],simplify)
```

$[x_1 \geq 0]$

Find the values of $a \in \mathbb{R}$ for which the function $f(x, y, z) = x^2 + x\,z + a\,y\,z + z^2$ is convex:

```
> convex(x^2+x*z+a*y*z+z^2,[x,y,z])
```

false

Note that the function is convex for $a = 0$. However, the `convex` command does not support equalities as convexity constraints.

Find all values $a \in \mathbb{R}$ for which the function $f(x, y, z) = x^2 + 2\,y^2 + a\,z^2 - 2\,x\,y + 2\,x\,z - 6\,y\,z$ is convex on $\mathbb{R}^3$:

```
> convex(x^2+2y^2+a*z^2-2x*y+2x*z-6y*z,[x,y,z],simplify)
```

$[a \geq 5]$

Find the set $S \subset \mathbb{R}^2$ on which the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x_1, x_2) = \mathrm{e}^{x_1} + \mathrm{e}^{x_2} + x_1\,x_2$ is convex:

```
> condition:=convex(exp(x1)+exp(x2)+x1*x2,[x1,x2],simplify)
```

$[\mathrm{e}^{x_1}\mathrm{e}^{x_2} - 1 \geq 0]$

```
> lin(condition)
```

$$\left[e^{x_1+x_2} - 1 \geq 0\right]$$

(See Section 10.3.4, p. 206.) From here you conclude that f is convex when $x_1 + x_2 \geq 0$. The set S is therefore the half-space defined by this inequality.

The domain of the input function can be restricted by setting assumptions on variables:

```
> assume(x1>0),assume(x2>0):;
  convex(exp(x1)+exp(x2)+x1*x2,[x1,x2])
```

true

which was already verified in the previous example.

13.8 Calculus of variations

13.8.1 Motivation: the Brachistochrone

The *Brachistochrone problem* is perhaps the original problem in the calculus of variations. The problem is to find the curve from two points in a plane such that an object falling under its own weight will get from the first point to the second in the shortest time.

If the points are $(0, y_0)$ and $(x_1, 0)$, with $y_0 > 0$ and $x_1 > 0$, this becomes the problem of minimizing the objective functional

$$T(y) = \int_0^{x_1} L(t, y(x), y'(x))\,\mathrm{d}x$$

where the function L is defined by

$$L(t, y(x), y'(x)) = \sqrt{\frac{1 + y'(x)^2}{2\,g\,y(x)}}$$

for $y : [0, x_1] \to \mathbb{R}$ such that $y(0) = y_0$ and $y(x_1) = 0$ (the constant g is the gravitational acceleration).

More generally, one type of problem in the Calculus of variations is to minimize (or maximize) a functional

$$F(y) = \int_a^b f(x, y, y')\,\mathrm{d}x$$

over all functions $y \in C^2[a, b]$ with boundary conditions $y(a) = A$ and $y(b) = B$, where $A, B \in \mathbb{R}$. The function f is called the *Lagrangian*.

13.8.2 Euler-Lagrange equations

The *Euler-Lagrange equations* for a Lagrangian function $f(x, y, y')$ are differential equations which must be satisfied by extrema of the functional $F(y)$.

The `euler_lagrange` command finds the Euler-Lagrange equations for a Lagrangian f.

- `euler_lagrange` takes one mandatory argument and two optional arguments:
  - *expr*, an expression involving an independent variable, a dependent variable, and the dependent variable derivative.
  - Optionally, *indvar*, the independent variable (by default x).
  - Optionally, *depvar*, the dependent variable (by default y).

    If a function $y \in \mathbb{R}^n$ is required (by default $n = 1$), you can enter $y = (y_1, y_2, \ldots, y_n)$ as a vector $[y_1, y_2, \ldots, y_n]$. In that case, $y' = (y_1', y_2', \ldots, y_n')$.

An alternate way to specify the independent and dependent variables is by replacing both optional arguments by either, for example, $y(x)$ or $[y_1(x), y_2(x), \ldots, y_n(x)]$.

- `euler_lagrange(`*expr*⟨, *indvar*, *depvar*⟩`)` returns the system of differential Euler-Lagrange equations.

  For $n = 1$, a single equation is returned: $\frac{\partial f}{\partial y} = \frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial f}{\partial y'}$.

  In general, n equations are returned: $\frac{\partial f}{\partial y_k} = \frac{\mathrm{d}}{\mathrm{d}x}\frac{\partial f}{\partial y'_k}$, $k = 1, 2, \ldots, n$.

The degrees of these differential equations are kept as low as possible. If, for example, $\frac{\partial f}{\partial y} = 0$, the equation $\frac{\partial f}{\partial y'} = K$ is returned, where $K \in \mathbb{R}$ is an arbitrary constant. Similarly, using the *Hamiltonian*

$$H(x, y, y') = y' \frac{\partial}{\partial y'} f(x, y, y') - f(x, y, y')$$

the Euler-Lagrange equation is simplified in the case $n = 1$ and $\frac{\partial f}{\partial t} = 0$ to:

$$H(x, y, y') = K, \tag{13.3}$$

since it can be shown that $\frac{\mathrm{d}}{\mathrm{d}x} H(y, y', x) = 0$. Therefore the Euler-Lagrange equations, which are generally of order two in y, are returned in a simpler form of order one in the aforementioned cases. If $n = 1$ and $\frac{\partial f}{\partial x} = 0$, then both equations are returned, each of them being sufficient to determine y (one of the returned equations is usually simpler than the other).

Examples

Minimize the functional F for $0 < a < b$ and $f(x, y, y') = x^2\, y'(x)^2 + y(x)^2$.

```
> eq:=euler_lagrange(x^2*diff(y(x),x)^2+y^2)
```

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2} y\,(x) = \frac{-2\frac{\mathrm{d}}{\mathrm{d}x} y\,(x)\, x + y\,(x)}{x^2}$$

This can be solved by assuming $y(x) = x^r$ for some $r \in \mathbb{R}$.

```
> solve(subs(eq,y(x)=x^r),r)
```

$$\left[-\frac{\sqrt{5}+1}{2}, -\frac{-\sqrt{5}+1}{2}\right]$$

The same pair of solutions is also returned by the `kovacicsols` command (see Section 13.4.3, p. 299):

```
> assume(x>=0):;
  kovacicsols(y''=(y-2x*y')/x^2,x,y)
```

$$\left[\sqrt{x^{\sqrt{5}-1}}, \sqrt{x^{-\sqrt{5}-1}}\right]$$

You can conclude that $y = C_1\, x^{-\frac{\sqrt{5}+1}{2}} + C_2\, x^{\frac{\sqrt{5}+1}{2}}$. The values of C_1 and C_2 are determined from the boundary conditions. Finally, to prove that f is convex:

```
> convex(x^2*diff(y(x),x)^2+y^2,y(x))
```

true

Therefore, y minimizes F on $[a, b]$.

Find the function y in $\left\{y \in C^1\left[\frac{1}{2},1\right] : y\left(\frac{1}{2}\right) = -\frac{\sqrt{3}}{2}, y(1) = 0\right\}$ which minimizes the functional

$$F(y) = \int_{1/2}^{1} \frac{\sqrt{1+y'(x)^2}}{x}\,\mathrm{d}x.$$

To obtain the corresponding Euler-Lagrange equation:

```
> eq:=euler_lagrange(sqrt(1+diff(y(x),x)^2)/x)
```

$$\frac{\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)}{x\sqrt{\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^2+1}} = K_2$$

```
> sol:=dsolve(eq)
```

$$\left[-\frac{\sqrt{-K_3^2x^2+1}}{K_3}+c_0\right]$$

The sought solution is the function of the above form which satisfies the boundary conditions.

```
> y0,c:=sol[0],[K_3,c_0]:;
  v:=solve([subs(y0,x=1/2)=-sqrt(3)/2,subs(y0,x=1)=0],c)
```

$$\begin{bmatrix}1 & 0\end{bmatrix}$$

```
> y0:=normal(subs(y0,c,v[0]))
```

$$-\sqrt{-x^2+1}$$

If the integrand in $F(y)$ is convex, then $y_0(x) = -\sqrt{1-x^2}$ is a minimizer for F. Indeed:

```
> convex(sqrt(1+y'^2)/x,y(x))
```

$$[x \geq 0]$$

You can similarly find the minimizer for

$$F(y) = \int_0^{\pi} \left(2\,\sin(x)\,y(x) + y'(x)^2\right)\,\mathrm{d}x$$

where $y \in C^1[0,\pi]$ and $y(0) = y(\pi) = 0$.

```
> eq:=euler_lagrange(2*sin(x)*y(x)+diff(y(x),x)^2)
```

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}y\left(x\right) = \sin x$$

```
> dsolve(eq and y(0)=0 and y(pi)=0,x,y)
```

$$-\sin x$$

The above function is the sought minimizer as the integrand $2\,\sin(x)\,y(x) + y'(x)^2$ is convex:

```
> convex(2*sin(x)*y(x)+diff(y(x),x)^2,y(x))
```

$$\mathrm{true}$$

Minimize the functional $F(y) = \int_0^1 (y'(x)^4 - 4\,y(x))\,\mathrm{d}x$ on $C^1[0,1]$ with boundary conditions $y(0) = 1$ and $y(1) = 2$.

First, solve the associated Euler-Lagrange equation:

```
> eq:=euler_lagrange(y'^4-4y,x,y)
```

$$\left[3\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^4 + 4y\left(x\right) = K_6, \frac{\mathrm{d}^2}{\mathrm{d}x^2}y\left(x\right) = -\frac{1}{3\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^2}\right]$$

```
> dsolve(eq[1] and y(0)=1 and y(1)=2,x,y)
```

$$\left[-\frac{3}{4}\left(-x + 1.52832425067\right)^{\frac{4}{3}} + 2.32032831141\right]$$

Next, check if the integrand in $F(y)$ is convex:

```
> convex(y'^4-4y,[x,y])
```

true

Hence the minimizer is $y_0(x) = -\frac{3}{4}\left(1.52832425067 - x\right)^{4/3} + 2.32032831141$, $0 \le x \le 1$.

Find Euler-Lagrange equations for a bivariate functional F:

```
> euler_lagrange(sqrt(x'(t)^2+y'(t)^2),[x(t),y(t)])
```

$$\left[\frac{\frac{\mathrm{d}}{\mathrm{d}t}x\left(t\right)}{\sqrt{\left(\frac{\mathrm{d}}{\mathrm{d}t}x\left(t\right)\right)^2 + \left(\frac{\mathrm{d}}{\mathrm{d}t}y\left(t\right)\right)^2}} = K_0, \frac{\frac{\mathrm{d}}{\mathrm{d}t}y\left(t\right)}{\sqrt{\left(\frac{\mathrm{d}}{\mathrm{d}t}x\left(t\right)\right)^2 + \left(\frac{\mathrm{d}}{\mathrm{d}t}y\left(t\right)\right)^2}} = K_1\right]$$

where $K_0, K_1 \in \mathbb{R}$ are arbitrary constants (note that these symbols are generated automatically).

It can be proven that if f is convex (as a function of three independent variables, see Section 13.7.8, p. 312), then a solution y to the Euler-Lagrange equations minimizes the functional F.

13.8.3 Solving the Brachistochrone Problem

To solve the brachistochrone problem (see Section 13.8.1, p. 314), you can first find the Euler-Lagrange equations for the Lagrangian

$$L(x, y(x), y'(x)) = \sqrt{\frac{1 + y'(x)^2}{2\,g\,y(x)}}.$$

You can simplify this somewhat by assuming that you are using units where $2g = 1$.

```
> assume(y>=0):; euler_lagrange(sqrt((1+y'^2)/y),x,y)
```

$$\left[-\frac{1}{\sqrt{\left(\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^2 + 1\right)y\left(x\right)}} = K_2, \frac{\mathrm{d}^2}{\mathrm{d}x^2}y\left(x\right) = \frac{-\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^2 - 1}{2y\left(x\right)}\right]$$

It is easier to solve the first equation for y which can be rewritten as

$$\frac{\mathrm{d}y}{\mathrm{d}x} = -\sqrt{\frac{C}{x} - 1}$$

for appropriate C, which can be solved by separation of variables, getting you the parametric equations

$$\begin{cases} x = \frac{1}{2}C\left(2\theta - \sin(2\theta)\right) \\ y = \frac{1}{2}C\left(1 - \cos(2\theta)\right) \end{cases}$$

which parameterize a cycloid. This implicitly defines a function $y = \overline{y}(x)$ as the only stationary function for L. The problem is to prove that it minimizes T, which would be easy if the integrand L was convex. However, it's not the case here:

```
> assume(y>=0):;
  convex(sqrt((1+y'^2)/y),y(x))
```

$$\left[-\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^{2}+3\geq 0\right]$$

This is equivalent to $|y'(t)| \leq \sqrt{3}$, which is certainly not satisfied by the cycloid $\overline{y}$ near the point $x = 0$.

Using the substitution $y(x) = z(x)^2/2$, you get $y'(x) = z'(x)\,z(x)$ and

$$L(x, y(x), y'(x)) = P(x, z(x), z'(x)) = \sqrt{2(z(x)^{-2} + z'(x)^2)}.$$

The function P is convex:

```
> assume(z>=0):; convex(sqrt(2*(z^(-2)+z'^2)),z(x))
```

true

Hence the function $\overline{z}(t) = \sqrt{2\,\overline{y}(t)}$, stationary for P (verified directly), minimizes the objective functional

$$U(z) = \int_0^{x_1} P(x, z(x), z'(x))\,\mathrm{d}x.$$

From here and $U(z) = T(y)$ it easily follows that $\overline{y}$ minimizes T and is therefore the brachistochrone.

13.8.4 Jacobi equation

When determining whether a solution y_0 to the Euler-Lagrange equations is an extremum, checking the convexity of the Lagrangian f does not always work. In such cases you may use the Jacobi equation:

$$-\frac{\mathrm{d}}{\mathrm{d}t}\left(f_{y'y'}(y_0, y_0', t)\,h'\right) + \left(f_{yy}(y_0, y_0', t) - \frac{\mathrm{d}}{\mathrm{d}t}\,f_{yy'}(y_0, y_0', t)\right)\,h = 0. \tag{13.4}$$

for an unknown function h. If the Jacobi equation has a solution such that $h(a) = 0$, $h(c) = 0$ for some $c \in (a, b]$ (the interval given in the variational problem) and h not identically zero on $[a, c]$, then c is called a *conjugate* to a. If a conjugate exists, then y_0 does not minimize F. But the function y_0 minimizes F if $f_{y'y'}(y_0, y_0', x) > 0$ for all $x \in [a, b]$ and there are no points conjugate to a in $(a, b]$.

The `jacobi_equation` command computes the Jacobi equation.

- `jacobi_equation` takes five or six arguments:
  - $f(y, y', x)$, an expression involving an independent variable, a dependent variable, and the dependent variable derivative.
  - *depvar*, the independent variable.
  - *indvar*, the dependent variable. This argument and the previous one can be combined to a single argument *depvar(indvar)*, which case the call has five arguments.
  - expression y_0 representing a function in $C^1[a, b]$ which is stationary for the functional $F(y) = \int_a^b f(y, y', x)\,\mathrm{d}x$.
  - h, a symbol for the unknown function in the Jacobi equation.
  - a, a real number which is the lower bound for x.
- `jacobi_equation(`$f(y, y', x), x, y, y_0, a$`)` returns the Jacobi equation and possibly the solution.

If the Jacobi equation can be solved by `dsolve` (see Section 13.4.1, p. 292), a sequence containing the equation (13.4) and its solution is returned. Otherwise, if (13.4) cannot be solved immediately, only the Jacobi equation is returned.

Example

```
> jacobi_equation(-1/2*y'(t)^2+y(t)^2/2,t,y,sin(t),h,0)
```

$$-\frac{\mathrm{d}^2}{\mathrm{d}t^2}h\left(t\right)-h\left(t\right)=0,c_0\sin t$$

13.8.5 Finding conjugate points

The `conjugate_equation` command computes conjugate points.

- `conjugate_equation` takes four arguments:
  - y_0, an expression which depends on an independent variable and two parameters. The expression y_0 is assumed to represent a stationary function for the problem of minimizing some functional $F(y)=\int_a^b f(x,y,y')\,\mathrm{d}x$.
  - $[\alpha,\beta]$, a list of parameters which y_0 depends on.
  - $[A,B]$, a list of the values of parameters α and β, respectively.
  - x, the independent variable.
  - a, a real number equal to the lower or to the upper bound for x.
- `conjugate_equation`$(y_0,[\alpha,\beta],[A,B],x,a)$ returns the expression

$$\frac{\partial y_0(t)}{\partial\alpha}\frac{\partial y_0(a)}{\partial\beta}-\frac{\partial y_0(a)}{\partial\alpha}\frac{\partial y_0(t)}{\partial\beta}, \tag{13.5}$$

  at $\alpha=A$ and $\beta=B$, which is zero if and only if t is conjugate to a.

To find any conjugate points, solve for zeros of the returned expression.

Example

Find a minimum for the functional

$$F(y)=\int_0^{\frac{\pi}{2}}\left(y'(x)^2-x\,y(x)-y(x)^2\right)\mathrm{d}x$$

on $D=\{y\in C^1[0,\pi/2]:y(0)=y(\pi/2)=0\}$. The corresponding Euler-Lagrange equation is:

```
> eq:=euler_lagrange(y'(x)^2-x*y(x)-y(x)^2,y(x))
```

$$\frac{\mathrm{d}^2}{\mathrm{d}x^2}y\left(x\right)=-\frac{x}{2}-y\left(x\right)$$

The general solution is:

```
> y0:=dsolve(eq,x,y)
```

$$c_0\cos x+c_1\sin x-\frac{x}{2}$$

The stationary function depends on two parameters c_0 and c_1 which are fixed by the boundary conditions:

```
> c:=solve([subs(y0,x,0)=0,subs(y0,x,pi/2)=0],[c_0,c_1])
```

$$\left[\left[0,\frac{1}{4}\pi\right]\right]$$

```
> conjugate_equation(y0,[c_0,c_1],c[0],x,0)
```

$$\sin x$$

The above expression obviously has no zeros in $(0, \pi/2]$, hence there are no points conjugate to 0. Since $f_{y'y'} = 2 > 0$, where $f(y, y', x)$ is the integrand in $F(y)$ (the strong Legendre condition), y_0 minimizes F on D. To obtain y_0 explicitly:

```
> subs(y0,[c_0,c_1],c[0])
```

$$\frac{1}{4}\pi \sin x - \frac{x}{2}$$

13.8.6 An example: finding the surface of revolution with minimal area

In this section, you will find the function $y_0 \in D = \{y \in C^1[0,1] : y(0) = 1, y(1) = 2/3\}$ for which the area of the corresponding surface of revolution is minimal.

The area of the surface of revolution is measured by the functional

$$F(y) = 2\,\pi \int_0^1 y(x)\,\sqrt{1 + y'(x)^2}\,\mathrm{d}x.$$

First, set $f(y, y', x) = y(x)\,\sqrt{1 + y'(x)^2}$ and compute the associated Euler-Lagrange equation:

```
> f:=y(x)*sqrt(1+diff(y(x),x)^2):; eq:=euler_lagrange(f)
```

$$\left[-\frac{y\left(x\right)}{\sqrt{\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^{2}+1}} = K_0, \frac{\mathrm{d}^{2}}{\mathrm{d}x^{2}}y\left(x\right) = \frac{\left(\frac{\mathrm{d}}{\mathrm{d}x}y\left(x\right)\right)^{2}+1}{y\left(x\right)}\right]$$

You can obtain the stationary function by finding the general solution of the first equation.

```
> sol:=collect(simplify(dsolve(eq[0],x,y)))
```

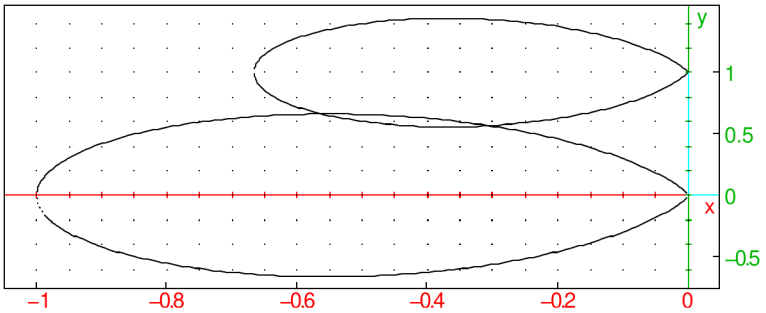
$$\left[-K_0, \frac{K_0\left(-\left(\mathrm{e}^{\frac{x-c_1}{K_0}}\right)^{2}-1\right)}{2\mathrm{e}^{\frac{x-c_1}{K_0}}}\right]$$

(See Section 11.1.18, p. 219.) Obviously the constant solution $-K_0$ is not in D, so set y_0 to be the second element of the above list. That function, which can be written as $y_0(x) = -K_0 \cosh\left(\frac{x-c_1}{K_0}\right)$, is called a *catenary*.

```
> y0,p:=sol[1],[K_0,c_1]:;
```

To find the values of K_0 and c_1 from the boundary conditions, first plot the curves $y_0(0) = 1$ and $y_0(1) = \frac{2}{3}$ for $K_0 \in [-1, 1]$ and $c_1 \in [-1, 2]$ to see where they intersect each other.

```
> eq1,eq2:=subs(y0,x=0)=1,subs(y0,x=1)=2/3:;
  implicitplot([eq1,eq2],K_0=-1..1,c_1=-1..2)
```



Observe that there are exactly two catenaries satisfying the Euler-Lagrange necessary conditions and the given boundary conditions: the first with $K_0 \approx -0.5$ and $c_1 \approx 0.6$ and the second with $K_0 \approx -0.3$ and $c_1 \approx 0.5$. You can obtain the values of these constants more precisely by using `fsolve`.

```
> p1,p2:=fsolve([eq1,eq2],p,[-0.5,0.6]),fsolve([eq1,eq2],p,[-0.3,0.5])
```

$$[-0.56237423894, 0.662588703113], [-0.30613431407, 0.567138261119]$$

You can check, for each catenary, that $f_{y'y'}(x, y_k, y_k') > 0$ holds for $k = 1, 2$.

```
> y1,y2:=subs(y0,p,p1),subs(y0,p,p2):;
  D2f:=diff(f,diff(y(x),x),2):;
  solve([eval(exact(subs(D2f,y=y1,y(x)=y1)))<=0,x>=0,x<=1],x);
  solve([eval(exact(subs(D2f,y=y2,y(x)=y2)))<=0,x>=0,x<=1],x)
```

$$[], []$$

You can conclude that the strong Legendre condition is satisfied in both cases, so you can proceed by attempting to find the points conjugate to 0 for each catenary. The function y_0 depends on two parameters, so use `conjugate_equation` to find these points easily.

```
> fsolve(conjugate_equation(y0,p,p1,x,0)=0,x=0..1);
  fsolve(conjugate_equation(y0,p,p2,x,0)=0,x=0..1)
```

$$[0.0], [0.0, 0.799514772606]$$

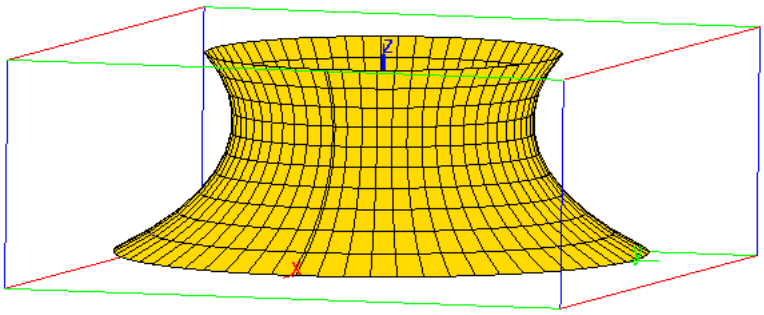
You can conclude that there are no points conjugate to 0 in $(0, 1]$ for the catenary y_1, so it minimizes the functional F. However, for the other catenary there is a conjugate point in the relevant interval, therefore y_2 is not a minimizer. You can verify this by computing surface areas for catenaries y_1 and y_2:

```
> int(y1*sqrt(1+diff(y1,x)^2),x=0..1); int(y2*sqrt(1+diff(y2,x)^2),x=0..1)
```

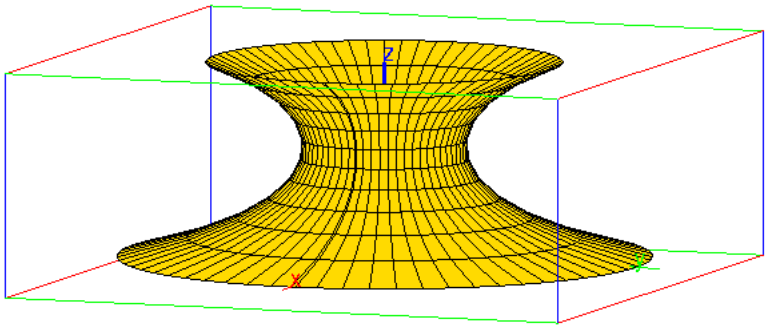
$$0.81396915825, 0.826468466845$$

You can see that the surface formed by rotating the curve y_1 is indeed smaller than the area of the surface formed by rotating the curve y_2. Finally, you can visualize both surfaces for convenience.

```
> plot3d([y1*cos(t),y1*sin(t),x],x=0..1,t=0..2*pi,display=gold+filled)
```



```
> plot3d([y2*cos(t),y2*sin(t),x],x=0..1,t=0..2*pi,display=gold+filled)
```



14 Vectors, matrices, and tables

14.1 Functions for vectors

14.1.1 Norms of a vector

There are different norms for vectors in $\mathbb{R}^n$, and Xcas has different commands for computing them. Let $L = [a_1, \ldots, a_n]$ be a list.

`maxnorm(`L`)` returns the ℓ^∞ norm of L, namely $\max\{|a_1|, |a_2|, \ldots, |a_n|\}$. For example:

```
> maxnorm([3,-4,2])
```

$$4$$

Indeed, $4 = \max\{|3|, |-4|, |2|\}$.

`l1norm(`L`)` returns the ℓ^1 norm of L, namely $|a_1| + |a_2| + \cdots + |a_n|$. For example:

```
> l1norm([3,-4,2])
```

$$9$$

Indeed, $9 = |3| + |-4| + |2|$.

`norm(`L`)` and `l2norm(`L`)` both return the ℓ^2 norm of L, namely $\sqrt{|a_1|^2 + |a_2|^2 + \cdots + |a_n|^2}$. For example:

```
> norm([3,-4,2])
```

$$\sqrt{29}$$

Indeed, $29 = |3|^2 + |-4|^2 + |2|^2$.

14.1.2 Normalizing a vector

The `normalize` or `unitV` command finds the unit vector in the direction of a given vector.

- `normalize` takes V, a vector (list).
- `normalize(`V`)` normalizes this vector for the ℓ^2 norm (the square root of the sum of the squares of its coordinates).

Example

```
> normalize([3,4,5])
```

$$\left[\frac{3}{5\sqrt{2}}, \frac{4}{5\sqrt{2}}, \frac{5}{5\sqrt{2}}\right]$$

14.1.3 Term by term sum of two lists

The infixed operators `+` and `.+` as well as the prefixed operator `'+'` return the term by term sum of two lists. If the two lists do not have the same size, the smaller list is padded with zeros.

Note the difference with sequences: if the infixed operator `+` or the prefixed operator `'+'` is applied to two sequences, it merges the sequences, hence return the sum of all the terms of the two sequences.

Examples

```
> [1,2,3]+[4,3,5]
```

or:

```
> [1,2,3].+[4,3,5]
```

or:

```
> '+'([1,2,3],[4,3,5])
```

$$[5, 5, 8]$$

```
> [1,2,3,4,5,6]+[4,3,5]
```

or:

```
> [1,2,3,4,5,6].+[4,3,5]
```

or:

```
> '+'([[1,2,3,4,5,6],[4,3,5]])
```

$$[5, 5, 8, 4, 5, 6]$$

Remark. When the operator `+` is prefixed, it should be quoted (`'+'`).

14.1.4 Term by term difference of two lists

The infixed operators `-` and `.-` as well as the prefixed operator `'-'` return the term by term difference of two lists. If the two lists do not have the same size, the smaller list is padded with zeros.

Example

```
> [1,2,3]-[4,3,5]
```

or:

```
> [1,2,3].-[4,3,5]
```

or:

```
> '-'([1,2,3],[4,3,5])
```

or:

```
> '-'([[1,2,3],[4,3,5]])
```

$$[-3, -1, -2]$$

Remark. When the operator `-` is prefixed, it should be quoted (`'-'`).

14.1.5 Term by term product of two lists

The infixed operator `.*` returns the term by term product of two lists of the same size.

Example

```
> [1,2,3].*[4,3,5]
```

$$[4, 6, 15]$$

14.1.6 Term by term quotient of two lists

The infixed operator `./` returns the term by term quotient of two lists of the same size.

Example

```
> [1,2,3]./[4,3,5]
```

$$\left[\frac{1}{4}, \frac{2}{3}, \frac{3}{5}\right]$$

14.1.7 Scalar product

The `dot` command finds the dot product of two vectors.

`dotP`, `dotprod`, `scalar_product`, and `scalarProduct` are synonyms for `dot`. The infixed operator `*` and its prefixed form `'*'` will also find dot products.

- `dot` takes two arguments: V_1 and V_2, two lists (vectors) of the same size.
- `dot`(V_1, V_2) returns the dot product of V_1 and V_2.

Example

```
> dot([1,2,3],[4,3,5])
```

or:

```
> scalar_product([1,2,3],[4,3,5])
```

or:

```
> [1,2,3]*[4,3,5]
```

or:

```
> '*'([1,2,3],[4,3,5])
```

$$25$$

Indeed, $25 = 1 \cdot 4 + 2 \cdot 3 + 3 \cdot 5$.

Note that `*` may be used to find the product of two polynomials represented as list of their coefficients, but to avoid ambiguity, the polynomial lists must be `poly1[...]`.

14.1.8 Cross product

The `cross` or `crossP` or `crossproduct` command finds the cross product of two vectors.

- `cross` takes two arguments: V_1 and V_2, two vectors of length 3.
- `cross`(V_1, V_2) returns the cross product of V_1 and V_2.

Example

```
> cross([1,2,3],[4,3,2])
```

$$[-5, 10, -5]$$

Indeed, $-5 = 2 \cdot 2 - 3 \cdot 3$, $10 = -1 \cdot 2 + 4 \cdot 3$, $-5 = 1 \cdot 3 - 2 \cdot 4$.

14.2 Matrices

14.2.1 Entering matrices

A matrix is represented by a list of lists, all having the same size.

Examples

```
> [[1,2,3],[4,5,6]]
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

You can give a matrix a name with assignment.

```
> A:=[[1,2,6],[3,4,8],[1,0,1]]
```

$$\begin{bmatrix} 1 & 2 & 6 \\ 3 & 4 & 8 \\ 1 & 0 & 1 \end{bmatrix}$$

14.2.2 Special matrices

Identity matrix. The `idn` or `identity` command creates identity matrices.

- `idn` takes n, a positive integer or
  A, a square matrix.
- `idn(`n`)` returns the $n \times n$ identity matrix.
- `idn(`A`)` returns the identity matrix the same size as A.

Examples

```
> idn(3)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
> idn([[2,3],[4,5]])
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Zero matrix. The `newMat` command creates a matrix of all zeros.

- `newMat` takes two arguments: n and p, two positive integers. `newMat(`n, p`)` returns the $n \times p$ zero matrix.

Example

```
> newMat(4,4)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Diagonals of matrices and diagonal matrices. The `diag` or `BlockDiagonal` command either creates a diagonal matrix or finds the diagonal elements of an existing matrix or creates diagonal matrices.

- `diag` takes L, a list or a square matrix.
- `diag(`L`)` (for a list L) returns the diagonal matrix with the entries of L on the diagonal.
- `diag(`L`)` (for a matrix L) returns a list consisting of the diagonal elements of L.

Examples

```
> diag([1,4])
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

```
> diag([[1,2],[3,4]])
```

$$[1, 4]$$

Jordan blocks. The `JordanBlock` command creates a Jordan block, i.e. a square matrix with the same value for all diagonal elements, ones just above the diagonal, and zeros everywhere else.

- `JordanBlock` takes two arguments:
  - a, an expression.
  - n, a positive integer.

  `JordanBlock(`a, n`)` returns the $n \times n$ matrix with as on the principal diagonal, ones above this diagonal and zeros everywhere else.

Example

```
> JordanBlock(7,3)
```

$$\begin{bmatrix} 7 & 1 & 0 \\ 0 & 7 & 1 \\ 0 & 0 & 7 \end{bmatrix}$$

Hilbert matrix. A Hilbert matrix is a square matrix whose element in the ith row and jth column (recall that the indices are zero-based) is

$$a_{j,k} = \frac{1}{j+k+1}.$$

The `hilbert` command creates Hilbert matrices. See Section 21.4.6, p. 592 for other uses of `hilbert`.

- `hilbert` takes n, a positive integer.
- `hilbert(`n`)` returns the $n \times n$ Hilbert matrix.

Examples

```
> hilbert(4)
```

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix}$$

Hilbert matrix is known for being regular but severely ill-conditioned. For example:

```
> cond(hilbert(10))
```

$$3.53654789112 \times 10^{13}$$

Vandermonde matrix. A Vandermonde matrix is a square matrix where each row starts with a 1 and is in geometric progression. The `vandermonde` command finds a Vandermonde matrix.

- `vandermonde` takes $X = [x_0, \ldots, x_{n-1}]$, a vector.
- `vandermonde(`X`)` returns the corresponding Vandermonde matrix; namely, the kth row of the matrix is the vector whose components are x_i^k for $i = 0, \ldots, n-1$ and $k = 0, \ldots, n-1$.

Remark. Remember that the indices of the rows and columns begin at 0 with Xcas.

Example

If `a` is symbolic else enter `purge(a)` before:

```
> vandermonde([a,2,3])
```

$$\begin{bmatrix} 1 & a & a^2 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{bmatrix}$$

14.2.3 Combining matrices

Making a matrix with a list of matrices. The `blockmatrix` command combines several matrices into one larger matrix.

- `blockmatrix` takes three arguments:
  - m and n, two positive integers.
  - L, a list of mn matrices such that the first m matrices have the same number of rows; the next m matrices have the same number of rows, etc; and the number of columns in each group of m matrices is the same (for example, all the matrices in L could have the same dimension), so that the n groups of m matrices can be stacked above each other to form a larger matrix.
- `blockmatrix(`m, n, L`)` returns the larger matrix formed by the matrices in L by putting each group of m matrices next to each other, and stacking the resulting n matrices on top of each other.

Examples

```
> blockmatrix(2,3,[idn(2),idn(2),idn(2),idn(2),idn(2),idn(2)])
```

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

```
> blockmatrix(3,2,[idn(2),idn(2),idn(2),idn(2),idn(2),idn(2)])
```

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

```
> blockmatrix(2,2,[idn(2),newMat(2,3),newMat(3,2),idn(3)])
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

```
> blockmatrix(3,2,[idn(1),newMat(1,4),newMat(2,3),idn(2),newMat(1,2),[[1,1,1]]])
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

```
> A:=[[1,1],[1,1]];B:=[[1],[1]]:;
  blockmatrix(2,3,[2*A,3*A,4*A,5*B,newMat(2,4),6*B])
```

$$\begin{bmatrix} 2 & 2 & 3 & 3 & 4 & 4 \\ 2 & 2 & 3 & 3 & 4 & 4 \\ 5 & 0 & 0 & 0 & 0 & 6 \\ 5 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Making a matrix by concatenating columns of two matrices. The `semi_augment` command concatenates two matrices with the same number of columns.

- `semi_augment` takes two arguments: A and B, two matrices with the same number of columns.
- `semi_augment`(A, B) returns the matrix which has the rows of A followed by the rows of B.

Examples

```
> semi_augment([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

```
> semi_augment([[3,4,2]],[[1,2,4]])
```

$$\begin{bmatrix} 3 & 4 & 2 \\ 1 & 2 & 4 \end{bmatrix}$$

Note the difference with `concat`.

```
> concat([[3,4,2]],[[1,2,4]]
```

$$\begin{bmatrix} 3 & 4 & 2 & 1 & 2 & 4 \end{bmatrix}$$

Indeed, when the two matrices A and B have the same dimension, `concat` makes a matrix with the same number of rows as A and B by gluing them side by side.

```
> concat([[3,4],[2,1],[0,1]],[[1,2],[4,5]]
```

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

But input:

```
> concat([[3,4],[2,1]],[[1,2],[4,5]]
```

$$\begin{bmatrix} 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 5 \end{bmatrix}$$

Making a matrix by gluing two matrices together. The `augment` or `concat` command glues two matrices, either side by side or one on top of the other.

- `augment` has two arguments: A and B, two matrices with the same number of rows or the same number of columns.
- `augment`(A, B) returns the matrix consisting of:
  - if A and B have the same number of rows, then the matrix being returned consists of the columns of A followed by the columns of B; in other words, A and B are glued side by side.
  - if A and B do not have the same number of rows but have the same number of columns, then the matrix being returned consists of the rows of A followed by the rows of B; in other words, A and B are glued one on top of the other.

Note that if A and B have the same dimension, then `augment`(A, B) will return a matrix with the same number of rows as A and B by horizontal gluing. In that case, if you want to combine them by vertical gluing, you must use `semi_augment`(A, B).

Examples

```
> augment([[3,4,5],[2,1,0]],[[1,2],[4,5]])
```

$$\begin{bmatrix} 3 & 4 & 5 & 1 & 2 \\ 2 & 1 & 0 & 4 & 5 \end{bmatrix}$$

```
> augment([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

```
> augment([[3,4,2]],[[1,2,4]]
```

$$\begin{bmatrix} 3 & 4 & 2 & 1 & 2 & 4 \end{bmatrix}$$

Appending a column to a matrix. The `border` command adds a column to a matrix.

- `border` takes two arguments:
  - A, a matrix.
  - b, a list whose length equals the number of rows of A.
- `border`(A, L) returns a matrix equal to A with the transpose of L forming an additional column to the right. Equivalent commands are `tran([op(tran(`A`)),`b`])` and `tran(append(tran(`A`),`b`))`.

Examples

```
> border([[1,2,4],[3,4,5]],[6,7])
```

$$\begin{bmatrix} 1 & 2 & 4 & 6 \\ 3 & 4 & 5 & 7 \end{bmatrix}$$

```
> border([[1,2,3,4],[4,5,6,8],[7,8,9,10]],[1,3,5])
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 1 \\ 4 & 5 & 6 & 8 & 3 \\ 7 & 8 & 9 & 10 & 5 \end{bmatrix}$$

14.2.4 Creating a matrix with a formula or function

Use a function or a formula to specify the elements of a matrix with the `makemat` or `matrix` command.

- `makemat` takes three arguments:
  - f, a function of two variables `j` and `k` which returns the value of $a_{j,k}$, the element at row index `j` and column index `k` of the resulting matrix.
  - n and p, two positive integers.
- `makemat`(f, n, p) returns the $n \times p$ matrix $A = [a_{jk}]$ with $a_{jk} = f(j, k)$ for $j = 1 \ldots, n$ and $k = 1, \ldots, p$.

The `matrix` command can be used similarly, but note that the arguments are given in a different order and the indices start at 1. (See Section 14.4.2, p. 349 for other uses of `matrix`.)

- `matrix` takes two mandatory arguments and one optional argument:
  - n and p, two integers.
  - Optionally, f, a function of two variables `j` and `k` which should return the value of $a_{j,k}$, the element at row index `j` and column index `k` of the resulting matrix.

- `matrix`(n,p) returns the $n \times p$ matrix consisting of all zeros.
- `matrix`(n,p,f) returns the $n \times p$ matrix $A = [a_{jk}]$ with $a_{jk} = f(j,k)$ for $j = 1,\ldots,n$ and $k = 1,\ldots,p$.

Examples

```
> makemat((j,k)->j+k,4,3)
```

or:

```
> h(j,k):=j+k:;
  makemat(h,4,3)
```

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

Note that the indices are counted starting from 0.

```
> matrix(2,3)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
> matrix(4,3,(j,k)->j+k)
```

or:

```
> h(j,k):=j+k:;
  matrix(4,3,h)
```

$$\begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

14.2.5 Getting the parts of a matrix

Accessing rows of a matrix. The rows of a matrix are the elements of a list, and can be accessed with indices using the postfix `[...]` or the prefix `at` (see Section 6.1.6, p. 71). For example:

```
> A:=[[1,2,6],[3,4,8],[1,0,1]]
```

then:

```
> A[0]
```

or:

```
> at(A,0)
```

$$[1,2,6]$$

To extract a column of a matrix, you can first turn the columns into rows with `transpose` (see Section 15.1.1, p. 355), then extract the row as above. For example:

```
> tran(A)[1]
```

or:

```
> at(tran(A),1)
```

$$[2,4,0]$$

Individual elements are simply elements of the rows. For example:

```
> A[0][1]
```

$$2$$

This can be abbreviated by listing the row and column separated by a comma:

```
> A[0,1]
```

or:

```
> at(A,[0,1])
```

$$2$$

The indexing begins with 0; you can have the indices start with 1 by enclosing them in double brackets.

```
> A[[1,2]]
```

$$2$$

Use a range (see Section 6.5.1, p. 98) of indices to get submatrices. Some examples:

```
> A[1,0..2]
```

$$[3, 4, 8]$$

```
> A[0..2,1]
```

$$[2, 4, 0]$$

```
> A[0..2,1..2]
```

$$\begin{bmatrix} 2 & 6 \\ 4 & 8 \\ 0 & 1 \end{bmatrix}$$

```
> A[0..1,1..2]
```

$$\begin{bmatrix} 2 & 6 \\ 4 & 8 \end{bmatrix}$$

This gives you another way to extract a full column, by specifying all the rows as an index interval.

```
> A[0..2,1]
```

$$[2, 4, 0]$$

Recall that an index of -1 returns the last element of a list, an index of -2 the second to last element, etc.

```
> A[-1]
```

$$[1, 0, 1]$$

```
> A[1,-1]
```

$$8$$

Extracting rows or columns of a matrix (Maple compatibility). The `row` (resp. `col`) command extracts one or several rows (resp. columns) of a matrix.

- `row` takes two arguments:
  - A, a matrix.
  - r, a row index or a range $n_1, \ldots, n_2$.
- `row`(A, r) returns the row or sequence of rows given by r.
- `col` takes two arguments:
  - A, a matrix.
  - c, a column index or a range $n_1, \ldots, n_2$.
- `col`(A, c) returns the column or sequence of columns given by c.

Examples

```
> row([[1,2,3],[4,5,6],[7,8,9]],1)
```

$$[4, 5, 6]$$

```
> row([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

$$[1, 2, 3], [4, 5, 6]$$

```
> col([[1,2,3],[4,5,6],[7,8,9]],1)
```

$$[2, 5, 8]$$

```
> col([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

$$[1, 4, 7], [2, 5, 8]$$

Extracting a sub-matrix of a matrix (TI compatibility). The `subMat` command finds submatrices of a matrix.

- `subMat` takes one mandatory argument and four optional arguments:
  - A, a matrix.
  - Optionally, r_1, an integer, the row index for the beginning of the submatrix (by default, $r_1 = 0$).
  - Optionally, c_1, an integer, the column index for the beginning of the submatrix (by default, $c_1 = 0$).
  - Optionally, r_2, an integer, the row index for the end of the submatrix (by default, r_2 equals one less than the number of rows of A).
  - Optionally, c_2, an integer, the column index for the end of the submatrix (by default, c_1 equals one less than the number of columns of A).
- `subMat`$(A \langle, r_1, c_1, r_2, c_2\rangle)$ returns the sub-matrix of A from position (r_1, c_1) to (r_2, c_2).

Examples

```
> A:=[[3,4,5],[1,2,6]]
```

$$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

```
> subMat(A,0,1,1,2)
```

$$\begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

```
> subMat(A,0,1,1,1)
```

$$\begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

```
> subMat(A,1)
```

or:

```
> subMat(A,1,0)
```

or:

```
> subMat(A,1,0,1)
```

or:

```
> subMat(A,1,0,1,2)
```

$$\begin{bmatrix} 1 & 2 & 6 \end{bmatrix}$$

14.2.6 Modifying matrices

Modifying matrix elements by assignment. You can change the elements of a named matrix by assignment (see Section 3.3.2, p. 36). For example:

```
> A:=[[1,2,6],[3,4,8],[1,0,1]]
```

then:

```
> A[0,1]:=5:; A
```

$$\begin{bmatrix} 1 & 5 & 6 \\ 3 & 4 & 8 \\ 1 & 0 & 1 \end{bmatrix}$$

Recall that the elements are indexed starting at 0, using double brackets allows you to use indices starting at 1.

```
> A[[1,2]]:=7:; A
```

$$\begin{bmatrix} 1 & 7 & 6 \\ 3 & 4 & 8 \\ 1 & 0 & 1 \end{bmatrix}$$

Use assignment to change several entries of a matrix at one. For example, create a diagonal matrix with a diagonal of `[1,2,3]`:

```
> M:=matrix(3,3)
```

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
> M[0..2,0..2]:=[1,2,3]
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

To make the last column `[4,5,6]`:

```
> M[0..2,2]:=[4,5,6]
```

$$\begin{bmatrix} 1 & 0 & 4 \\ 0 & 2 & 5 \\ 0 & 0 & 6 \end{bmatrix}$$

Modifying matrix elements by reference. When you change an element of a matrix with the `:=` assignment, a new copy of the matrix is created with the modified element. Particularly for large matrices, it is more efficient to use the `=<` assignment (see Section 3.3.3, p. 37), which will change the element of the matrix without making a copy.

For example, enter:

```
> A:=[[4,5],[2,6]]
```

The following commands will all return the matrix `A` with the element in the second row, first column, changed to 3.

```
> A[1,0]:=3
```

or:

```
> A[1,0]=<3
```

or:

```
> A[[2,1]]:=3
```

or:

```
> A[[2,1]]=<3
```

then:

```
> A
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

You can change larger parts of a matrix simultaneously. For example, the following commands will change the second row to `[3,7]`:

```
> A[1]:=[3,7]
```

or:

```
> A[1]=<[3,7]
```

or:

```
> A[[2]]:=[3,7]
```

or:

```
> A[[2]]=<[3,7]
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 7 \end{bmatrix}$$

The =< assignment must be used carefully, since it not only modifies a matrix A, it modifies all objects pointing to A. In a program, initialization should contain a line like A:=copy(B) (see Section 3.3.4, p. 37) so modifications done on A do not affect B, and modifications done on B do not affect A.

```
> B:=[[4,5],[2,6]]
```

then:

```
> A:=B
```

or:

```
> A=<B
```

creates two matrices equal to $\begin{bmatrix}4 & 5\\2 & 6\end{bmatrix}$.

```
> A[1]=<[3,7]
```

or:

```
> B[1]=<[3,7]
```

transforms both A and B to $\begin{bmatrix}4 & 5\\3 & 7\end{bmatrix}$. On the other hand, creating A and B with:

```
> B:=[[4,5],[2,6]]; A:=copy(B)
```

will again create two matrices equal to $\begin{bmatrix}4 & 5\\2 & 6\end{bmatrix}$. But:

```
> A[1]=<[3,7]
```

will change A to $\begin{bmatrix}4 & 5\\3 & 7\end{bmatrix}$, while B will still be equal to $\begin{bmatrix}4 & 5\\2 & 6\end{bmatrix}$.

Modifying an element or a row of a matrix. The subsop command modifies elements of lists (see Section 6.3.7, p. 81), and so use it to modify elements or rows of matrices. It is used mainly for Maple and MuPAD compatibility, and the argument list is in a different order in Maple mode. Unlike := or =<, it does not require the matrix to be stored in a variable.

Let A be the matrix given by:

```
> A:=[[4,5],[2,6]]
```

We use this variable in the examples that follow. (Recall that the indexing in Xcas mode begins with 0, while in MuPAD and TI modes it begins with 1.)

- To modify an element, subsop takes two arguments:
  - A, a matrix.
  - $[r,c]$=v, an equality between a matrix position (given as a list) and a value.

    The two sides of the equality can also be given as separate arguments.
- subsop(A,$[r,c]$=v) returns the matrix which is the same as A except that the element in row r, column c is now v.
- To modify a row, subsop takes two arguments:
  - A, a matrix.
  - $r = L$, an equality between a row index and a list with the same length as the rows of A.

    The two sides of the equality can also be given as separate arguments.
- subsop($A, r = L$) returns the matrix which is the same as A except that row r is now equal to the list L.

Examples

Input in XCAS mode:

```
> subsop(A,[1,0]=3)
```

or:

```
> subsop(A,[1,0],3)
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

Input in MUPAD or TI mode:

```
> subsop(A,[2,1]=3)
```

or:

```
> subsop(A,[2,1],3)
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

Input in XCAS mode:

```
> subsop(A,1=[3,3])
```

or:

```
> subsop(A,1,[3,3])
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

Input in MUPAD or TI mode:

```
> subsop(A,2=[3,3])
```

or:

```
> subsop(A,2,[3,3])
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

In Maple mode: Recall that the indexing in MAPLE mode is 1-based.

- To modify an element, `subsop` takes two arguments:
  - $[r, c]$=v, an equality between a matrix position (given as a list) and a value.

    The two sides of the equality can also be given as separate arguments.
  - A, a matrix.
- `subsop(`$[r, c]$=v, A`)` returns the matrix which is the same as A except that the element in row r, column c is now v.
- To modify a row, `subsop` takes two arguments:
  - $r = L$, an equality between a row index and a list with the same length as the rows of the second argument A.
  - A, a matrix.
- `subsop(`$r = L, A$`)` returns the matrix which is the same as A except that row r is now equal to the list L.

Examples

Input in MAPLE mode:

```
> subsop([2,1]=3,A)
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

```
> subsop(2=[3,3],A)
```

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

In all modes: If the matrix is stored in a variable, for example with the matrix `A` as above, it is easier to enter `A[1,0]:=3` and `A[1]=[3,3]` to modify `A` as above.

Also, note that `subsop` with a `'n=NULL'` argument deletes row number `n`.

Example

Input in XCAS mode:

```
> subsop(A,'1=NULL')
```

$$\begin{bmatrix} 4 & 5 \end{bmatrix}$$

Removing rows or columns of a matrix. The `delrows` (resp. `delcols`) command removes one or more rows (resp. columns) from a matrix.

- `delrows` takes two arguments:
  - A, a matrix.
  - r, an integer or a range of integers.
- `delrows`(A, r) returns the matrix equal to A with the row(s) given by r removed.
- `delcols` takes two arguments:
  - A, a matrix.
  - c, an integer or a range of integers.
- `delrows`(A, c) returns the matrix equal to A with the column(s) given by c removed.

Examples

```
> delrows([[1,2,3],[4,5,6],[7,8,9]],1)
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{bmatrix}$$

```
> delrows([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

$$\begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$

```
> delcols([[1,2,3],[4,5,6],[7,8,9]],1)
```

$$\begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix}$$

```
> delcols([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

$$\begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}$$

Resizing a matrix or vector. The `REDIM` or `redim` command resizes matrices and vectors.

- For matrices, `REDIM` takes two arguments:
  - A, a matrix.
  - $[m, n]$, a list of two positive integers.
- `REDIM(`A`,[`m,n`])` returns A resized to an $m \times n$ matrix, removing elements (if necessary) to make it smaller and adding zeros (if necessary) to make it larger.
- For vectors, `REDIM` takes two arguments:
  - L, a list.
  - n, a positive integer.
- `REDIM(`L,n`)` returns L resized to a list of length n, removing elements (if necessary) to make it smaller and adding zeros (if necessary) to make it larger.

Examples

```
> REDIM([[4,1,-2],[1,2,-1],[2,1,0]],[5,4])
```

$$\begin{bmatrix} 4 & 1 & -2 & 0 \\ 1 & 2 & -1 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> REDIM([[4,1,-2],[1,2,-1],[2,1,0]],[2,1])
```

$$\begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

```
> REDIM([4,1,-2,1,2,-1],10)
```

$$[4, 1, -2, 1, 2, -1, 0, 0, 0, 0]$$

```
> REDIM([4,1,-2,1,2,-1],3)
```

$$[4, 1, -2]$$

Replacing part of a matrix or vector. The `REPLACE` or `replace` command replaces part of a matrix or vector.

- For matrices, `REPLACE` takes three arguments:
  - A, a matrix.
  - $[m, n]$, a list of two positive integers.
  - B, a matrix.

  `REPLACE(`A`,[`m`,`n`],`B`)` returns the matrix equal to A but with the upper left corner of B placed at row m, column n, replacing the previous elements of A. The matrix B will be shrunk, if necessary, to fit.
- For lists, `REPLACE` takes three arguments:
  - L, a list.
  - n, a positive integer.
  - M, another list.

  `REPLACE(`L`,`n`,`M`)` returns the list equal to L but with the elements beginning at index n replaced by the elements of M, replacing the previous elements of L. The list M will be shrunk, if necessary, to fit.

Examples

```
> REPLACE([[1,2,3],[4,5,6]],[0,1],[[5,6],[7,8]])
```

$$\begin{bmatrix} 1 & 5 & 6 \\ 4 & 7 & 8 \end{bmatrix}$$

```
> REPLACE([[1,2,3],[4,5,6]],[1,2],[[7,8],[9,0]])
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \end{bmatrix}$$

```
> REPLACE([4,1,-2,1,2,-1],2,[10,11])
```

$$[4, 1, 10, 11, 2, -1]$$

```
> REPLACE([4,1,-2,1,2,-1],1,[10,11,13])
```

$$[4, 10, 11, 13, 2, -1]$$

Applying a function to the elements of a matrix. The `apply` command can apply a function to the elements of a matrix. (See Section 6.3.28, p. 92 for other uses of `apply`.)

- `apply` takes three arguments:
  - f, a function of one variable.
  - A, a matrix.
  - `matrix`, the symbol.

  `apply(`f`,`A`,matrix)` returns a matrix whose elements are $f(x)$ for the elements x of A.

Example

```
> apply(x->x^2,[[1,2,3],[4,5,6]],matrix)
```

$$\begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \end{bmatrix}$$

14.3 Functions for matrices

14.3.1 Matrix evaluation

In Maple, `evalm` is used for matrix evaluation. In Xcas, matrices are evaluated by default, the command `evalm` is only available for compatibility (it is equivalent to `eval`, see Section 9.1.1, p. 169).

14.3.2 Addition and subtraction of two matrices

The infixed operator `+` or `.+` (resp. `-` or `.-`) is used for matrix addition (resp. subtraction). Note that `+` and `-` can also be used as prefixed operators, in which case they must be quoted, like `'+'` and `'-'` (see Section 14.1.3, p. 322 and Section 14.1.4, p. 323).

Examples

```
> [[1,2],[3,4]]+[[5,6],[7,8]]
```

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

```
> [[1,2],[3,4]]-[[5,6],[7,8]]
```

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

```
> '+'([[1,2],[3,4]],[[5,6],[7,8]],[[2,2],[3,3]])
```

$$\begin{bmatrix} 8 & 10 \\ 13 & 15 \end{bmatrix}$$

```
> '-'([[1,2],[3,4]],[[5,6],[7,8]])
```

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

14.3.3 Multiplication of two matrices

The infixed operator `*` and `&*` are used for the multiplication of two matrices.

Example

```
> [[1,2],[3,4]]*[[5,6],[7,8]]
```

or:

```
> [[1,2],[3,4]]&*[[5,6],[7,8]]
```

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

14.3.4 Addition of elements of a column of a matrix

The `sum` command (see also Section 13.3.3, p. 285) can add the elements of the columns of a matrix.

- `sum` takes A, a matrix.
- `sum(`A`)` returns the list whose elements are the sum of the elements of each column of the matrix A.

Example

```
> sum([[1,2],[3,4]])
```

$$[4, 6]$$

14.3.5 Cumulated sum of elements of each column of a matrix

The `cumSum` command finds the cumulated sum of each column of a matrix (see also Section 6.3.26, p. 90).

- `cumSum` takes A, a matrix.
- `cumSum(`A`)` returns the matrix whose columns are the cumulated sum of the elements of the corresponding column of the matrix A.

Example

```
> cumSum([[1,2],[3,4],[5,6]])
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 6 \\ 9 & 12 \end{bmatrix}$$

since the cumulated sums of the first column are: 1, $1+3=4$, $1+3+5=9$ and the accumulated sums of the second column are: 2, $2+4=6$, $2+4+6=12$.

14.3.6 Product of elements of each column of a matrix

The `product` command can multiply the elements of the columns of a matrix (see Section 6.3.27, p. 91 for other things `product` can do).

- `product` takes A, a matrix.
- `product(`A`)` returns the list whose elements are the product of the elements of each column of the matrix A.

Example

```
> product([[1,2],[3,4]])
```

$$[3, 8]$$

14.3.7 Power of a matrix

The infixed operator `^` (or `&^`) is used to raise a matrix to an integral power.

Example

```
> [[1,2],[3,4]]^5
```

or:

```
> [[1,2],[3,4]]&^5
```

$$\begin{bmatrix} 1069 & 1558 \\ 2337 & 3406 \end{bmatrix}$$

14.3.8 Hadamard product

The `hadamard` command finds the Hadamard product of two matrices; namely, the term-by-term product of the two matrices.

The `product` command can do the same thing (see also Section 6.3.27, p. 91 for other uses of `product`).

- `hadamard` takes two arguments: A and B, two matrices of the same size.
- `hadamard`(A, B) returns the matrix where each element is the product of the corresponding elements of A and B.

The infixed operator `.*` also finds the Hadamard product, and also works on lists.

Examples

```
> hadamard([[1,2],[3,4]],[[5,6],[7,8]])
```

or:

```
> hadamard([[1,2],[3,4]],[[5,6],[7,8]])
```

or:

```
> [[1,2],[3,4]].*[[5,6],[7,8]]
```

$$\begin{bmatrix} 5 & 12 \\ 21 & 32 \end{bmatrix}$$

```
> [1,2,3,4].*[5,6,7,8]
```

$$[5, 12, 21, 32]$$

14.3.9 Hadamard division

The infixed operator `./` finds the Hadamard quotient of two matrices or lists A and B of the same size; namely, it returns the matrix or the list where each element is the term by term quotient of the corresponding elements of A and B.

Example

```
> [[1,2],[3,4]]./[[5,6],[7,8]]
```

$$\begin{bmatrix} \frac{1}{5} & \frac{1}{3} \\ \frac{3}{7} & \frac{1}{2} \end{bmatrix}$$

14.3.10 Hadamard power

The infixed operator `.^` finds the Hadamard power of a matrix or list A to a real number b; namely, it returns the matrix or the list where each element is the corresponding element of A raised to the bth power.

Example

```
> [[1,2],[3,4]].^2
```

$$\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$$

14.3.11 Elementary row operations

Adding a row to another row. The `rowAdd` command adds one row of a matrix to another row.

- `rowAdd` takes three arguments:
  - A, a matrix.
  - n_1 and n_2, two integers.
- `rowAdd`(A, n_1, n_2) returns the matrix obtained by replacing in A, the row of index $n2$ by the sum of the rows of index $n1$ and $n2$.

Example

```
> rowAdd([[1,2],[3,4]],0,1)
```

$$\begin{bmatrix} 1 & 2 \\ 4 & 6 \end{bmatrix}$$

Multiplying a row by an expression. The `mRow`, `scale` and `SCALE` commands multiply a row of a matrix by an expression.

- `mRow` takes three arguments:
  - *expr*, an expression.
  - A, a matrix.
  - n, an integer.
- `mRow`$(expr, A, n)$ returns the matrix obtained by replacing in A, the row of index n by the product of the row of index n by *expr*.

The `scale` or `SCALE` command is similar to `mRow` except for a different order of arguments.

- `scale` takes three arguments:
  - A, a matrix.
  - *expr*, an expression.
  - n, an integer.
- `scale`$(A, expr, n)$ returns the matrix obtained by replacing in A, the row of index n by the product of the row of index n by *expr*.

Examples

```
> mRow(12,[[1,2],[3,4]],1)
```

$$\begin{bmatrix} 1 & 2 \\ 36 & 48 \end{bmatrix}$$

```
> scale([[1,2],[3,4]],12,1)
```

$$\begin{bmatrix} 1 & 2 \\ 36 & 48 \end{bmatrix}$$

Adding k times a row to another row. The `mRowAdd`, `scaleadd` and `SCALEADD` commands add a multiple of one row of a matrix to another.

- `mRowAdd` takes four arguments:
  - k, a real number.
  - A, a matrix.
  - n_1 and n_2, two integers.
- `mRowAdd`(k, A, n_1, n_2) returns the matrix obtained by replacing in A, the row with index n_2 by the sum of the row with index n_2 and k times the row with index n_1.

The `scaleadd` or `SCALEADD` command is similar `mRowAdd` except for a different order of arguments.

- `scaleadd` takes four arguments:
  - A, a matrix.
  - k, a real number.
  - n_1 and n_2, two integers.
- `scaleadd`(A, k, n_1, n_2) returns the matrix obtained by replacing in A, the row with index n_2 by the sum of the row with index n_2 and k times the row with index n_1.

Examples

```
> mRowAdd(1.1,[[5,7],[3,4],[1,2]],1,2)
```

$$\begin{bmatrix} 5 & 7 \\ 3 & 4 \\ 4.3 & 6.4 \end{bmatrix}$$

```
> scaleadd([[5,7],[3,4],[1,2]],1.1,1,2)
```

$$\begin{bmatrix} 5 & 7 \\ 3 & 4 \\ 4.3 & 6.4 \end{bmatrix}$$

Exchanging two rows. The `rowSwap` or `rowswap` or `swaprow` command switches two rows in a matrix.

- `rowSwap` takes three arguments:
  - A, a matrix.
  - n_1 and n_2, integers.
- `rowSwap`(A, n_1, n_2) returns the matrix obtained by exchanging in A, the row with index n_1 with the row with index n_2.

Example

```
> rowSwap([[1,2],[3,4]],0,1)
```

$$\begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}$$

Exchanging two columns. The `colSwap` or `colswap` or `swapcol` command switches two columns in a matrix.

- `colSwap` takes three arguments:
  - A, a matrix.
  - n_1 and n_2, integers.
- `colSwap`(A, n_1, n_2) returns the matrix obtained by exchanging in A, the column with index n_1 with the column with index n_2.

Example

```
> colSwap([[1,2],[3,4]],0,1)
```

$$\begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$$

14.3.12 Counting elements of a matrix which satisfy a given property

The `count` applies a function to the elements of a matrix or list and adds up the obtained values. In particular, if the function returns a boolean, then `count` returns the number of elements satifying the property that the function tests for.

- `count` takes two arguments:
  - f, a real-valued function.
  - A, a matrix or a list.
- `count`(f, A) returns the sum of the function f applied to the elements of A.

Examples

```
> count(x->x,[[2,12],[45,3],[7,78]])
```

147

Indeed, $2+12+45+3+7+78=147$.

```
> count(x->x<10,[[2,12],[45,3],[7,78]])
```

3

14.3.13 Counting elements equal to a given value

The `count_eq` command counts the number of elements in a matrix which are equal to a given value.

- `count_eq` takes two arguments:
  - a, a value.
  - A, a matrix or a list.
- `count_eq`(a, A) returns the number of elements of A that are equal to a.

Example

```
> count_eq(12,[[2,12,45],[3,7,78]])
```

1

14.3.14 Counting elements smaller than a given value

The `count_inf` command counts the number of elements in a matrix which are less than a given value.

- `count_inf` takes two arguments:
  - a, a real number.
  - A, a matrix or a list of real numbers.
- `count_inf`(a, A) returns the number of elements of A that are strictly less than a.

Example

```
> count_inf(12,[2,12,45,3,7,78])
```

3

14.3.15 Counting elements greater than a given value

The `count_sup` command counts the number of elements in a matrix which are greater than a given value.

- `count_sup` takes two arguments:
  - a, a real number.
  - A, a matrix or a list of real numbers.
- `count_sup`(a, A) returns the number of elements of A that are strictly greater than a.

Example

```
> count_sup(12,[[2,12,45],[3,7,78]])
```

2

14.3.16 Dimension of a matrix

The `dim` command finds the dimension of a matrix.

- `dim` takes A, a matrix
- `dim`(A) returns a list of the number of rows and columns of the matrix A.

Example

```
> dim([[1,2,3],[3,4,5]])
```

$[2, 3]$

14.3.17 Number of rows

The `rowdim` or `rowDim` or `nrows` command finds the number of rows of a matrix.

- `rowdim` takes A, a matrix.
- `rowdim`(A) returns the number of rows of the matrix A.

Example

```
> rowdim([[1,2,3],[3,4,5]])
```

2

14.3.18 Number of columns

The `coldim` or `colDim` or `ncols` command finds the number of columns of a matrix.

- `coldim` takes A, a matrix.
- `coldim`(A) returns the number of columns of the matrix A.

Example

```
> coldim([[1,2,3],[3,4,5]])
```

3

14.4 Sparse matrices

14.4.1 Tables

A *table* is a map (associative container) used to store information associated to indices which are much more general than integers, such as strings or sequences. For example, use one to store a table of phone numbers indexed by names.

In Xcas, the indices in a table may be any kind of Xcas objects. Access is done by a binary search algorithm, where the sorting function first sorts by type and then uses an order for each type (e.g. $<$ for numeric types, lexicographic order for strings, etc.)

The `table` command creates a table.

- `table` takes *seq*, a list or sequence of equalities of the form *key*=*value*.
- `table(`*seq*`)` returns a table. The elements of the table can be retrieved using index bracket notation. If T is the name of the table, then T`[`*key*`]` returns *value*.

Tables can also be created and the elements of a table can be changed by using the `:=` assignment.

- If `T` is a symbolic variable, then `T[`*key*`]:=`*value* creates a table `T` with one element.
- If n is an integer, then the assignment `T[`n`]:=`*obj* will do the following:
  - If the variable `T` was assigned to a list or a sequence, then the nth element of `T` is modified.
  - If the variable `T` was not assigned, a table `T` is created with one entry (corresponding to the index n). Note that after the assignment `T` is not a list, despite the fact that n is an integer.

Example

```
> T:=table(3=-10,"a"=10,"b"=20,"c"=30,"d"=40):;
> T["b"]
```

20

```
> T[3]
```

-10

```
> T[3]:=-15:;
  T[3]
```

-15

14.4.2 Defining sparse matrices

A matrix is *sparse* if most of its elements are 0. To specify a sparse matrix, it is easier to define the non-zero elements. This can be done with a table (see Section 14.4.1, p. 349). The `matrix` command (see Section 14.2.4, p. 330) or the `convert` command (see Section 10.1.10, p. 195) can then turn the table into a matrix.

Example

First, define the non-zero elements.

```
> A:=table((0,0)=1,(1,1)=2,(2,2)=3,(3,3)=4,(4,4)=5)
```

or:

```
> purge(A);
  A[0..4,0..4]:=[1,2,3,4,5]
```

Key	Value
$(0,0)$	1
$(1,1)$	2
$(2,2)$	3
$(3,3)$	4
$(4,4)$	5

This table can be converted to a matrix with either the `convert` command or the `matrix` command.

```
> a:=convert(A,array)
```

or:

```
> a:=matrix(A)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

14.4.3 Operations on sparse matrices

All matrix operations can be done on tables that are used to define sparse matrices.

Example

Create some sparse matrices.

```
> purge(A):;
  A[0..2,0..2]:=[1,2,3]
```

Key	Value
$(0,0)$	1
$(1,1)$	2
$(2,2)$	3

```
> purge(B):;
  B[0..1,1..2]:=[1,2]:;
  B[0..2,0]:=5
```

Key	Value
$(0,0)$	5
$(0,1)$	1
$(1,0)$	5
$(1,2)$	2
$(2,0)$	5

The usual operations will work on `A` and `B`.

```
> A+B
```

Key	Value
$(0,0)$	6
$(0,1)$	1
$(1,0)$	5
$(1,1)$	2
$(1,2)$	2
$(2,0)$	5
$(2,2)$	3

```
> A*B
```

Key	Value
$(0,0)$	5
$(0,1)$	1
$(1,0)$	10
$(1,2)$	4
$(2,0)$	15

```
> 2*A
```

Key	Value
$(0,0)$	2
$(1,1)$	4
$(2,2)$	6

14.5 Statistics on lists and matrices

14.5.1 Lists

The functions described here can be used if a statistics series is contained in a list. See Section 14.5.2, p. 353 for statistics on matrices and Section 20, p. 496 for more general statistics.

Let L be a list.

- `mean(`L`)` computes the arithmetic mean of a list.
- `stddev(`L`)` computes the standard deviation of a population, for the population L.
- `stddevp(`L`)` computes an unbiased estimate of the standard deviation of the population for the sample L. The following relation holds:

$$\texttt{stddevp}(L)^2 = \frac{\texttt{size}(L) \cdot \texttt{stddev}(L)^2}{\texttt{size}(L) - 1}$$

- `variance(`L`)` computes the variance of L, which is the square of `stddevp(`L`)`.
- `median(`L`)` computes the median of L.
- `quantile(`L, d`)` computes the deciles of L, where d is the decile.
- `quartiles(`L`)` returns a list consisting of the minimum, the first quartile, the median, the third quartile and the maximum of L.
- `boxwhisker(`L`)` draws the whisker box of a statistics series stored in L.

Examples

```
> mean([3,4,2])
```

$$3$$

```
> stddev([3,4,2])
```

$$\frac{\sqrt{6}}{3}$$

```
> stddevp([3,4,2])
```

$$1$$

```
> variance([3,4,2])
```

$$\frac{2}{3}$$

```
> median([0,1,3,4,2,5,6])
```

$$3.0$$

To obtain the first quartile:

```
> quantile([0,1,3,4,2,5,6],0.25)
```

$$1.0$$

To obtain the median:

```
> quantile([0,1,3,4,2,5,6],0.5)
```

$$3.0$$

To obtain the third quantile:

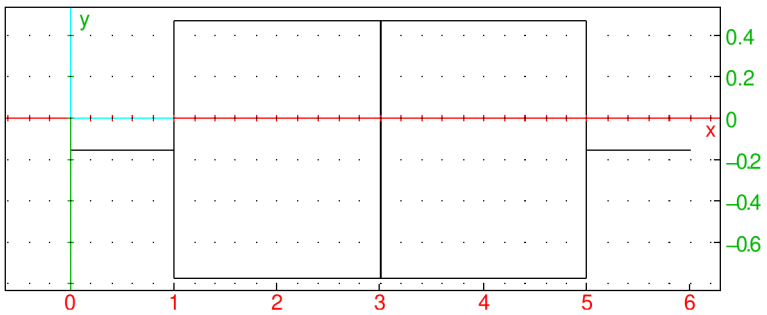
```
> quantile([0,1,3,4,2,5,6],0.75)
```

$$5.0$$

```
> quartiles([0,1,3,4,2,5,6])
```

$$\begin{bmatrix} 0.0 \\ 1.0 \\ 3.0 \\ 5.0 \\ 6.0 \end{bmatrix}$$

```
> boxwhisker([0,1,3,4,2,5,6])
```



14.5.2 Matrices

The functions described in Section 14.5.1, p. 351 can also find finds statistics for the columns of a matrix.

Let A be a matrix.

- `mean(`A`)` computes the arithmetic means of the columns of the matrix A.
- `stddev(`A computes the standard deviations for the populations given by the columns of A.
- `stddevp(`A`)` computes the unbiased estimates of the standard deviations of the populations for the samples given by the columns of A.
- `variance(`A`)` computes the variances of the columns of A.
- `median(`A`)` computes the medians of the columns of A.
- `quantile(`A, d`)` computes the deciles of the columns of A, where d is the decile.
- `quartiles(`A`)` returns a matrix where each column consists of the minimum, the first quartile, the median, the third quartile and the maximum of the corresponding column of A.

  The output is a matrix, its first row is the minima of each column, its second row is the first quartiles of each column, its third row the medians of each column, its fourth row the third quartiles of each column and its last row the maxima of each column.
- `boxwhisker(`A`)` draws the whisker box of the statistics series stored in the columns of A.

Examples

With

```
> A:=[[3,4,2],[1,2,6]]
```

input:

```
> mean(A)
```

$$[2, 3, 4]$$

```
> stddev(A)
```

$$[1, 1, 2]$$

```
> stddevp(A)
```

$$\left[\sqrt{2}, \sqrt{2}, 2\sqrt{2}\right]$$

```
> variance(A)
```

$$[1, 1, 4]$$

With

```
> B:=[[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],[1,3,4,2,5,6,0],
      [3,4,2,5,6,0,1],[4,2,5,6,0,1,3],[2,5,6,0,1,3,4]]
```

input:

```
> median(B)
```

$$[2.0, 2.0, 3.0, 3.0, 2.0, 2.0, 3.0]$$

To obtain the first quartiles of the columns:

```
> quantile(B,0.25)
```

$$[1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0]$$

To obtain the third quartiles of the columns:

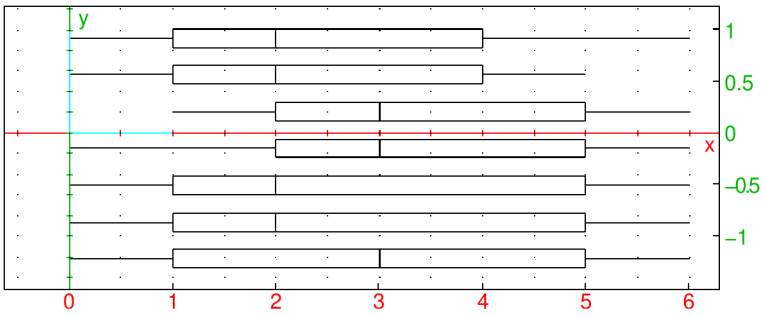
```
> quantile(B,0.75)
```

$$[4.0, 4.0, 5.0, 5.0, 5.0, 5.0, 5.0]$$

```
> quartiles(B)
```

$$\begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 2.0 & 2.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 3.0 & 3.0 & 2.0 & 2.0 & 3.0 \\ 4.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\ 6.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 & 6.0 \end{bmatrix}$$

```
> boxwhisker(B)
```



15 Linear algebra

15.1 Basic matrix operations

15.1.1 Transpose of a matrix

The `tran` or `transpose` command finds the transpose of a matrix.

- `tran` takes A, a matrix.
- `tran`(A) returns the transpose matrix A^T of A.

Example

```
> tran([[1,2],[3,4]])
```

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

15.1.2 Inverse of a matrix

The `inv` command finds the inverse of a matrix.

- `inv` takes A, a matrix.
- `inv`(A) returns the inverse matrix A^{-1} of A.

Note that `1/`A and A`^-1` are two alternative ways of computing the inverse of A.

Example

```
> inv([[1,2],[3,4]])
```

or:

```
> 1/[[1,2],[3,4]]
```

or:

```
> A:=[[1,2],[3,4]];1/A
```

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

15.1.3 Trace of a matrix

The *trace* of a square matrix is the sum of the diagonal elements. The `trace` command finds the trace of a matrix.

- `trace` takes A, a matrix.
- `trace`(A) returns the trace $\mathrm{tr}(A)$ of the matrix A.

Example

```
> trace([[1,2],[3,4]])
```

5

15.1.4 Determinant of a matrix

The `det` and `det_minor` commands compute the determinant of a matrix.

- `det` takes one mandatory argument and one optional argument:
  - A, a matrix.
  - Optionally, *method*, which determines how the determinant will be computed and can be one of:
    - `lagrange` When the matrix elements are polynomials or rational functions, this method computes the determinant by evaluating the elements and using Lagrange interpolation.
    - `rational_det` This method uses Gaussian elimination without converting to to the internal format for fractions.
    - `bareiss` This uses the Gauss-Bareiss algorithm.
    - `linsolve` This uses the p-adic algorithm for matrices with integer coefficients.
    - `minor_det` This uses expansion by minor determinants, which requires 2^n operations, but can still be faster for average sized matrices (up to about $n = 20$).
- `det(`A $\langle$, *method*$\rangle$`)` returns the determinant $\det(A)$ of the matrix A.

Examples

```
> det([[1,2],[3,4]])
```

-2

```
> det(idn(3))
```

1

The `det_minor` command finds the determinant of a matrix by expanding the determinant using Laplace's algorithm.

- `det_minor` takes A, a matrix.
- `det_minor(`A`)` returns the determinant $\det(A)$ of the matrix A.

Examples

```
> det_minor([[1,2],[3,4]])
```

-2

```
> det_minor(idn(3))
```

1

15.1.5 Rank of a matrix

The `rank` command finds the rank of a matrix.

- `rank` takes A, a matrix.
- `rank`(A) returns the rank of the matrix A.

Examples

```
> rank([[1,2],[3,4]])
```

$$2$$

```
> rank([[1,2],[2,4]])
```

$$1$$

15.1.6 Transconjugate of a matrix

The *transconjugate* of a matrix is the conjugate of the transpose of the matrix. The `trn` command finds the transconjugate of a matrix.

- `trn` takes A, a matrix.
- `trn`(A) returns the transconjugate A^* of A.

Example

```
> trn([[i,1+i],[1,1-i]])
```

$$\begin{bmatrix} -\mathrm{i} & 1 \\ 1-\mathrm{i} & 1+\mathrm{i} \end{bmatrix}$$

15.1.7 Equivalent matrix

The `changebase` command changes a matrix to represent the same linear function in a different basis.

- `changebase` takes two arguments:
  - A, a matrix.
  - P, a change-of-basis matrix.
- `changebase`(A, P) returns the matrix $P^{-1}AP$.

Examples

```
> changebase([[1,2],[3,4]],[[1,0],[0,1]])
```

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

```
> changebase([[1,1],[0,1]],[[1,2],[3,4]])
```

$$\begin{bmatrix} -5 & -8 \\ \frac{9}{2} & 7 \end{bmatrix}$$

Indeed:

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}^{-1} \cdot \begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} \cdot \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}-5 & -8\\ \frac{9}{2} & 7\end{bmatrix}.$$

15.1.8 Basis of a linear subspace

The `basis` command finds a basis of a linear subspace of $\mathbb{R}^n$ given a spanning set.

- `basis` takes L, a list of vectors generating a linear subspace of $\mathbb{R}^n$.
- `basis(`L`)` returns a list of vectors that is a basis of this linear subspace.

Example

```
> basis([[1,2,3],[1,1,1],[2,3,4]])
```

$$\{[-1, 0, 1], [0, -1, -2]\}$$

15.1.9 Basis of the intersection of two subspaces

The `ibasis` command finds a basis for the intersection of two subspaces of $\mathbb{R}^n$.

- `ibasis` takes two arguments: L_1 and L_2, two lists of vectors generating two subspaces of $\mathbb{R}^n$.
- `ibasis(`L_1, L_2`)` returns a list of vectors forming a basis for the intersection of these two subspaces.

Example

```
> ibasis([[1,2]],[[2,4]])
```

$$\{[1, 2]\}$$

15.1.10 Image of a linear function

The `image` command finds a basis for the image of a linear function.

- `image` takes A, a matrix representing a linear function with respect to the standard basis.
- `image(`A`)` returns a list of vectors that is a basis of the image of the linear function.

Example

```
> image([[1,1,2],[2,1,3],[3,1,4]])
```

$$\begin{bmatrix}-1 & 0 & 1\\ 0 & -1 & -2\end{bmatrix}$$

15.1.11 Kernel of a linear function

The `ker` or `kernel` or `nullspace` command finds a basis for the kernel of a linear function.

- `ker` takes A, a matrix representing a linear function with respect to the standard basis.
- `ker(`A`)` returns a list of vectors that is a basis of the kernel of the linear function.
- The `Nullspace` command is the inert form of `nullspace`.

Example

```
> ker([[1,1,2],[2,1,3],[3,1,4]])
```

$$\begin{bmatrix} 1 & 1 & -1 \end{bmatrix}$$

Remark. The `Nullspace` command is only useful in MAPLE mode. (See Section 2.5.2, p. 14; you can get into MAPLE mode by hitting the state line red button then Prog style, then choosing MAPLE and clicking Apply.)

- `Nullspace` takes A, an integer matrix representing a linear function with respect to the standard basis.
- `Nullspace(`A`) mod` p returns a list of vectors that is a basis for the kernel of the linear transformation $\mathbb{Z}/p\mathbb{Z}[X]$.

Examples

```
> Nullspace([[1,1,2],[2,1,3],[3,1,4]])
```

$$\text{nullspace}\left(\begin{bmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 3 & 1 & 4 \end{bmatrix}\right)$$

Input in MAPLE mode:

```
> Nullspace([[1,2],[3,1]]) mod 5
```

$$[2, -1]$$

In XCAS mode, the equivalent input is:

```
> nullspace([[1,2],[3,1]] % 5)
```

$$\begin{bmatrix} 2 \,\%\, 5 & -1 \end{bmatrix}$$

15.1.12 Subspace generated by the columns of a matrix

The `colspace` command finds a basis for the column space of a matrix.

- `colspace` takes one mandatory argument and one optional argument:
  - A, a matrix.
  - Optionally, *var*, a variable name.
- `colspace(`$A\,\langle, var\rangle$`)` returns a matrix whose columns are a basis of the subspace generated by the columns of A. With the optional argument *var*, XCAS will store the dimension of the subspace generated by the columns of A.

Examples

```
> colspace([[1,1,2],[2,1,3],[3,1,4]])
```

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & -2 \end{bmatrix}$$

```
> colspace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & -2 \end{bmatrix}$$

```
> dimension
```

$$2$$

15.1.13 Subspace generated by the rows of a matrix

The `rowspace` command finds a basis for the row space of a matrix.

- `rowspace` takes one mandatory argument and one optional argument:
  - A, a matrix.
  - Optionally, *var*, a variable name.
- `rowspace(`$A\,\langle, var\rangle$`)` returns a list of vectors which form a basis of the subspace generated by the rows of A. With the optional argument *var*, XCAS will store the dimension of the subspace generated by the rows of A.

Examples

```
> rowspace([[1,1,2],[2,1,3],[3,1,4]])
```

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

```
> rowspace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

```
> dimension
```

$$2$$

15.1.14 Testing positive definiteness of a symmetric matrix

The `isposdef` command checks whether the given symmetric matrix $A \in \mathbb{R}^{n\times n}$ is positive definite, i.e. whether $x^T Ax > 0$ for all vectors $x \in \mathbb{R}^n$.

`isposdef` takes a symmetric matrix A as its only argument and returns `true` if A is positive definite and `false` otherwise. Note that this procedure does not check whether A is symmetric.

Example

```
> isposdef([[1,-1,2],[-1,4,3],[2,3,-5]])
```

$$1$$

```
> isposdef([[1,-1,2],[-1,-2,3],[2,3,-5]])
```

$$0$$

15.2 Matrix reduction

15.2.1 Eigenvalues

The `eigenvals` command finds eigenvalues of a matrix.

- `eigenvals` takes A, a square matrix.
- `eigenvals(`A`)` returns the sequence of the eigenvalues of A, including multiplicities. (If A is an $n \times n$ matrix, then the sequence will have n values.)

Remark. Xcas may not be able to find the exact roots of the characteristic polynomial in some cases. In that case, `eigenvals(`A`)` will return approximate eigenvalues of A if the coefficients are numeric or a subset of the eigenvalues if the coefficients are symbolic.

Examples

```
> eigenvals([[4,1,-2],[1,2,-1],[2,1,0]])
```

$$2, 2, 2$$

```
> eigenvals([[4,1,0],[1,2,-1],[2,1,0]])
```

$$\frac{\text{rootof}\left([[1,0,-20,0,100],[1,0,-24,0,144,0,-148]]\right)}{18},$$
$$\frac{\text{rootof}\left([[-1,0,20,18,8],[1,0,-24,0,144,0,-148]]\right)}{36},$$
$$\frac{\text{rootof}\left([[-1,0,20,-18,8],[1,0,-24,0,144,0,-148]]\right)}{36}$$

```
> evalf(eigenvals([[4,1,0],[1,2,-1],[2,1,0]]))
```

$$1.46081112719, 4.21431974338, 0.324869129433$$

15.2.2 Jordan normal form

The `egvl` or `eigenvalues` or `eigVl` command finds the Jordan form of a matrix.

- `egvl` takes A, a square matrix.
- `egvl(`A`)` returns the Jordan normal form of A.

Examples

```
> egvl([[4,1,-2],[1,2,-1],[2,1,0]])
```

$$\begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

```
> egvl([[4,1,0],[1,2,-1],[2,1,0]])
```

$$\begin{bmatrix} \frac{\text{rootof}([[1,0,-20,0,100],[1,0,-24,0,144,0,-148]])}{18} & 0 & 0 \\ 0 & \frac{\text{rootof}([[-1,0,20,18,8],[1,0,-24,0,144,0,-148]])}{36} & 0 \\ 0 & 0 & \frac{\text{rootof}([[-1,0,20,-18,8],[1,0,-24,0,144,0,-148]])}{36} \end{bmatrix}$$

See Section 11.1.21, p. 221 for a discussion of `rootof`.

```
> evalf(egvl([[4,1,0],[1,2,-1],[2,1,0]]))
```

$$\begin{bmatrix} 1.46081112719 & 0.0 & 0.0 \\ 0.0 & 4.21431974338 & 0.0 \\ 0.0 & 0.0 & 0.324869129433 \end{bmatrix}$$

15.2.3 Eigenvectors

The `egv` or `eigenvectors` or `eigenvects` or `eigVc` command finds the eigenvectors of a diagonalizable matrix.

- `egv` takes A, a square matrix.
- `egv(`A`)` returns a matrix whose columns are the eigenvectors of the matrix A if A is diagonalizable, otherwise it will fail.

See also Section 15.2.5, p. 364 for characteristic vectors.

Examples

```
> egv([[1,1,3],[1,3,1],[3,1,1]])
```

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 2 & 0 \\ 1 & -1 & -1 \end{bmatrix}$$

```
> egv([[4,1,-2],[1,2,-1],[2,1,0]])
```

"Not diagonalizable at eigenvalue 2"

Input in complex mode:

```
> egv([[2,0,0],[0,2,-1],[2,1,2]])
```

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & -1 & -1 \\ 0 & -\mathrm{i} & \mathrm{i} \end{bmatrix}$$

15.2.4 Rational Jordan matrix

The `rat_jordan` command finds the rational Jordan form of a matrix.

- `rat_jordan` takes one mandatory and one optional argument:
  - A, a square matrix (preferably with exact coefficients).
  - Optionally, *var*, a variable name.
- `rat_jordan(`A`)` (in all modes but MAPLE) returns a sequence $[P, J]$ of two matrices, where J is the rational Jordan matrix of A (the most reduced matrix in the field of the coefficients of A or the complexified field in complex mode) and

$$J = P^{-1}AP$$

  The coefficients of P and J belongs to the same field as the coefficients of A. If A is diagonalizable in the field of its coefficients, then the columns of P are the eigenvectors of A.

  `rat_jordan(`A`)` (in MAPLE mode) only returns the matrix J.
- `rat_jordan(`$A\,\langle, var\rangle$`)` returns the matrix J, as above, and assigns the matrix P to the variable *var*.

Examples

Input not in MAPLE mode:

```
> rat_jordan([[1,0,0],[1,2,-1],[0,0,1]])
```

$$\begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 1\end{bmatrix}, \begin{bmatrix}2 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

```
> rat_jordan([[1,0,0],[1,2,-1],[0,0,1]],P)
```

$$\begin{bmatrix}2 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

```
> P
```

$$\begin{bmatrix}0 & 1 & 0\\ 1 & 0 & 1\\ 0 & 1 & 1\end{bmatrix}$$

```
> rat_jordan([[1,0,1],[0,2,-1],[1,-1,1]])
```

$$\begin{bmatrix}1 & 1 & 2\\ 0 & 0 & -1\\ 0 & 1 & 2\end{bmatrix}, \begin{bmatrix}0 & 0 & -1\\ 1 & 0 & -3\\ 0 & 1 & 4\end{bmatrix}$$

```
> rat_jordan([[1,0,0],[0,1,1],[1,1,-1]])
```

$$\begin{bmatrix}-1 & 0 & 0\\ 1 & 1 & 1\\ 0 & 0 & 1\end{bmatrix}, \begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 2\\ 0 & 1 & 0\end{bmatrix}$$

If A is symmetric and has eigenvalues with multiple orders, the matrix P returned by `rat_jordan(`A`)` will contain orthogonal eigenvectors (not always of norm equal to 1); that is, $P^T P$ will be a diagonal matrix where the diagonal is the square norm of the eigenvectors.

```
> rat_jordan([[4,1,1],[1,4,1],[1,1,4]])
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

15.2.5 Jordan form of a matrix

The `jordan` command finds the Jordan form of a matrix.

- `jordan` takes one mandatory and one optional argument:
  - A, a square matrix.
  - Optionally, *var*, a variable name.
- `jordan(`A`)` (in all modes but Maple) returns a sequence $[P, J]$ of two matrices, where the columns of P are the eigenvectors of A, J is the Jordan form of A, and

  $$J = P^{-1}AP$$

  `jordan(`A`)`, in Maple mode, only returns the matrix J.
- `jordan(`$A\,\langle, var\rangle$`)` returns the matrix J, as above, and assigns the matrix P to the variable *var*.

Examples

Input not in Maple mode:

```
> jordan([[4,1,1],[1,4,1],[1,1,4]])
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

```
> jordan([[4,1,1],[1,4,1],[1,1,4]],P)
```

$$\begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

```
> P
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}$$

If A is symmetric and has eigenvalues with multiple orders, the matrix P returned by `jordan(`A`)` will contain orthogonal eigenvectors (not always of norm equal to 1); that is, $P^T P$ will be a diagonal matrix where the diagonal is the square norm of the eigenvectors.

```
> jordan([[4,1,1],[1,4,1],[1,1,4]])
```

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

15.2.6 Powers of a square matrix

The `matpow` command finds the power of a square matrix, computed using the Jordan form.

- `matpow` command takes two arguments:
  - A, a square matrix.
  - n, an integer.
- `matpow`(A, n) returns A^n.

Example

```
> matpow([[1,2],[2,1]],n)
```

$$\begin{bmatrix} \frac{3^n+(-1)^n}{2} & \frac{3^n-(-1)^n}{2} \\ \frac{3^n-(-1)^n}{2} & \frac{3^n+(-1)^n}{2} \end{bmatrix}$$

Note:

```
> jordan([[1,2],[2,1]])
```

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$$

15.2.7 Characteristic polynomial

The *characteristic polynomial* of a square matrix A is the polynomial

$$P(x) = \det(xI - A).$$

The `charpoly` or `pcar` command finds the characteristic polynomial of a matrix.

- `charpoly` takes one mandatory argument and one or two optional argument(s):
  - A, a square matrix.
  - Optionally, x, a variable name.
  - Optionally, `fadeev`, the symbol.
- `charpoly`$(A\,\langle, x\rangle)$ returns the characteristic polynomial of A. It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument. If the last argument is `fadeev`, then `charpoly` will use the Faddeev–LeVerrier algorithm.

Examples

```
> charpoly([[4,1,-2],[1,2,-1],[2,1,0]])
```

$$[1, -6, 12, -8]$$

Hence, the characteristic polynomial of this matrix is $x^3 - 6x^2 + 12x - 8$.

```
> charpoly([[4,1,-2],[1,2,-1],[2,1,0]],X)
```

$$X^3 - 6X^2 + 12X - 8$$

15.2.8 Characteristic polynomial using Hessenberg algorithm

The `pcar_hessenberg` command finds the characteristic polynomial of a matrix. It computes the polynomial using the Hessenberg algorithm[1] which is more efficient ($O(n^3)$ deterministic) if the coefficients of the matrix are in a finite field or use a finite representation like approximate numeric coefficients. Note however that this algorithm behaves badly if the coefficients are, for example, in $\mathbb{Q}$.

- `pcar_hessenberg` takes one mandatory argument and one optional argument:
  - A, a square matrix.
  - Optionally, x, a variable name.
- `pcar_hessenberg(`$A\,\langle,x\rangle$`)` returns the characteristic polynomial of A. It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument.

Examples

```
> pcar_hessenberg([[4,1,-2],[1,2,-1],[2,1,0]] % 37)
```

$$[1 \,\%\, 37, (-6) \,\%\, 37, 12 \,\%\, 37, (-8) \,\%\, 37]$$

```
> pcar_hessenberg([[4,1,-2],[1,2,-1],[2,1,0]] % 37,x)
```

$$(1 \,\%\, 37)\, x^3 + ((-6) \,\%\, 37)\, x^2 + (12 \,\%\, 37)\, x + (-8) \,\%\, 37$$

Hence, the characteristic polynomial of $\begin{bmatrix} 4 & 1 & -2 \\ 1 & 2 & -1 \\ 2 & 1 & 0 \end{bmatrix}$ in $\mathbb{Z}/37\mathbb{Z}$ is $x^3 - 6x^2 + 12x - 8$.

15.2.9 Minimal polynomial

The minimal polynomial of a square matrix A is the polynomial P having minimal degree such that $P(A) = 0$. The `pmin` command finds the minimal polynomial of a matrix.

- `pmin` takes one mandatory argument and one optional argument:
  - A, a square matrix.
  - Optionally, x, a variable name.
- `pmin(`$A\,\langle,x\rangle$`)` returns the minimal polynomial A. It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument.

Examples

```
> pmin([[1,0],[0,1]])
```

$$[1, -1]$$

```
> pmin([[1,0],[0,1]],x)
```

$$x - 1$$

[1] See e.g. Henri Cohen, *A Course in Computational Algebraic Number Theory*

Hence the minimal polynomial of $\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$ is $x-1$.

```
> pmin([[2,1,0],[0,2,0],[0,0,2]])
```

$$[1,-4,4]$$

```
> pmin([[2,1,0],[0,2,0],[0,0,2]],x)
```

$$x^2-4x+4$$

Hence, the minimal polynomial of $\begin{bmatrix}2 & 1 & 0\\ 0 & 2 & 0\\ 0 & 0 & 2\end{bmatrix}$ is x^2-4x+4.

15.2.10 Adjoint matrix

The *comatrix* of a square matrix A of size n is the matrix B defined by $AB=\det(A)I$. The *adjoint* matrix $Q(x)$ of A is the comatrix of $xI-A$. It is a polynomial of degree $n-1$ in x having matrix coefficients and satisfies:

$$(xI-A)Q(x)=\det(xI-A)I=P(x)I,$$

where $P(x)$ is the characteristic polynomial of A. Since the polynomial $P(x)I-P(A)$ (with matrix coefficients) is also divisible by $xI-A$ (by algebraic identities), this means that $P(A)=0$. We also have $Q(x)=Ix^{n-1}+\cdots+B_0$ where B_0 is the comatrix of A (times -1 if n is odd).

The `adjoint_matrix` command finds the characteristic polynomial and adjoint of a given matrix.

- `adjoint_matrix` takes A, a square matrix.
- `adjoint_matrix(`A`)` returns the list of the coefficients of $P(x)$ (the characteristic polynomial of A), and the list of the matrix coefficients of $Q(x)$ (the adjoint matrix of A).

Examples

Let $A=\begin{bmatrix}4 & 1 & -2\\ 1 & 2 & -1\\ 2 & 1 & 0\end{bmatrix}$. Input:

```
> adjoint_matrix([[4,1,-2],[1,2,-1],[2,1,0]])
```

$$\left[[1,-6,12,-8],\left[\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix},\begin{bmatrix}-2 & 1 & -2\\ 1 & -4 & -1\\ 2 & 1 & -6\end{bmatrix},\begin{bmatrix}1 & -2 & 3\\ -2 & 4 & 2\\ -3 & -2 & 7\end{bmatrix}\right]\right]$$

Hence the characteristic polynomial is:

$$P(x)=x^3-6x^2+12x-8.$$

The determinant of A is equal to $-P(0)=8$. The comatrix of A is equal to:

$$B=Q(0)=\begin{bmatrix}1 & -2 & 3\\ -2 & 4 & 2\\ -3 & -2 & 7\end{bmatrix}$$

Hence the inverse of A is equal to:

$$\frac{1}{8}\begin{bmatrix}1 & -2 & 3\\ -2 & 4 & 2\\ -3 & -2 & 7\end{bmatrix}$$

The adjoint matrix of A is:

$$\begin{bmatrix} x^2-2x+1 & x-2 & -2x+3 \\ x-2 & x^2-4x+4 & -x+2 \\ 2x-3 & x-2 & x^2-6x+7 \end{bmatrix}.$$

Let $A = \begin{bmatrix} 4 & 1 \\ 1 & 2 \end{bmatrix}$. Input:

```
> adjoint_matrix([[4,1],[1,2]])
```

$$\left[[1,-6,7], \left[\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix}\right]\right]$$

Hence the characteristic polynomial P is:

$$P(x) = x^2 - 6x + 7.$$

The determinant of A is equal to $P(0) = 7$. The comatrix of A is equal to

$$Q(0) = -\begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix}$$

Hence the inverse of A is equal to:

$$-\frac{1}{7}\begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix}$$

The adjoint matrix of A is:

$$-\begin{bmatrix} x-2 & 1 \\ 1 & x-4 \end{bmatrix}$$

15.2.11 Companion matrix of a polynomial

The `companion` command finds a matrix given its characteristic polynomial; specifically, if the polynomial is $P(x) = x^n + a_{n-1}x^{n-1} + \cdots + a_{-1}x + a_0$, this matrix is equal to the identity matrix of size $n-1$ bordered with $[0, 0.., 0, -a_0]$ as first row, and with $[-a_0, -a_1, \ldots, -a_{n-1}]$ as last column.

- `companion` takes two arguments:
  - P, a unitary polynomial.
  - x, the name of its variable.
- `companion(`P, x`)` returns the matrix whose characteristic polynomial is P.

Examples

```
> companion(x^2+5x-7,x)
```

$$\begin{bmatrix} 0 & 7 \\ 1 & -5 \end{bmatrix}$$

```
> companion(x^4+3x^3+2x^2+4x-1,x)
```

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & -4 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \end{bmatrix}$$

15.2.12 Hessenberg matrix reduction

A *Hessenberg* matrix is a square matrix where the coefficients below the sub-principal diagonal are all zeros. The `hessenberg` command finds a Hessenberg matrix equivalent to a given square matrix.

- `hessenberg` takes one mandatory argument and one optional argument:
  - A, a matrix.
  - n, an integer, either 0, -1, -2 or a prime number greater than 1 (by default $n = 0$).
- `hessenberg(`$A\,\langle,n\rangle$`)` returns a list $[P, B]$ with $B = P^{-1}AP$ and:
  - if $n = 0$, B is a Hessenberg matrix.
  - if $n = -1$, the calculations are approximate and B is upper triangular.
  - if $n = -2$, the calculations are approximate, P is orthogonal and B has zero sub-subdiagonal elements.

`SCHUR(`A`)` is equivalent to `hessenberg(`A`,-1)`, which is compatible with HP calculators.

Example

Let

$$A = \begin{bmatrix} 3 & 2 & 2 & 2 & 2 \\ 2 & 1 & 2 & -1 & -1 \\ 2 & 2 & 1 & -1 & 1 \\ 2 & -1 & -1 & 3 & 1 \\ 2 & -1 & 1 & 1 & 2 \end{bmatrix}.$$

Input:

```
> A:=[[3,2,2,2,2],[2,1,2,-1,-1],[2,2,1,-1,1],[2,-1,-1,3,1],[2,-1,1,1,2]];
  P,B:=hessenberg(A)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & \frac{1}{4} & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 3 & 8 & 5 & \frac{5}{2} & 2 \\ 2 & 1 & \frac{1}{2} & -\frac{5}{4} & -1 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 0 & 2 & \frac{3}{2} & 2 \\ 0 & 0 & 0 & \frac{13}{8} & \frac{7}{2} \end{bmatrix}$$

Indeed, `pcar(A)` and `pcar(B)` both return $[1, -10, 13, 71, -50, -113]$ and it is easily verified that $B = P^{-1}AP$.

```
> B:=epsilon2zero(hessenberg(A,-1))
```

Output (to 2 digits):

$$\begin{bmatrix} 0.729361953258 & -0.420012536934 & -0.509269713814 & -0.170245642208 & -0.057283416961 \\ 0.24959754936 & 0.716215130873 & -0.0478044979307 & -0.376346706145 & -0.529919650904 \\ 0.354791329292 & -0.301345983892 & 0.742917332375 & 0.18980875771 & -0.441995682565 \\ 0.340274973326 & 0.168149195581 & 0.427145129505 & -0.462826562173 & 0.67770008281 \\ 0.405053403494 & 0.437654888177 & -0.0630869571782 & 0.761014971552 & 0.247520076117 \end{bmatrix},$$

$$\begin{bmatrix} 6.70109038302 & 0 & 0 & 0 & 0 \\ 0 & -1.86020364148 & 0 & 0 & 0 \\ 0 & 0 & -1.15956801687 & 0 & 0 \\ 0 & 0 & 0 & 1.68836042443 & 0 \\ 0 & 0 & 0 & 0 & 4.6303208509 \end{bmatrix}$$

15.2.13 Hermite normal form

The *Hermite normal form* of a matrix A with integer coefficients is a sort of integer row-echelon form. It is an upper triangular matrix B such that $B = UA$ for a matrix U which is invertible in $\mathbb{Z}$ ($\det(U) = \pm 1$). The `ihermite` command finds the Hermite normal form of a matrix.

- `ihermite` takes A, a matrix with coefficients in $\mathbb{Z}$.
- `ihermite(`A`)` return a list $[U, B]$ as above, and the absolute value of the elements above the diagonal of B are less than the pivot of the column divided by 2.

The result is obtained by a Gauss-like reduction algorithm using only operations of rows with integer coefficients and invertible in $\mathbb{Z}$.

Example

```
> A:=[[9,-36,30],[-36,192,-180],[30,-180,180]]:;
  ihermite(A)
```

$$\begin{bmatrix} 13 & 9 & 7 \\ 6 & 4 & 3 \\ 20 & 15 & 12 \end{bmatrix}, \begin{bmatrix} 3 & 0 & 30 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{bmatrix}$$

Application: Compute a $\mathbb{Z}$-basis of the kernel of a matrix having integer coefficients. Let M be a matrix with integer coefficients. To find the nullspace of M, you want find what you can multiply M by on the right to get the zero vector, but the Hermite command returns a matrix that you multiply on the left of M. So consider the transpose of M, M^T.

Let A be the Hermite normal form of M^T, and U an invertible matrix in $\mathbb{Z}$ such that $A = UM^T$. Transposing this, you will get $A^T = MU^T$. Note that M times a column of U^T equals the corresponding column of A^T. So if a column of A^T is the zero vector, then M times the corresponding column of U^T will be the zero vector. In fact, these columns of U^T will be a $\mathbb{Z}$-basis for the nullspace of M.

Any columns of A^T which are all zeros correspond to the rows of A which are all zeros. Since A is in Hermite form, these will be at the bottom, and so the corresponding rows of U will be at the bottom, and will be a $\mathbb{Z}$-basis for the nullspace of M.

As an example, consider the matrix M:

```
> M:=[[1,4,7],[2,5,8],[3,6,9]]
```

Find the Hermite decomposition:

```
> (U,A):=ihermite(transpose(M))
```

$$\begin{bmatrix} -3 & 1 & 0 \\ 4 & -1 & 0 \\ -1 & 2 & -1 \end{bmatrix}, \begin{bmatrix} 1 & -1 & -3 \\ 0 & 3 & 6 \\ 0 & 0 & 0 \end{bmatrix}$$

Only the third row of A consists of all zeros, so a $\mathbb{Z}$-basis for the nullspace of M consists of only the third row of U, namely $[-1, 2, -1]$.

You can check that this is in the nullspace:

```
> M*U[2]
```

$$[0, 0, 0]$$

15.2.14 Smith normal form in $\mathbb{Z}$

A matrix B is in *Smith normal form* if the only non-zero entries are on the diagonal (for non-square matrices, this simply means that $b_{ij} = 0$ for $i \neq j$) and $b_{i,i}$ divides $b_{i+1,i+1}$. The elements $b_{i,i}$ are called invariant factors and are used to describe the structure of finite abelian groups.

For any matrix A with coefficients in $\mathbb{Z}$, there exist matrices U and V, invertible in $\mathbb{Z}$, such that $B = UAV$ is in Smith normal form and has coefficients in $\mathbb{Z}$. The `ismith` command finds the matrices U, B and V.

- `ismith` takes A, a matrix with coefficients in $\mathbb{Z}$.
- `ismith(`A`)` returns a list $[U, B, V]$ of three matrices such that $B = UAV$ is in Smith normal form and U and V are invertible in $\mathbb{Z}$.

Example

```
> A:=[[9,-36,30],[-36,192,-180],[30,-180,180]]:;
  U,B,V:=ismith(A)
```

$$\begin{bmatrix} -3 & 0 & 1 \\ 6 & 4 & 3 \\ 20 & 15 & 12 \end{bmatrix}, \begin{bmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{bmatrix}, \begin{bmatrix} 1 & 24 & -30 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The invariant factors are 3, 12 and 60.

15.2.15 Smith normal form

The `smith` command finds the Smith normal form of a matrix with elements in a field K.

- `smith` takes A, a square matrix with elements in a field K.
- `smith(`A`)` returns a list $[U, V, D]$ of three matrices where U and V are invertible, D is diagonal, and $D = UAV$.Input:

Examples

```
> M:=([[5,-2,3,6],[1,-3,1,3],[7,-6,-4,7],[-2,-4,-3,0]]) % 17:;
  A:=x*idn(4)-M
```

$$\begin{bmatrix} x + (-5) \,\%\, 17 & 2 \,\%\, 17 & (-3) \,\%\, 17 & (-6) \,\%\, 17 \\ (-1) \,\%\, 17 & x + 3 \,\%\, 17 & (-1) \,\%\, 17 & (-3) \,\%\, 17 \\ (-7) \,\%\, 17 & 6 \,\%\, 17 & x + 4 \,\%\, 17 & (-7) \,\%\, 17 \\ 2 \,\%\, 17 & 4 \,\%\, 17 & 3 \,\%\, 17 & x \end{bmatrix}$$

```
> U,D,V:=smith(A):;
```

$$\begin{bmatrix} 0 \,\%\, 17 & (-1) \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 \\ 0 \,\%\, 17 & 0 \,\%\, 17 & 6 \,\%\, 17 & 4 \,\%\, 17 \\ (-2x+5) \,\%\, 17 & (-4x-5) \,\%\, 17 & (-3x-6) \,\%\, 17 & (x^2-3x+6) \,\%\, 17 \\ (2x^2+5x+6) \,\%\, 17 & (4x^2+8x+2) \,\%\, 17 & (3x^2+4x+1) \,\%\, 17 & (-x^3-2x^2+2x-6) \,\%\, 17 \end{bmatrix},$$

$$\begin{bmatrix} 1 \,\%\, 17 & (x+3) \,\%\, 17 & (-6x^2-3x-7) \,\%\, 17 & (6x^5+2x^4-2x^3+x^2-8x+6) \,\%\, 17 \\ 0 \,\%\, 17 & 1 \,\%\, 17 & (-6x-2) \,\%\, 17 & (6x^4+x^3-6x^2+5x-6) \,\%\, 17 \\ 0 \,\%\, 17 & 0 \,\%\, 17 & 1 \,\%\, 17 & (-x^3+3x^2+7) \,\%\, 17 \\ 0 \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 & 1 \,\%\, 17 \end{bmatrix},$$

$$\begin{bmatrix} 1 \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 \\ 0 \,\%\, 17 & 1 \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 \\ 0 \,\%\, 17 & 0 \,\%\, 17 & 1 \,\%\, 17 & 0 \,\%\, 17 \\ 0 \,\%\, 17 & 0 \,\%\, 17 & 0 \,\%\, 17 & (-x^4-2x^3+8x^2-3x+2) \,\%\, 17 \end{bmatrix}$$

You can check this:

```
> normal(U*A*V-D)
```

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

```
> B:=[[x^2+x-1,1,0,1],[-1,x,0,-1],[0,x^2+1,x,0],[1,0,1,x^2+x+1]] % 3:;
  L:=smith(B)
```

$$\begin{bmatrix} 0 \,\%\, 3 & (-1) \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 \\ 1 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 & (-x^2-x+1) \,\%\, 3 \\ 0 \,\%\, 3 & (x^2+1) \,\%\, 3 & (-x) \,\%\, 3 & (x^2+1) \,\%\, 3 \\ (-1) \,\%\, 3 & (-x^4-x^3+x+1) \,\%\, 3 & (x^3+x^2-x+1) \,\%\, 3 & (-x^4-x^3+x^2-x) \,\%\, 3 \end{bmatrix},$$

$$\begin{bmatrix} 1 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 \\ 0 \,\%\, 3 & 1 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 \\ 0 \,\%\, 3 & 0 \,\%\, 3 & 1 \,\%\, 3 & 0 \,\%\, 3 \\ 0 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 & (-x^6+x^5+x+1) \,\%\, 3 \end{bmatrix},$$

$$\begin{bmatrix} 1 \,\%\, 3 & x \,\%\, 3 & (x^3+x^2-x) \,\%\, 3 & (-x^7+x^6+x^4+x^3+x^2+x-1) \,\%\, 3 \\ 0 \,\%\, 3 & 1 \,\%\, 3 & (x^2+x-1) \,\%\, 3 & (-x^6+x^5+x^3+x^2+x+1) \,\%\, 3 \\ 0 \,\%\, 3 & 0 \,\%\, 3 & 1 \,\%\, 3 & (-x^4-x^3-x^2-x) \,\%\, 3 \\ 0 \,\%\, 3 & 0 \,\%\, 3 & 0 \,\%\, 3 & 1 \,\%\, 3 \end{bmatrix}$$

15.3 Matrix factorizations

Note that most matrix factorization algorithms are implemented numerically, only a few of them will work symbolically.

15.3.1 Cholesky decomposition

If M is a square symmetric positive definite matrix, the Cholesky decomposition is $M = P^T P$, where P is a lower triangular matrix. The `cholesky` command finds the matrix P.

- `cholesky` takes M, a square symmetric positive definite matrix.
- `cholesky(`M`)` returns a symbolic or numeric matrix P given by the Cholesky decomposition.

Examples

```
> cholesky([[1,1],[1,5]])
```

$$\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}$$

```
> cholesky([[3,1],[1,4]])
```

$$\begin{bmatrix} \sqrt{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{33}}{3} \end{bmatrix}$$

```
> cholesky([[1,1],[1,4]])
```

$$\begin{bmatrix} 1 & 0 \\ 1 & \sqrt{3} \end{bmatrix}$$

Remark. If the matrix argument A is not a symmetric matrix, `cholesky(A)` does not return an error, bu instead uses the symmetric matrix B of the the quadratic form q corresponding to the (non symmetric) bilinear form of the matrix A.

Example

```
> cholesky([[1,-1],[-1,4]])
```

or:

```
> cholesky([[1,-3],[1,4]])
```

$$\begin{bmatrix} 1 & 0 \\ -1 & \sqrt{3} \end{bmatrix}$$

15.3.2 QR decomposition

The QR decomposition of a square matrix A is $A = QR$, where Q is an orthogonal matrix ($Q^TQ = I$) and R is upper triangular. The `qr` command finds the QR decomposition of a matrix.

- `qr` takes A, a numeric square matrix.
- `qr(A)` returns a list $[Q, R]$ with Q and R from the QR decomposition.

Examples

```
> A:=[[3,5],[4,5]]:;
  qr(A)
```

$$\begin{bmatrix} \frac{3}{5} & \frac{4}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{bmatrix}, \begin{bmatrix} 5 & 7 \\ 0 & 1 \end{bmatrix}$$

```
> qr([[1,2],[3,4]])
```

$$\begin{bmatrix} \frac{1}{\sqrt{10}} & \frac{3}{5\frac{\sqrt{10}}{5}} \\ \frac{3}{\sqrt{10}} & -\frac{1}{5\frac{\sqrt{10}}{5}} \end{bmatrix}, \begin{bmatrix} \sqrt{10} & \frac{7}{5}\sqrt{10} \\ 0 & \frac{\sqrt{10}}{5} \end{bmatrix}$$

15.3.3 QR decomposition (for TI compatibility)

The `QR` command finds the QR decomposition of a matrix.

- `QR` takes three arguments:
  - A, a square matrix.
  - q and r, two variable names.
- `QR`(A, q, r) returns the matrix R from the QR decomposition of A, and assigns the matrices Q and R to the variables q and r.

Example

```
> QR([[3,5],[4,5]],Q,R)
```

$$\begin{bmatrix} 5 & 7 \\ 0 & 1 \end{bmatrix}$$

The result is the matrix `R`. To obtain Q:

```
> Q
```

$$\begin{bmatrix} \frac{3}{5} & \frac{4}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{bmatrix}$$

15.3.4 LQ decomposition (HP compatible)

The LQ decomposition of a matrix A is $A = LQP$, where L is lower triangular the same size as A (if A is not square, then $\ell_{i,j} = 0$ for $i > j$), Q is an orthogonal matrix, and P is a permutation matrix.

The `LQ` command finds the LQ decomposition of a matrix.

- `LQ` takes A, a matrix.
- `LQ`(A) returns a list $[L, Q, P]$ of the matrices given by the LQ decomposition.

Examples

```
> L,Q,P:=LQ([[4,0,0],[8,-4,3]])
```

$$\left[\begin{bmatrix} 4.0 & 0.0 & 0.0 \\ 8.0 & 5.0 & -4.4408920985 \times 10^{-16} \end{bmatrix}, \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & -0.8 & 0.6 \\ 0.0 & -0.6 & -0.8 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\right]$$

```
> L,Q,P:=LQ([[24,18],[30,24]])
```

$$\left[\begin{bmatrix} -30.0 & 0.0 \\ -38.4 & -1.2 \end{bmatrix}, \begin{bmatrix} -0.8 & -0.6 \\ 0.6 & -0.8 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}\right]$$

In the above examples, $LQ = PA$.

15.3.5 LU decomposition

The LU decomposition of a square matrix A is $PA = LU$, where P is a permutation matrix, L is lower triangular with ones on the diagonal, and U is upper triangular. The `lu` command finds the LU decomposition of a matrix.

- `lu` takes A, a square matrix.
- `lu(`A`)` returns a list $[p, L, U]$ where p is a permutation that determines P, and P, L and U are the LU decomposition of A.

The permutation matrix P is defined from p by:

$$P_{i,p(i)} = 1, \quad P_{i,j} = 0 \text{ if } j \neq p(i).$$

In other words, it is the identity matrix where the rows are permuted according to the permutation p. You can get the permutation matrix from p by P`:=permu2mat(`p`)` (see Section 12.2.6, p. 272).

Example

```
> A:=[[3.,5.],[4.,5.]]:;
  (p,L,U):=lu(A)
```

$$[1, 0], \begin{bmatrix} 1 & 0 \\ 0.75 & 1 \end{bmatrix}, \begin{bmatrix} 4.0 & 5.0 \\ 0 & 1.25 \end{bmatrix}$$

Verification:

```
> permu2mat(p)*A; L*U
```

$$\begin{bmatrix} 4.0 & 5.0 \\ 3.0 & 5.0 \end{bmatrix}, \begin{bmatrix} 4.0 & 5.0 \\ 3.0 & 5.0 \end{bmatrix}$$

Note that the permutation is different for exact input (the choice of pivot is the simplest instead of the largest in absolute value).

```
> lu([[1,2],[3,4]])
```

$$[0, 1], \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix}$$

```
> lu([[1.0,2],[3,4]])
```

$$[1, 0], \begin{bmatrix} 1 & 0 \\ 0.333333333333 & 1 \end{bmatrix}, \begin{bmatrix} 3.0 & 4.0 \\ 0 & 0.666666666667 \end{bmatrix}$$

15.3.6 LU decomposition (for TI compatibility)

The `LU` command finds the LU decomposition of a matrix.

- `LU` takes four arguments:
  - A, a numeric square matrix.
  - l, u and p, three variable names.
- `LU(`A`)` returns the matrix P from the LU decomposition of A, and assigns L, U and P to the variables l, u and p. Namely, P is a permutation matrix, L is lower triangular with ones on the diagonal, and U is upper triangular with $PA = LU$.

Example

```
> LU([[3,5],[4,5]],L,U,P)
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
> L,U
```

$$\begin{bmatrix} 1 & 0 \\ \frac{4}{3} & 1 \end{bmatrix}, \begin{bmatrix} 3 & 5 \\ 0 & -\frac{5}{3} \end{bmatrix}$$

15.3.7 Singular values (HP compatible)

The singular values of a matrix A are the positive square roots of the eigenvalues of AA^T. So, if A is symmetric, the singular values are the absolute values of the eigenvalues of A. The `SVL` or `svl` command finds the singular values of a matrix.

- `SVL` takes A, a matrix.
- `SVL(`A`)` returns a list of the singular values of A.

Examples

```
> SVL([[1,2],[3,4]])
```

$$[0.365966190626, 5.46498570422]$$

```
> evalf(sqrt(eigenvals([[1,2],[3,4]]*transpose([[1,2],[3,4]]))))
```

$$5.46498570422, 0.365966190626$$

```
> svl([[1,4],[4,1]])
```

$$[5.0, 3.0]$$

```
> abs(eigenvals([[1,4],[4,1]]))
```

$$5, 3$$

15.3.8 Singular value decomposition

The *singular value decomposition* of a matrix A is a factorization $A = USQ^T$, where U and Q are orthogonal and S is a diagonal matrix. The `svd` command finds the singular value decomposition of a matrix.

- `svd` takes A, a numeric square matrix.
- `svd(`A`)` returns a list $[U, s, Q]$ where U and Q are the orthogonal matrices of the singular value decomposition and s is the diagonal of the matrix S.

You can get the diagonal matrix S from s with S`=diag(`s`)` (see Section 14.2.2, p. 326).

Examples

```
> svd([[1,2],[3,4]])
```

$$\begin{bmatrix} -0.404553584834 & -0.914514295677 \\ -0.914514295677 & 0.404553584834 \end{bmatrix}, [5.46498570422, 0.365966190626],$$
$$\begin{bmatrix} -0.576048436766 & 0.81741556047 \\ -0.81741556047 & -0.576048436766 \end{bmatrix}$$

```
> (U,s,Q):=svd([[3,5],[4,5]])
```

$$\begin{bmatrix} -0.672988041811 & -0.739653361771 \\ -0.739653361771 & 0.672988041811 \end{bmatrix}, [8.6409011028, 0.578643354497],$$
$$\begin{bmatrix} -0.576048436766 & 0.81741556047 \\ -0.81741556047 & -0.576048436766 \end{bmatrix}$$

Verification:

```
> U*diag(s)*tran(Q)
```

$$\begin{bmatrix} 3.0 & 5.0 \\ 4.0 & 5.0 \end{bmatrix}$$

15.3.9 LDL decomposition

The `ldl` command computes the LDL decomposition of the given symmetric (real or complex) or Hermitian matrix A.

- `ldl` takes one mandatory argument and a sequence of optional arguments:
  - A, a symmetric or Hermitian matrix.
  - Optionally, *opts*, a sequence of options, each of which is one of the following:
    - `hermite`, which must be passed alongside a Hermitian matrix A since no check is performed.
    - `zip`, which forces merging the output matrices L and D into matrix M.
- `ldl(`$A\,\langle, opts\rangle$`)` returns the sequence p, L, D or p, M (if the option `zip` is passed), where p is the permutation defining the permutation matrix P (using `permu2mat` command), L is a unit lower triangular matrix and D is a tridiagonal matrix with 1- or 2-blocks on the diagonal. Matrix M is obtained by merging L and D together which saves the space while allowing unambiguous reconstruction of L and D. The following relation holds:

$$PAP^T = LDL^T.$$

Example

```
> A:=[[1,-1,2],[-1,4,3],[2,3,-5]]
```

$$\begin{bmatrix} 1 & -1 & 2 \\ -1 & 4 & 3 \\ 2 & 3 & -5 \end{bmatrix}$$

```
> ldl(A)
```

$$[0,2,1]\,,\begin{bmatrix}1&0&0\\2&1&0\\-1&-\frac{5}{9}&1\end{bmatrix},\begin{bmatrix}1&0&0\\0&-9&0\\0&0&\frac{52}{9}\end{bmatrix}$$

15.3.10 Computing inertia of a symmetric matrix

The `inertia` command computes the inertia of a real symmetric matrix, i.e. the number of positive, negative and zero eigenvalues, by performing LDL decomposition (see Section 15.3.9, p. 377). It can also use the factorization for solving a system of linear equations if certain inertia-related conditions are satisfied.

- `inertia` takes one mandatory argument and one or two optional arguments:
  - A, a symmetric matrix of order n with real coefficients.
  - Optionally, B, a matrix of type $m \times n$ or a vector of length n.
  - Optionally, p_0, an integer such that $0 \le p_0 \le n$.

  Note that `inertia` does not check whether A is symmetric.
- `inertia(`A`)` returns the list $[p, n, z]$ where p, n and z are the numbers of positive, negative and zero eigenvalues of A, respectively.
- `inertia(`$A, B\,\langle, p_0\,\rangle$`)` returns the sequence containing two elements: the list $[p, n, z]$ and the solution X to the system $AX^T = B^T$ in case $z = 0$. If p_0 is given, then X is not computed if $p \ne p_0$ and an empty list is returned instead.
- The inertia of A is computed from the LDL factorization of A. The matrix X can be computed at a small cost once L and D are known. The purpose of the argument p_0 is to prevent unnecessary computation if the matrix A has to be adjusted for the desired value of p before solving the system.

Examples

```
> A:=[[1,-1,2],[-1,4,3],[2,3,-5]]
```

$$\begin{bmatrix}1&-1&2\\-1&4&3\\2&3&-5\end{bmatrix}$$

```
> inertia(A)
```

$$[2,1,0]$$

```
> inertia(A,[[1,2,3],[4,5,6]])
```

$$[2,1,0]\,,\begin{bmatrix}\frac{15}{13}&\frac{8}{13}&\frac{3}{13}\\\frac{177}{52}&\frac{71}{52}&\frac{51}{52}\end{bmatrix}$$

If we set $p_0 = 1$, then X is not computed.

```
> inertia(A,[[1,2,3],[4,5,6]],1)
```

$$[2,1,0]\,,[]$$

15.3.11 Short basis of a lattice

The `lll` command finds a short basis for the $\mathbb{Z}$-modules generated by the rows of a matrix.

- `lll` takes M, an invertible matrix with integer coefficients.
- `lll(`M`)` returns the sequence (S, A, L, O) where:
  - the rows of S is a short basis of the $\mathbb{Z}$-module generated by the rows of M,
  - A is the change-of-basis matrix from the short basis to the basis defined by the rows of M ($AM = S$),
  - L is a lower triangular matrix, the modulus of its non diagonal coefficients are less than $1/2$,
  - O is a matrix with orthogonal rows such that $LO = S$.

Examples

```
> (S,A,L,O):=lll(M:=[[2,1],[1,2]])
```

$$\begin{bmatrix} -1 & 1 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ \frac{3}{2} & \frac{3}{2} \end{bmatrix}$$

So the original basis is $v_1 = [2, 1], v_2 = [1, 2]$ and the short basis is $w_1 = [-1, 1], w_2 = [2, 1]$. Since $w_1 = -v_1 + v_2$ and $w_2 = v_1$ then $AM = S$ and $LO = S$.

As another example, input:

```
> M:=[[3,2,1],[1,2,3],[2,3,1]]:;
  S,A,L,O:=lll(M)
```

$$\begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Properties: $AM = S$ and $LO = S$.

15.4 Matrix norms

See Section 14.1.1, p. 322 for different norms on vectors.

15.4.1 Frobenius norm

The *Frobenius norm* of a matrix A is $\sqrt{\sum_{i,j} a_{i,j}^2}$. The `frobenius_norm` command finds the Frobenius norm of a matrix.

- `frobenius_norm` takes A, a matrix.
- `frobenius_norm(`A`)` returns the Frobenius norm of A.

Example

```
> frobenius_norm([[1,2,3],[3,-9,6],[4,5,6]])
```

$$\sqrt{217}$$

Indeed, $\sqrt{1^2 + 2^2 + 3^2 + 3^2 + (-9)^2 + 6^2 + 4^2 + 5^2 + 6^2} = \sqrt{217}$.

15.4.2 ℓ^2 matrix norm

The ℓ^2 norm of a matrix is an operator norm (see Section 15.4.6, p. 381) induced by the ℓ^2 norm on vectors (see Section 14.1.1, p. 322). The `l2norm` or `norm` command computes the ℓ^2 norm of a matrix.

- `l2norm` takes A, a matrix.
- `l2norm(`A`)` returns the ℓ^2 norm of A.

Example

```
> l2norm([[1,2],[3,-4]])
```

5.11667273602

15.4.3 ℓ^∞ matrix norm

The ℓ^∞ norm of a matrix A is $\max_{j,k}(|a_{j,k}|)$. The `maxnorm` command finds the ℓ^∞ norm of a matrix. (See also Section 14.1.1, p. 322.)

- `maxnorm` takes A, a matrix.
- `maxnorm(`A returns the ℓ^∞ norm of A.

Example

```
> maxnorm([[1,2],[3,-4]])
```

4

15.4.4 Matrix row norm

The row norm of a matrix A is $\max_k(\sum_j |a_{j,k}|)$. (This is also an operator norm; see Section 15.4.6, p. 381.) The `rownorm` or `rowNorm` command finds the row norm of a matrix. For matrices, `linfnorm` is also a synonym.

- `rownorm` takes A, a matrix
- `rownorm(`A`)` returns the row norm of A.

Example

```
> rownorm([[1,2],[3,-4]])
```

7

Indeed, $\max\{1+2, 3+4\} = 7$.

15.4.5 Matrix column norm

The column norm of a matrix A is $\max_j(\sum_k |a_{j,k}|)$. (This is also an operator norm; see Section 15.4.6, p. 381.) The `colnorm` or `colNorm` command finds the column norm of a matrix. For matrices, `l1norm` is also a synonym.

- `colnorm` takes A, a matrix
- `colnorm(`A`)` returns the column norm of A.

Example

```
> colnorm([[1,2],[3,-4]])
```

6

Indeed, $\max\{1+3, 2+4\} = 6$.

15.4.6 Operator norm of a matrix

Operator norms. In mathematics, particularly functional analysis, a linear function between two normed spaces $f : E \to F$ is continuous exactly when there is a number K such that $\|f(x)\|_F \leq K\|x\|$ for all x in E. (See Section 14.1.1, p. 322 for norms on $\mathbb{R}^n$.) For this reason, they are also called bounded linear functions. The infimum of all such K is defined to be the operator norm of f, and it depends on the norms of E and F. There are other characterizations of the operator norm of f, such as the supremum of $\|f(x)\|_F$ over all x in E with $\|x\|_E \leq 1$.

If E and F are finite dimensional, then any linear function $f : E \to F$ will be bounded.

Any $m \times n$ matrix $A = (a_{jk})$ corresponds to a linear function $f : \mathbb{R}^n \to \mathbb{R}^m$ defined by $f(x) = Ax$. The operator norm of A will be the operator norm of f.

- If $\mathbb{R}^n$ and $\mathbb{R}^m$ both have the ℓ_1 norm, namely for $x = (x_1, x_2, \dots)$ the norm is $\|x\| = \sum_j |x_j|$, the operator norm of A is
$$\max_k \sum_j |a_{jk}|.$$
This is the column norm given by `colnorm(`A`)` (see Section 15.4.5, p. 380).
- If $\mathbb{R}^n$ and $\mathbb{R}^m$ both have the ℓ_2 norm, namely for $x = (x_1, x_2, \dots)$ the norm is $\|x\| = \sqrt{\sum_j x_j^2}$ (the usual Euclidean norm), the operator norm of A is the largest eigenvalue of $f^* \circ f$, where f^* is the transpose of f, and so the largest singular value of f. This is given by `l2norm` (see Section 15.4.2, p. 380) or `max(SVL(`A`))` (see Section 15.3.7, p. 376).
- If $\mathbb{R}^n$ and $\mathbb{R}^m$ both have the ℓ_∞ norm, namely for $x = (x_1, x_2, \dots)$ the norm is $|x| = \max_j |x_j|$, the operator norm of A is
$$\max_j \sum_k |a_{jk}|.$$
This is given by `rownorm(`A`)` (see Section 15.4.4, p. 380).

Computing operator norms. The `matrix_norm` command is a command which finds any of the above operator norms.

- `matrix_norm` takes two arguments:
  - A, a matrix.
  - *arg*, which can be `1`, `2` or `inf`.
- `matrix_norm(`A`,`*arg*`)` returns an operator norm of the operator associated to the matrix, the norm is determined by *arg*.
  - If *arg* is `1`, it is based on the ℓ^1 norm on $\mathbb{R}^n$.

    `matrix_norm(`A`,1)` is the same as `colnorm(`A`)` and `l1norm(`A`)`.
  - If *arg* is `2`, it is based on the ℓ^2 norm on $\mathbb{R}^n$.

    `matrix_norm(`A`,2)` is the same as `l2norm(`A`)` and `norm(`A`)`.
  - If *arg* is `inf`, it is based on the ℓ^∞ norm on $\mathbb{R}^n$.

    `matrix_norm(`A`,inf)` is the same as `rownorm(`A`)` and `linfnorm(`A`)`.

Examples

```
> B:=[[1,2,3],[3,-9,6],[4,5,6]]
```

then:

```
> matrix_norm(B,1)
```

or:

```
> l1norm(B)
```

or:

```
> colnorm(B)
```

16

since $\max\{1+3+4, 2+9+5, 3+6+6\} = 16$.

```
> matrix_norm(B,2)
```

or:

```
> l2norm(B)
```

11.2449175989

```
> matrix_norm(B,inf)
```

or:

```
> linfnorm(B)
```

or:

```
> rowNorm(B)
```

18

Indeed, $\max\{1+2+3, 3+9+6, 4+5+6\} = 18$.

15.5 Isometries

An isometry of $\mathbb{R}^n$ is a distance-preserving map. In $\mathbb{R}^2$, the isometries are made up of:

- reflection across a line.
- rotation about a point.
- translation.

In $\mathbb{R}^3$, the isometries are made up of:

- reflection across a plane.
- rotation about a line.
- translation.

An isometry is direct if it preserves orientation (it does not involve a reflection), otherwise it is indirect.

An $n \times n$ matrix A determines an isometry of the function $f : \mathbb{R}^n \to \mathbb{R}^n$ by $f(\mathbf{x}) = A\mathbf{x}$) is an isometry. Such isometries fix the origin, so they cannot involve translation, can only rotate about the origin, and the line or plane of reflection will pass through the origin.

An isometry in $\mathbb{R}^2$ can be characterized by:

- The angle of rotation (for a direct isometry) or a normal to the line of reflection (for an indirect isometry).
- `1` or `-1` to indicate whether it is direct or indirect; `1` for direct and `-1` for indirect.

An isometry in $\mathbb{R}^3$ can be characterized by:

- The direction of an axis of rotation.
- The angle of rotation (for a direct isometry) or a normal to the plane of reflection (for an indirect isometry).
- `1` or `-1` to indicate whether it is direct or indirect; `1` for direct and `-1` for indirect.

15.5.1 Recognizing an isometry

The `isom` command determines whether or not a 2×2 or 3×3 matrix determines an isometry, and if it does, finds a characterization.

- `isom` takes A, a 2×2 or 3×3 matrix.
- `isom(`A`)` returns `[0]` is A does not determine an isometry, otherwise it returns a list $[char, n]$, where *char* is the characteristic element of a list of characteristic elements and n is `1` for a direct isometry and `-1` for an indirect isometry.
  - For a 2×2 matrix, *char* is the angle of rotation about the origin for a direct isometry or a vector determining the line (through the origin) of reflection for an indirect symmetry.
  - For a 3×3 matrix, *char* is a list consisting of the axis direction and angle of rotation for a direct isometry or a vector normal to the plane of reflection for an indirect isometry.

Examples

```
> isom([[0,0,1],[0,1,0],[1,0,0]])
```

$$[[1, 0, -1], -1]$$

which means that this isometry is a 3D symmetry with respect to the plane $x - z = 0$.

```
> isom(sqrt(2)/2*[[1,-1],[1,1]])
```

$$\left[\frac{\pi}{4}, 1\right]$$

Hence, this isometry is a 2D rotation of angle $\frac{\pi}{4}$.

```
> isom([[0,0,1],[0,1,0],[0,0,1]])
```

$$[0]$$

therefore this transformation is not an isometry.

15.5.2 Finding the matrix of an isometry

The `mkisom` command finds the matrix of an isometry given the characteristic elements.

- `mkisom` takes two arguments:

– *char*, the characteristic element (for isometries of $\mathbb{R}^2$ or a list of characteristic elements (for isometries of $\mathbb{R}^3$.

For isometries of $\mathbb{R}^2$, *char* will be the angle of rotation for direct isometries or a vector determining the line (through the origin) of reflection for an indirect symmetry.

For isometries of $\mathbb{R}^3$, *char* will be the list consisting of the axis direction and angle of rotation for a direct isometry or a vector normal to the plane of reflection for an indirect isometry.

– n, either `+1` for a direct isometry or `-1` an indirect isometry.

- `mkisom(`*char*$,n$`)` returns a matrix of the corresponding isometry.

Examples

To obtain the matrix of the rotation about axis $[-1,2,-1]$ of angle π, input:

```
> mkisom([[-1,2,-1],pi],1)
```

$$\begin{bmatrix} -\frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \end{bmatrix}$$

To obtain the matrix of the symmetry with respect to O, input:

```
> mkisom([pi],-1)
```

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

To obtain the matrix of the symmetry with respect to the plane $x+y+z=0$, input:

```
> mkisom([1,1,1],-1)
```

$$\begin{bmatrix} \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ -\frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

To obtain the matrix of the product of a rotation of axis $[1,1,1]$ and angle $\frac{\pi}{3}$ and of a symmetry with respect to the plane $x+y+z=0$, input:

```
> mkisom([[1,1,1],pi/3],-1)
```

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{bmatrix}$$

To obtain the matrix of the plane rotation of angle $\frac{\pi}{2}$, input:

```
> mkisom(pi/2,1)
```

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

To obtain matrix of the plane symmetry with respect to the line of equation $x+2y=0$, input:

```
> mkisom([1,2],-1)
```

$$\begin{bmatrix} \frac{3}{5} & -\frac{4}{5} \\ -\frac{4}{5} & -\frac{3}{5} \end{bmatrix}$$

15.6 Quadratic forms

15.6.1 Matrix of a quadratic form

The `q2a` command finds the matrix corresponding to a quadratic form.

- `q2a` takes two arguments:
  - q, the symbolic expression of a quadratic form q.
  - *vars*, a vector of variable names.
- `q2a(`q`,`*vars*`)` returns the matrix of the quadratic form q.

Example

```
> q2a(2*x*y,[x,y])
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

15.6.2 Transforming a matrix into a quadratic form

The `a2q` command finds the quadratic form corresponding to a symmetric matrix.

- `a2q` takes two arguments:
  - A, a symmetric matrix of a quadratic form.
  - *vars*, a vector of variable names whose size is the same as the number of rows of A.
- `a2q(`A`,`*vars*`)` returns the symbolic expression for the quadratic form.

Examples

```
> a2q([[0,1],[1,0]],[x,y])
```

$$2xy$$

```
> a2q([[1,2],[2,4]],[x,y])
```

$$x^2 + 4xy + 4y^2$$

15.6.3 Reducing a quadratic form

The `gauss` command uses Gauss's algorithm to write a quadratic form as a sum or difference of squares.

- `gauss` takes two arguments:
  - q, a symbolic expression representing a quadratic form.
  - *vars*, a vector of variable names.
- `gauss(`q`,`*vars*`)` returns q written as sum or difference of squares.

Example

```
> gauss(2*x*y,[x,y])
```

$$\frac{(x+y)^2}{2} - \frac{(-x+y)^2}{2}$$

15.6.4 Conjugate gradient algorithm

The `conjugate_gradient` command uses the conjugate gradient algorithm to solve a linear system of equations.

- `conjugate_gradient` takes two mandatory arguments and two optional arguments.
  - A, an $n \times n$ positive definite symmetric matrix.
  - y, a vector of length n.
  - Optionally, x_0, a vector of length n, an initial approximation.
  - ε, a positive number (by default `epsilon`, see Section 2.5.7, p. 15, item 2.5.7).
- `conjugate_gradient(`$A, y\,\langle, x_0, \varepsilon\rangle$`)` returns the solution to $Ax = y$ to within ε.

Examples

We can solve the system $\begin{cases} 2x + y = 1 \\ x + 5y = 0 \end{cases}$ by entering one of the following commandlines.

```
> conjugate_gradient([[2,1],[1,5]],[1,0])
```

$$\left[\frac{5}{9}, -\frac{1}{9}\right]$$

```
> conjugate_gradient([[2,1],[1,5]],[1,0],[0.55,-0.11],1e-2)
```

$$[0.555, -0.11]$$

```
> conjugate_gradient([[2,1],[1,5]],[1,0],[0.55,-0.11],1e-10)
```

$$[0.555555555556, -0.111111111111]$$

15.6.5 Gram-Schmidt orthonormalization

The `gramschmidt` command uses the Gram-Schmidt procedure to find an orthonormal set of vectors with the same span as a given set.

- `gramschmidt` takes one or two arguments:
  - A, a matrix viewed as a list of row vectors or a list of elements that is a basis of a vector subspace.
  - f, a function that defines a scalar product on this vector space. If A is a matrix, then this is optional, and by default will be the standard scalar product.
- `gramschmidt(`A`)` or `gramschmidt(`L, f`)` returns an orthonormal basis for this scalar product.

Examples

```
> normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]]))
```

or:

```
> normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]],dot))
```

$$\begin{bmatrix} \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ -\frac{\sqrt{6}}{6} & -\frac{\sqrt{6}}{6} & \frac{\sqrt{6}}{3} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \end{bmatrix}$$

Define a scalar product on the vector space of polynomials by $P \cdot Q = \int_{-1}^{1} P(x)Q(x)\,\mathrm{d}x$. Then:

```
> p_scal(p,q):=integrate(p*q,x,-1,1):;
  gramschmidt([1,1+x],p_scal)
```

$$\left[\frac{1}{\sqrt{2}}, \frac{1+x-1}{\frac{\sqrt{6}}{3}}\right]$$

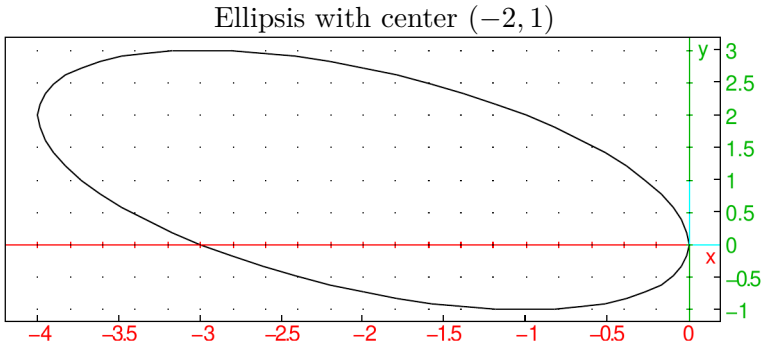
15.6.6 Graph of a conic

The `conic` command draws a conic.

- `conic` takes one mandatory argument and one or two optional arguments:
  - *eq*, the equation of a conic.
  - Optionally, *vars*, a list of the variables (by default $[x, y]$). The variables can also be given as two separate arguments.
- `conic(`*eq* $\langle$, *vars* $\rangle$`)` draws the conic.

Example

```
> conic(2*x^2+2*x*y+2*y^2+6*x)
```



See also the next section for the parametric equation of the conic.

15.6.7 Conic reduction

The `reduced_conic` command finds the reduced equation of a conic.

- `reduced_conic` takes two arguments:
  - *eq*, the equation of a conic.

  - *vars*, a list of the variable names.

- `reduced_conic(`*eq*, *vars*`)` returns a list whose elements are:
  - the origin of the conic,
  - the matrix of a basis in which the conic is reduced,
  - 0 or 1 (0 if the conic is degenerate),
  - the reduced equation of the conic
  - a vector of its parametric equations.

Example

```
> reduced_conic(2*x^2+2*x*y+2*y^2+5*x+3,[x,y])
```

$$\left[\left[-\frac{5}{3},\frac{5}{6}\right],\begin{bmatrix}\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2}\\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{bmatrix},1,3x^2+y^2-\frac{7}{6},\left[\frac{-10+5\mathrm{i}}{6}+\left(\frac{\sqrt{2}}{2}+\frac{1}{2}\mathrm{i}\sqrt{2}\right)\left(\frac{3}{18}\sqrt{14}\cos t+\frac{1}{6}\mathrm{i}\sqrt{42}\sin t\right),\right.$$
$$\left.t,0,2\pi,\frac{2}{60}\pi,2x^2+2xy+2y^2+5x+3,\frac{-10+5\mathrm{i}}{6}+\frac{\left(\frac{\sqrt{2}}{2}+\frac{1}{2}\mathrm{i}\sqrt{2}\right)\left(\frac{3}{18}\sqrt{14}\left(1-t^2\right)+\frac{2}{6}\mathrm{i}\sqrt{42}t\right)}{1+t^2}\right]\right]$$

This means that the conic is not degenerate, its reduced equation is

$$3x^2+y^2-\frac{7}{6}=0$$

its origin is $-5/3+5\mathrm{i}/6$, its axes are parallel to the vectors $(-1,1)$ and $(-1,-1)$, and its parametric equation is

$$\frac{-10+5\mathrm{i}}{6}+\frac{(1+\mathrm{i})}{\sqrt{2}}\cdot\frac{\left(\sqrt{14}\cos t+\mathrm{i}\sqrt{42}\sin t\right)}{6}$$

where the suggested parameter values for drawing are t from 0 to 2π with `tstep`$=2\pi/60$.

Remark. Note that if the conic is degenerate and is made of 1 or 2 line(s), the lines are not given by their parametric equation but by the list of two points of the line.

Example

```
> reduced_conic(x^2-y^2+3*x+y+2)
```

$$\left[\left[-\frac{3}{2},\frac{1}{2}\right],\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix},0,x^2-y^2,\begin{bmatrix}\frac{-3+\mathrm{i}}{2} & \frac{-1+3\mathrm{i}}{2}\\ \frac{-3+\mathrm{i}}{2} & \frac{-1-\mathrm{i}}{2}\end{bmatrix}\right]$$

15.6.8 Graph of a quadric

The `quadric` command draws a quadric.

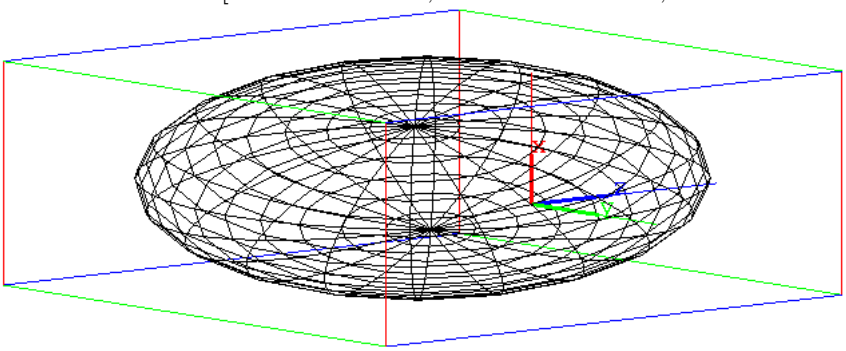
- `quadric` takes one mandatory argument and one optional argument:
  - q, the expression of a quadric.
  - Optionally, *vars*, a list of three variable names (by default, $[x,y,z]$). These names can also be given a separate arguments.

- `quadric(`$q\,\langle,$ *vars*$\rangle$`)` draws this quadric.

Example

```
> quadric(7*x^2+4*y^2+4*z^2+4*x*y-4*x*z-2*y*z-4*x+5*y+4*z-18)
```

Ellipsoid with center $[0.407407407407, -0.962962962963, -0.537037037037]$



See also the next section for the parametric equation of the quadric.

15.6.9 Quadric reduction

The `reduced_quadric` command finds the reduced equation of a quadric.

- `reduced_quadric` takes two arguments:
  - *eq*, the equation of a quadric.
  - *vars*, a vector of variable names.
- `reduced_quadric(`*eq*, *vars*`)` returns a list whose elements are:
  - the origin,
  - the matrix of a basis where the quadric is reduced,
  - 0 or 1 (0 if the quadric is degenerate),
  - the reduced equation of the quadric
  - a vector with its parametric equations.
- Note that `u,v` will be used as parameters of the parametric equations; these variables should not be assigned (purge them before calling `reduced_quadric`).

Example

```
> reduced_quadric(7*x^2+4*y^2+4*z^2+ 4*x*y-4*x*z-2*y*z-4*x+5*y+4*z-18)
```

$$\left[\left[\frac{11}{27},-\frac{26}{27},-\frac{29}{54}\right],\begin{bmatrix}\frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15}\\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6}\\ -\frac{\sqrt{6}}{6} & \frac{2}{5}\sqrt{5} & \frac{\sqrt{30}}{30}\end{bmatrix},[9,3,3],1,9x^2+3y^2+3z^2-\frac{602}{27},\right.$$
$$\left[\left[\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{3\cdot 243}+\frac{9\sqrt{5}\sqrt{602}\sin u\sin v}{5\cdot 81}-\frac{9\sqrt{30}\sqrt{602}\cos u}{15\cdot 81}+\frac{11}{27},\right.\right.$$
$$\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{6\cdot 243}+\frac{9\sqrt{30}\sqrt{602}\cos u}{6\cdot 81}-\frac{26}{27},$$
$$\left.-\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{6\cdot 243}+\frac{9\cdot 2\sqrt{5}\sqrt{602}\sin u\sin v}{5\cdot 81}+\frac{9\sqrt{30}\sqrt{602}\cos u}{30\cdot 81}-\frac{29}{54}\right],$$
$$\left.\left.u=0\ldots\pi,v=0\ldots 2\pi,\text{ustep}=\frac{\pi}{20},\text{vstep}=\frac{2}{20}\pi\right]\right]$$

The output is a list containing:

- The origin (center of symmetry) of the quadric

$$\left[\frac{11}{27}, -\frac{26}{27}, -\frac{29}{54}\right].$$

- The matrix of the basis change:

$$\begin{bmatrix} \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15} \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} \\ -\frac{\sqrt{6}}{6} & \frac{2}{5}\sqrt{5} & \frac{\sqrt{30}}{30} \end{bmatrix}.$$

- 1, hence the quadric is not degenerated.

- the reduced equation of the quadric:

$$9x^2 + 3y^2 + 3z^2 - \frac{602}{27}.$$

- The parametric equations (in the original frame):

$$\left[\left[\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{3\cdot 243} + \frac{9\sqrt{5}\sqrt{602}\sin u\sin v}{5\cdot 81} - \frac{9\sqrt{30}\sqrt{602}\cos u}{15\cdot 81} + \frac{11}{27},\right.\right.$$
$$\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{6\cdot 243} + \frac{9\sqrt{30}\sqrt{602}\cos u}{6\cdot 81} - \frac{26}{27},$$
$$\left.-\frac{9\sqrt{6}\sqrt{1806}\sin u\cos v}{6\cdot 243} + \frac{9\cdot 2\sqrt{5}\sqrt{602}\sin u\sin v}{5\cdot 81} + \frac{9\sqrt{30}\sqrt{602}\cos u}{30\cdot 81} - \frac{29}{54}\right],$$
$$\left.u = 0\ldots\pi, v = 0\ldots 2\pi, \text{ustep} = \frac{\pi}{20}, \text{vstep} = \frac{2}{20}\pi\right].$$

Hence the quadric is an ellipsoid $9x^2 + 3y^2 + 3z^2 + (-602)/27 = 0$. After the change of origin to $[11/27, (-26)/27, (-29)/54]$, the matrix of basis change is:

$$\begin{bmatrix} \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15} \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} \\ -\frac{\sqrt{6}}{6} & \frac{2\sqrt{5}}{5} & \frac{\sqrt{30}}{30} \end{bmatrix}.$$

Its parametric equation is:

$$\begin{cases} x = \frac{\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{3} + \frac{\sqrt{5}\sqrt{\frac{602}{81}}\sin(u)\sin(v)}{5} - \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u)}{15} + \frac{11}{27}, \\ y = \frac{\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{6} + \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u))}{6} - \frac{26}{27}, \\ z = \frac{-\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{6} + \frac{2\sqrt{5}\sqrt{\frac{602}{81}}\sin(u)\sin(v)}{5} + \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u)}{30} - \frac{29}{54}. \end{cases}$$

Remark. Note that if the quadric is degenerate and made of one or two plane(s), each plane is not given by its parametric equation but by the list of a point in the plane and a normal vector.

Example

```
> reduced_quadric(x^2-y^2+3*x+y+2)
```

$$\left[\left[-\frac{3}{2},\frac{1}{2},0\right],\begin{bmatrix}1&0&0\\0&1&0\\0&0&-1\end{bmatrix},\right.$$
$$[0,1,-1],x^2-y^2,$$
$$\left.\left[\text{hyperplan}\left([1,1,0],\left[-\frac{3}{2},\frac{1}{2},0\right]\right),\text{hyperplan}\left([1,-1,0],\left[-\frac{3}{2},\frac{1}{2},0\right]\right)\right]\right]$$

15.7 Linear systems

The *augmented matrix* of the system $AX = b$ is either the matrix obtained by gluing the column vector b to the right of the matrix A (as with `border(A,tran(b))`), representing $AX = b$, or the matrix obtained by gluing the column vector $-b$ to the right of the matrix A, representing $Ax - b = 0$.

15.7.1 Matrix of a system

The `syst2mat` command turns a system of linear equations into its augmented matrix. (For this command, the augmented matrix of $Ax = b$ has the column vector $-b$ glued to the right of A.)

- `syst2mat` takes two as arguments:
  - *eqns*, a list of the equations or expressions (assumed to be equal to zero) of a linear system.
  - *vars*, a list of the variable names.
- `syst2mat(`*eqns*, *vars*`)` returns the augmented matrix of the system.

Remark. Note that the variables must be purged before `syst2mat` is called.

Examples

```
> syst2mat([x+y,x-y-2],[x,y])
```

$$\begin{bmatrix}1&1&0\\1&-1&-2\end{bmatrix}$$

```
> syst2mat([x+y=0,x-y=2],[x,y])
```

$$\begin{bmatrix}1&1&0\\1&-1&-2\end{bmatrix}$$

15.7.2 Gauss reduction of a matrix

A matrix A is in row-echelon form if the first non-zero element of each row is 1 and each of these leading 1s is further right than the leading 1s of the preceding rows. Gaussian elimination will transform a matrix into row echelon form, and the row echelon form of the augmented matrix of a system of linear equations has the same set of solutions as the original, but in a form that is simple to solve.

The `ref` command transforms a matrix into a row echelon form of the matrix.

- `ref` takes A, a matrix.

- `ref(`A`)` returns a row echelon form of a matrix.

`ref` is typically used to solve a linear system of equations written in matrix form.

Example

Solve the system:

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

```
> ref([[3,1,-2],[3,2,2]])
```

$$\begin{bmatrix} 1 & \frac{1}{3} & -\frac{2}{3} \\ 0 & 1 & 4 \end{bmatrix}$$

Hence the solution is $y = 4$ (from the last row) and $x = -2$ (substitute y in the first row).

15.7.3 Gauss-Jordan reduction

The reduced row echelon form of a matrix is the row echelon form (see previous section) with zeros above the leading 1s in each row. Gauss-Jordan reduction will turn any matrix into reduced row echelon form, and the reduced row echelon form of a matrix is unique. If the matrix is the augmented matrix of a system, then the reduced row echelon form of the matrix is the simplest form to solve the system. The `rref` command finds the reduced row echelon form of a matrix (see also Section 11.8.17, p. 260).

- `rref` takes one mandatory and one optional argument:
  - A, a matrix.
  - Optionally, n, a positive integer, or `keep_pivot`, the symbol.
- `rref(`$A\,\langle, n, $`keep_pivot`$\rangle$`)` returns the reduced row echelon form of the matrix. With a second argument of n, Gauss-Jordan reduction will be performed on (at most) the first n columns. With the `keep_pivot` option, pivots are not normalized to 1s after reduction.

Examples

Solve the system:

$$\begin{cases} 3x + y &= -2, \\ 3x + 2y &= 2. \end{cases}$$

```
> rref([[3,1,-2],[3,2,2]])
```

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 4 \end{bmatrix}$$

Hence $x = -2$ and $y = 4$ is the solution of this system.

`rref` can also solve several linear systems of equations having the same matrix of coefficients by augmenting the matrix of coefficients by vectors representing the right hand side of the equations for each equation. For example, solve the systems:

$$\begin{cases} 3x + y &= -2, \\ 3x + 2y &= 2 \end{cases}$$

and

$$\begin{cases} 3x + y &= 1, \\ 3x + 2y &= 2. \end{cases}$$

```
> rref([[3,1,-2,1],[3,2,2,2]])
```

$$\begin{bmatrix} 1 & 0 & -2 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$$

Which means that $x = -2$ and $y = 4$ is the solution of the first system and $x = 0$ and $y = 1$ is the solution of the second system.

To perform Gauss-Jordan reduction only on the first column, input:

```
> rref([[3,1,-2,1],[3,2,2,2]],1)
```

$$\begin{bmatrix} 1 & \frac{1}{3} & -\frac{2}{3} & \frac{1}{3} \\ 0 & 3 & 12 & 3 \end{bmatrix}$$

15.7.4 Solving $AX = b$

The `simult` command can solve a linear system of equations or several linear systems of equations with the same matrix of coefficients (see also 15.7.3).

- `simult` takes two arguments:
  - A, a matrix (the matrix of coefficients of a system).
  - b, a column vector (representing the right hand side of the system) or a matrix (where each column represents the right hand side of an equation).
- `simult`(A, b) returns a column vector of the solutions (or a matrix where each column is the column vector of a solution).

Examples

Solve the system:

$$\begin{cases} 3x + y &= -2, \\ 3x + 2y &= 2. \end{cases}$$

```
> simult([[3,1],[3,2]],[[-2],[2]])
```

$$\begin{bmatrix} -2 \\ 4 \end{bmatrix}$$

Hence $x = -2$ and $y = 4$ is the solution.

Solve the systems:

$$\begin{cases} 3x + y &= -2, \\ 3x + 2y &= 2 \end{cases}$$

and

$$\begin{cases} 3x + y &= 1, \\ 3x + 2y &= 2. \end{cases}$$

```
> simult([[3,1],[3,2]],[[-2,1],[2,2]])
```

$$\begin{bmatrix} -2 & 0 \\ 4 & 1 \end{bmatrix}$$

Therefore $x = -2$ and $y = 4$ is the solution of the first system of equations and $x = 0$ and $y = 1$ is the solution of the second system.

15.7.5 Step by step Gauss-Jordan reduction of a matrix

One step of Gauss-Jordan elimination involves taking a non-zero element of a matrix (the pivot) and adding multiples of its row to the other rows to get zeros above and below the pivot. The `pivot` command performs this operation.

- `pivot` takes three arguments:
  - A, a a matrix with n rows and p columns.
  - l and c, two integers such that l is the index of a row of A and c is an index of column of A, and the element of A in row l and column c is non-zero.
- `pivot`(A, l, c) returns the result of performing one step of the Gauss-Jordan method using the element of A in row l, column c as pivot.

Examples

```
> pivot([[1,2],[3,4],[5,6]],1,1)
```

$$\begin{bmatrix} -2 & 0 \\ 3 & 4 \\ 2 & 0 \end{bmatrix}$$

```
> pivot([[1,2],[3,4],[5,6]],0,1)
```

$$\begin{bmatrix} 1 & 2 \\ 2 & 0 \\ 4 & 0 \end{bmatrix}$$

15.7.6 Solving linear systems

The `linsolve` command solves systems of linear equations. It can take its arguments as a list of equations or as a matrix of coefficients followed by a vector of the right-hand side. It can also take the matrix of coefficients after an LU factorization (see Section 15.3.5, p. 375), which can be useful if you have several systems which differ only by their right-hand side.

If the Step by step box is is checked in the CAS configuration (see Section 2.5.9, p. 18), `linsolve` will show you the steps in getting a solution.

Solving a system in symbolic form.

- To solve a system of symbolic equations, `linsolve` takes two arguments:
  - *eqns*, a list of linear equations or expressions (where each expression is assumed to be equal to zero).
  - *vars*, a list of variable names.
- `linsolve`$(eqns, vars)$ returns the solution of the equations as a list.

Examples

To solve the system

$$\begin{cases} 2x + y + z = 1, \\ x + y + 2z = 1, \\ x + 2y + z = 4, \end{cases}$$

input:

```
> linsolve([2*x+y+z=1,x+y+2*z=1,x+2*y+z=4],[x,y,z])
```

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2}\right]$$

Therefore $x = -\frac{1}{2}$, $y = \frac{5}{2}$ and $z = -\frac{1}{2}$.

An example of solving an underdetermined system:

```
> linsolve([x+2*y+3*z=1,3*x+2*y+z=2],[x,y,z])
```

$$\left[z + \frac{1}{2}, -2z + \frac{1}{4}, z\right]$$

Solving a system in matrix form.

- To solve a system represented by its coefficient matrix, `linsolve` takes two arguments:
  - A, the matrix of coefficients of a linear system.
  - b, a list of the values of the right hand side of the system.
- `linsolve(`A, b`)` returns the solution of the corresponding equations as a list.

Example

```
> linsolve([[2,1,1],[1,1,2],[1,2,1]],[1,1,4])
```

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2}\right]$$

If the Step by step option is checked in the CAS configuration, a window will also pop up showing:

$$\begin{bmatrix} 2 & 1 & 1 & -1 \\ 1 & 1 & 2 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

Reduce column 1 with pivot 1 at row 2

Exchange row 1 and row 2

$L_2 < -(1) \cdot L_2 - (2) \cdot L_1$ on $\begin{bmatrix} 1 & 1 & 2 & -1 \\ 2 & 1 & 1 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$

$L_3 < -1 \cdot L_3 - (1) \cdot L_1$ on $\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$

$\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 0 & 1 & -1 & -3 \end{bmatrix}$

Reduce column 2 with pivot 1 at row 3

Exchange row 2 and row 3

$L_1 < -(1) \cdot L_1 - (1) \cdot L_2 on \begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$

$L_3 < -(1) \cdot L_3 - (-1) \cdot L_2$ on $\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

Reduce column 3 with pivot -4 at row 3

$L_1 < -(1) \cdot L_1 - (-3/4) \cdot L_3$ on $\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$

$L_2 < -(1) \cdot L_2 - (1/4) \cdot L_3$ on $\begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$

End reduction $\begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & -\frac{5}{2} \\ 0 & 0 & -4 & -2 \end{bmatrix}$

$$\begin{bmatrix} 2 & 1 & 1 & -1 \\ 1 & 1 & 2 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

Reduce column 1 with pivot 1 at row 2

Exchange row 1 and row 2

$L_2 < -(1) \cdot L_2 - (2) \cdot L_1$ on $\begin{bmatrix} 1 & 1 & 2 & -1 \\ 2 & 1 & 1 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$

$L_3 < -(1) \cdot L_3 - (1) \cdot L_1$ on $\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$

$$\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 0 & 1 & -1 & -3 \end{bmatrix}$$

Reduce column 2 with pivot 1 at row 3

Exchange row 2 and row 3

$L_1 < -(1) \cdot L_1 - (1) \cdot L_2$ on $\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$

$L_3 < -(1) \cdot L_3 - (-1) \cdot L_2$ on $\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

Reduce column 3 with pivot -4 at row 3

$L_1 < -(1) \cdot L_1 - (-3/4) \cdot L_3$ on $\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$

$L_2 < -(1) \cdot L_2 - (1/4) \cdot L_3$ on $\begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$

End reduction $\begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & -\frac{5}{2} \\ 0 & 0 & -4 & -2 \end{bmatrix}$

Solving a system in factored form.

- To solve a system represented by its matrix in factored form, `linsolve` takes four arguments:

  - P, L, U, the LU decomposition of the matrix of coefficients.
  - b, a list of the values of the right hand side of the system.

- `linsolve`(P, L, U, b) returns the solution of the corresponding equations as a list.

Example

```
> p,l,u:=lu([[2,1,1],[1,1,2],[1,2,1]]);
  linsolve(p,l,u,[1,1,4])
```

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2}\right]$$

Solving a system with coefficients modulo n. The `linsolve` command also solves systems with coefficients in $\mathbb{Z}/n\mathbb{Z}$. For example:

```
> linsolve([2*x+y+z-1,x+y+2*z-1,x+2*y+z-4]%3,[x,y,z])
```

$$[1 \,\%\, 3, 1 \,\%\, 3, 1 \,\%\, 3]$$

15.7.7 Solving a linear system using the Jacobi method

The `jacobi_linsolve` command finds the solution of a linear system of equations using the Jacobi iterative method.

- `jacobi_linsolve` command takes two mandatory arguments and two optional arguments:
  - A, the matrix of coefficients of a system.
  - b, the right hand side of the system as a list.
  - Optionally, m, an integer indicating the maximum number of iterations (by default $m =$ `maxiter`, see Section 2.5.7, p. 15, item 2.5.7).
  - Optionally, ε, a positive number indicating the error tolerance (by default $\varepsilon =$ `epsilon`, see Section 2.5.7, p. 15, item 2.5.7).

  `jacobi_linsolve`$(A, b \,\langle, m, \varepsilon\rangle)$ returns the solution of the system.

Examples

```
> A:=[[100,2],[2,100]]:;
  jacobi_linsolve(A,[0,1],1e-12);
```

$$[-0.000200080032, 0.0100040016006]$$

```
> evalf(linsolve(A,[0,1]))
```

$$[-0.000200080032013, 0.0100040016006]$$

15.7.8 Solving a linear system using the Gauss-Seidel method

The `gauss_seidel_linsolve` command finds the solution of a linear system of equations using the Gauss-Seidel method.

- `gauss_seidel_linsolve` command takes two mandatory arguments and three optional arguments (including an optional first argument):
  - Optionally, ω, used for a general form of the Gauss-Seidel method (the successive overrelaxation method) (by default, $\omega = 1$).
  - A, the matrix of coefficients of a system.
  - b, the right hand side of the system as a list.
  - Optionally, ε, a positive number indicating the error tolerance (by default $\varepsilon =$ `epsilon`, see Section 2.5.7, p. 15, item 2.5.7).
  - Optionally, m, an integer indicating the maximum number of iterations (by default $m =$ `maxiter`, see Section 2.5.7, p. 15, item 2.5.7).

  `gauss_seidel_linsolve`$(A, b\langle\varepsilon, m\rangle)$ returns the solution of the system.

Examples

```
> A:=[[100,2],[2,100]]:;
  gauss_seidel_linsolve(A,[0,1],1e-12);
```

$$[-0.000200080032013, 0.0100040016006]$$

```
> gauss_seidel_linsolve(1.5,A,[0,1],1e-12);
```

$$[-0.000200080032218, 0.0100040016006]$$

15.7.9 Least squares solution of a linear system

The `lsq` or `LSQ` command finds the least squares solution to a matrix equation $AX = b$.

- `lsq` takes two arguments:
  - A, a matrix.
  - b, a vector or matrix.
- `lsq`(A, b) returns the least squares solution to the equation $AX = b$.

Examples

To solve $AX = b$ for $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$ and $b = \begin{bmatrix} 5 \\ 11 \end{bmatrix}$, input:

```
> lsq([[1,2],[3,4]],[5,11])
```

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

To solve $AX = B$ for A as above and $B = \begin{bmatrix} 5 & 7 \\ 11 & 9 \end{bmatrix}$, input:

```
> lsq([[1,2],[3,4]],[[5,7],[11,9]])
```

$$\begin{bmatrix} 1 & -5 \\ 2 & 6 \end{bmatrix}$$

Note that:

```
> linsolve([[1,2],[3,4],[3,6]]*[x,y]-[5,11,13],[x, y])
```

$$[]$$

since the linear system has no solution. You can still find the least squares solution:

```
> lsq([[1,2],[3,4],[3,6]],[5,11,13])
```

$$\begin{bmatrix} \frac{11}{5} \\ \frac{11}{10} \end{bmatrix}$$

The least squares solution:

```
> lsq([[3,4]],[12])
```

$$\begin{bmatrix} \frac{36}{25} \\ \frac{48}{25} \end{bmatrix}$$

represents the point on the line $3x + 4y = 12$ closest to the origin. Indeed:

```
> coordinates(projection(line(3*x+4*y=12),point(0)))
```

$$\left[\frac{36}{25}, \frac{48}{25}\right]$$

(See Section 26.12.4, p. 722, Section 26.14.8, p. 736, Section 19.3.1, p. 471 and Section 26.5.2, p. 685.)

15.7.10 Finding linear recurrences

The `reverse_rsolve` command finds a linear recurrence relation given the first few terms.

- `reverse_rsolve` takes $v = [v_0, \ldots, v_{2n-1}]$, a vector made of the first $2n$ terms of a sequence (v_n) which is supposed to verify a linear recurrence relation $x_n v_{n+k} + \cdots + x_0 v_k = 0$ of degree smaller than n, where the x_j are $n + 1$ unknowns.
- `reverse_rsolve(`v`)` returns the list $x = [x_n, \ldots, x_0]$ (if $x_n \neq 0$ then x_n is reduced to 1).

In other words, `reverse_rsolve` solves the linear system of n equations of the form

$$x_n v_{n+k} + \cdots + x_0 v_k = 0, \quad k = 0, 1, \ldots, n-1.$$

The matrix A of the system has n rows and $n + 1$ columns:

$$A = \begin{bmatrix} v_n & \cdots & v_0 & 0 \\ \vdots & & \vdots & \vdots \\ v_{n+k} & \cdots & v_k & 0 \\ \vdots & & \vdots & \vdots \\ v_{2n-1} & \cdots & v_{n-1} & 0 \end{bmatrix}$$

`reverse_rsolve` returns the solution $x = [x_n, \ldots, x_1, x_0]$ of $Ax = 0$ which satisfies $x_n = 1$.

Examples

Find a sequence satisfying a linear recurrence of degree at most 2 whose first elements are $1, -1, 3, 3$.

```
> reverse_rsolve([1,-1,3,3])
```

$$[1, -3, -6]$$

Hence $x_0 = -6$, $x_1 = -3$, $x_2 = 1$, and the recurrence relation is $v_{k+2} - 3v_{k+1} - 6v_k = 0$.

Find a sequence satisfying a linear recurrence of degree at most 3 whose first elements are $1, -1, 3, 3, -1, 1$.

```
> reverse_rsolve([1,-1,3,3,-1,1])
```

$$\left[1, -\frac{1}{2}, \frac{1}{2}, -1\right]$$

Hence, $x_0 = -1$, $x_1 = 1/2$, $x_2 = -1/2$, $x_3 = 1$, and the recurrence relation is

$$v_{k+3} - \frac{1}{2}v_{k+2} + \frac{1}{2}v_{k+1} - v_k = 0.$$

16 Optimization

16.1 Linear Programming

Linear programming problems involve maximizing a linear functionals under linear equality or inequality constraints. The simplest case can be solved directly by the so-called simplex algorithm. Most cases require you to solve an auxiliary linear programming problem to find an initial vertex for the simplex algorithm.

16.1.1 Simplex algorithm

The simple case. Linear programming problem in its simplest form is to find the maximum of an objective function

$$z(x_1, \ldots, x_n) = c_1 x_1 + \cdots + c_n x_n$$

subject to constraints (where $b_i \geq 0, i = 1, \ldots, m$):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &\leq b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &\leq b_m \end{aligned} \tag{16.1}$$

and

$$x_i \geq 0 \quad i = 1, \ldots, n.$$

This can be abbreviated as

$$\max(cx), \quad Ax \leq b,\ x \geq 0,\ b \geq 0,$$

where the vector inequalities mean term-by-term inequalities. The constraints determine a simplex in $\mathbb{R}^n$ and the standard approach to solving this problem is called the *simplex method*.

The simplex method. Xcas can apply the simplex method for you, but it can still help to get a rough idea of how it works. (You can search the Internet for "simplex method" to get more details.)

The constraint inequalities (16.1) can be turned into equalities by adding an auxiliary (surplus) variable for each inequality. The constraints are equivalent to:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n + s_1 + 0 + \cdots + 0 &= b_1 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n + 0 + \cdots + 0 + s_m &= b_m \end{aligned}$$

for some $s_1, \ldots, s_m \geq 0$. The straightforward solution to this system of equations, namely $x_i = 0$ for $i = 1, \ldots, n$ and $s_j = b_j$ for $j = 1, \ldots, m$, will be the initial solution. The objective function

$$\begin{aligned} z &= c_1 x_1 + \cdots + c_n x_n \\ &= c_1 x_1 + \cdots + c_n x_n + 0s_1 + \cdots + 0s_m \end{aligned}$$

will be zero at this solution. If one of the coefficients of the objective function is positive, you can find a larger value of x_i that will also be a solution of the equations (by making one of the s_j smaller) and will give a larger value of z.

The mechanics of this process involves writing the given information in a matrix. The matrix of the system of equalities including the surplus variables is $(AI_m b)$; the simplex method begins with this matrix above a row beginning with $-c$ followed by zeros. The last element of the last row is the value of the objective function at the current solution. The resulting matrix (the *simplex tableau*) looks like

$$\begin{bmatrix} A & I_m & b \\ -c & 0 & 0 \end{bmatrix}$$

The simplex method involves doing a specific series of row operations that effectively determine a new solution to the constraints that increase the values of the current values of the variables x_i and so increase the value of the objective function. The method ends when there are no negative values left in the bottom row.

The `simplex_reduce` command performs the simplex method on a linear programming problem of the form, for $b \geq 0$, $Ax \leq b$ with $x \geq 0$.

- `simplex_reduce` takes three arguments:
  - A, b and c from the problem.
- `simplex_reduce(`A, b, c`)` returns a sequence consisting of:
  - The maximum value of the objective function.
  - The solution of x (augmented since the algorithm works by adding auxiliary variables).
  - The reduced matrix.

Alternatively, you could combine the arguments A, b, c into the matrix that begins the algorithm:

$$\begin{bmatrix} A & I_m & b \\ -c & 0 & 0 \end{bmatrix}$$

which you can construct with the following command lines:

```
> B:=augment(A,idn(m));
  C:=border(B,b);
  d:=append(-c,0$(m+1));
  D:=augment(C,[d]);
  simplex_reduce(D)
```

The above matrix is obtained as D and passed to `simplex_reduce`.

Example

Maximize $X + 2Y$ subject to

$$\begin{aligned} -3X + 2Y &\leq 3, \\ X + Y &\leq 4, \\ X, Y &\geq 0. \end{aligned}$$

```
> simplex_reduce([[-3,2],[1,1]],[3,4],[1,2])
```

$$7, [1, 3, 0, 0], \begin{bmatrix} 0 & 1 & \frac{1}{5} & \frac{3}{5} & 3 \\ 1 & 0 & -\frac{1}{5} & \frac{2}{5} & 1 \\ 0 & 0 & \frac{1}{5} & \frac{8}{5} & 7 \end{bmatrix}$$

This means that the maximum of $X + 2Y$ under these conditions is 7, it is obtained for $X = 1, Y = 3$ because $[1, 3, 0, 0]$ is the augmented solution and the reduced matrix is:

$$\begin{bmatrix} 0 & 1 & \frac{1}{5} & \frac{3}{5} & 3 \\ 1 & 0 & -\frac{1}{5} & \frac{2}{5} & 1 \\ 0 & 0 & \frac{1}{5} & \frac{8}{5} & 7 \end{bmatrix}$$

Notice that the (non-zero) values of the solution have columns of the $m \times m$ identity matrix above them.

Reducing a more complicated case to the simple case. With the former call of `simplex_reduce`, you have to:

- rewrite constraints to the form $x_k \geq 0$,
- remove variables without constraints,
- add variables such that all the constraints have positive components.

For example, minimize $2x + y - z + 4$ subject to

$$\begin{aligned} x &\leq 1, \\ y &\geq 2, \\ x + 3y - z &= 2, \\ 2x - y + z &\leq 8, \\ -x + y &\leq 5. \end{aligned}$$

Let $x = 1 - X$, $y = Y + 2$, $z = 5 - X + 3Y$. The problem is equivalent to finding the minimum of $-2X + Y - (5 - X + 3Y) + 8$ under conditions

$$\begin{aligned} 2(1 - X) - (Y + 2) + 5 - X + 3Y &\leq 8, \\ -(1 - X) + (Y + 2) &\leq 5, \\ X, Y &\geq 0. \end{aligned}$$

or to find the minimum of $(-X - 2Y + 3)$ subject to

$$\begin{aligned} -3X + 2Y &\leq 3, \\ X + Y &\leq 4, \\ X, Y &\geq 0. \end{aligned}$$

i.e. to find the maximum of $-(-X - 2Y + 3) = X + 2Y - 3$ under the same conditions, hence it is the same problem as to find the maximum of $X + 2Y$ seen before. You found that the previous problem had a maximum of 7, hence the result here is $7 - 3 = 4$.

The general case. A linear programming problem may not in general be directly reduced like above to the simple case. The reason is that a starting solution must be found before applying the simplex algorithm.

The standard form of a linear programming problem is similar to the simplest case above, but with $Ax = b$ (instead of $Ax \leq b$) for $b \geq 0$ under the conditions $x \geq 0$. In this case, there is no straightforward solution. So the first problem is to find an x with $x \geq 0$ and $Ax = b$.

Finding an initial solution. Let m be the number of rows of A. Add artificial variables $s_1, \ldots, s_m$ and maximize $-\sum s_i$ under the conditions $Ax = b, x \geq 0, s \geq 0$ starting with initial value 0 for x variables and $y = b$. If a solution exists and is 0, then the s_i must all be 0 and so x will be solution of $Ax = b, x \geq 0$.

You can solve this with `simplex_reduce` by calling it with a single matrix argument:

$$\begin{bmatrix} A & I_m & b \\ 0 & 1 & 0 \end{bmatrix}$$

For example, to find the minimum of $2x + 3y - z + t$ with $x, y, z, t \geq 0$ and:

$$\begin{aligned} -x - y \quad + t &= 1 \\ y - z + t &= 3 \end{aligned}$$

you would start by finding a solution of

$$\begin{aligned} -x - y \quad + t + s_1 \quad &= 1 \\ y - z + t \quad + s_2 &= 3 \end{aligned}$$

as mentioned above.

```
> simplex_reduce([[-1,-1,0,1,1,0,1],[0,1,-1,1,0,1,3],[0,0,0,0,1,1,0]])
```

$$0, [0, 1, 0, 2, 0, 0], \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & \frac{1}{2} & \frac{1}{2} & 2 \\ \frac{1}{2} & 1 & -\frac{1}{2} & 0 & -\frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Since the solution $(x, y, z, t, s_1, s_2) = (0, 1, 0, 2, 0, 0)$ has $s_1 = s_2 = 0$, a non-negative solution of the equalities in x, y, z and t is $(x, y, z, t) = (0, 1, 0, 2)$.

Finding a solution of the LP problem. Now you can make a second call to `simplex_reduce` with the reduced matrix (less the columns corresponding to the variables s_1 and s_2) and the usual c. In this case, finding the requested minimum means finding the maximum of $-2x - 3y + z - t$, so $c = (-2, -3, 1, -1)$.

```
> simplex_reduce([[-1/2,0,-1/2,1,2],[1/2,1,-1/2,0,1],[2,3,-1,1,0]])
```

$$-5, [0, 1, 0, 2], \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & 2 \\ \frac{1}{2} & 1 & -\frac{1}{2} & 0 & 1 \\ 1 & 0 & 1 & 0 & -5 \end{bmatrix}$$

This maximum is -5, so the requested minimum is $-(-5) = 5$ at $(0, 1, 0, 2)$ ($x = 0$, $y = 1$, $z = 0$ and $t = 2$).

16.1.2 Solving (mixed integer) linear programming problems

The `lpsolve` command can solve linear programming problems, where a multivariate linear function needs to be maximized or minimized subject to linear (in)equality constraints, as well as (mixed) integer programming problems. You can either enter a problem directly (in symbolic or matrix form) or load it from a file in LP or (compressed) MPS format.

Solving an LP problem in symbolic form

`lpsolve` can solve LP problems in symbolic form in which the variables appear as symbols.

- `lpsolve` takes one mandatory argument and three optional arguments:

– *obj*, a symbolic expression representing the objective function or a path to a file containing the LP problem.

– Optionally, *constr*, list of linear constraints which may be equalities or inequalities or doubly bounded expressions in form $a \leq f(x) \leq b$ entered as `f(x)=a..b`. (If *obj* is a file name, this option is omitted.)

– Optionally, *bd*, a sequence of expressions of type `x=a..b`, specifying that the variable x is bounded below by $a \in \mathbb{R} \cup \{-\infty\}$ and above by $b \in \mathbb{R} \cup \{+\infty\}$. If the bounds are stored in a two-column matrix, consider using the conversion routine `box_constraints` (see Section 6.6.9).

– Optionally, *opts*, a sequence of solver settings in form *option=value*, where *option* may be one of:

  * `lp_assume` (or `assume`), which specifies a global constraint on variables and whose value can be one of:

    · `lp_nonnegative` (all variables are nonnegative)

    · `lp_integer` or `integer` (all variables are integers)

    · `lp_binary` (all variables are binary, i.e. 0 or 1)

    · `lp_nonnegint` or `nonnegint` (all variables are nonnegative integers)

    (by default, *unset*.)

  * `lp_integervariables`, whose value should be a list of identifiers or indices of integer variables (empty by default).

  * `lp_binaryvariables`, whose value should be a list of identifiers or indices of binary variables (empty by default).

  * `lp_maximize` (or `maximize`), whose value can be `true` or `false` setting the objective direction (by default `false`, meaning that the objective is minimized). You can enter only `maximize`, which is equivalent to `maximize=true`.

  * `lp_method`, setting the solver type, which can be one of: `exact`, `float`, `lp_simplex` (the default) or `lp_interiorpoint`.

  * `lp_depthlimit`, whose value can be a positive integer, which sets the maximum depth of the branch-and-bound tree (by default, *unlimited*).

  * `lp_nodelimit`, whose value can be a positive integer, which sets the maximum number of nodes in the branch-and-bound tree (by default, *unlimited*).

  * `lp_iterationlimit`, whose value can be a positive integer setting the maximum iterations of the simplex algorithm (by default, *unlimited*).

    If the maximum number of iterations is reached, the current feasible solution (not necessarily an optimal one) is returned.

  * `lp_timelimit`, whose value can be a positive real number, setting the maximum solving time in milliseconds (by default, *unlimited*).

  * `acyclic`, whose value can be either `true` or `false`, enabling/disabling the Bland rule safeguarding (by default, `true`).

  * `lp_maxcuts`, whose value can be a nonnegative integer setting the maximum GMI cuts per node (by default 5).

  * `lp_gaptolerance`, whose value can be a positive number, setting the relative integrality gap threshold (by default, 0).

  - `lp_presolve`, whose value can be either `true` or `false`, enabling/disabling the preprocessing step (by default, `true`).
  - `lp_heuristic`, whose value can be either `true` or `false`, enabling/disabling the rounding heuristic (by default, `true`).
  - `lp_nodeselect`, which sets the branching node selection strategy and whose value can be one of: `lp_depthfirst`, `lp_breadthfirst`, `lp_bestlocalbound` (the default), `lp_hybrid` or `lp_bestprojection`.
  - `lp_varselect`, which sets the branching variable selection strategy, whose value can be one of: `lp_firstfractional`, `lp_lastfractional`, `lp_mostfractional` or `lp_pseudocost` (the default).
  - `lp_verbose`, whose value can be `true` or `false` (by default `false`). You can enter only `lp_verbose`, which is equivalent to `lp_verbose=true`.

- `lpsolve(`*obj*⟨, *constr*, *bd*, *opts*⟩`)` returns a list [*optimum*, *soln*], where *optimum* is the minimum/-maximum value of the objective function and *soln* is the list of coordinates corresponding to the point at which the optimal value is attained, i.e. the optimal solution. If there is no feasible solution, an empty list is returned. When the objective function is unbounded, *optimum* is returned as `+infinity` (for maximization problems) or `-infinity` (for minimization problems). If an error is experienced while solving, then the process is terminated and `undef` is returned.

The given objective function is minimized by default. To maximize it, include the option `lp_maximize=true` or `lp_maximize` or simply `maximize`. Also note that all variables are, unless specified otherwise, assumed to be continuous and unrestricted in sign.

Examples

Minimize $2x + y - z + 4$ subject to

$$\begin{aligned} x &\le 1, \\ y &\ge 2, \\ x + 3y - z &= 2, \\ 2x - y + z &\le 8, \\ -x + y &\le 5. \end{aligned}$$

```
> lpsolve(2x+y-z+4,[x<=1,y>=2,x+3y-z=2,3x-y+z<=8,-x+y<=5])
```

$$[-4, [x = 0, y = 5, z = 13]]$$

Therefore, the minimum value of $f(x, y, z) = 2\,x + y - z + 4$ is equal to -4 under the given constraints. The optimal value is attained at point $(x, y, z) = (0, 5, 13)$.

Constraints may also take the form `expr=a..b` for bounded linear expressions. For example, minimize $x + 2y + 3z$ subject to $1 \le x + y \le 5$ and $2 \le y + z + 1 \le 4$, where $x, y \ge 0$.

```
> lpsolve(x+2y+3z,[x+y=1..5,y+z+1=2..4,x>=0,y>=0])
```

$$[-2, [x = 0, y = 5, z = -4]]$$

Use the `assume=lp_nonnegative` option to specify that all variables are nonnegative. It is easier than entering the nonnegativity constraints explicitly. For example, minimize $-x - y$ subject to $y \le 3x + \frac{1}{2}$ and $y \le -5x + 2$, where $x, y \ge 0$.

```
> lpsolve(-x-y,[y<=3x+1/2,y<=-5x+2],assume=lp_nonnegative)
```

$$\left[-\frac{5}{4}, \left[x = \frac{3}{16}, y = \frac{17}{16}\right]\right]$$

Bounds can be added separately for some variables. They should be entered after the constraints. For example, minimize $-6x + 4y + z$ subject to

$$5x - 10y \leq 20,$$
$$2z - 3y = 6,$$
$$-x + 3y \leq 3,$$

where $1 \leq x \leq 20$ and $y \geq 0$.

```
> lpsolve(-6x+4y+z,[5x-10y<=20,2z-3y=6,-x+3y<=3],x=1..20,y=0..inf)
```

$$\left[-\frac{133}{2}, \left[x = 18, y = 7, z = \frac{27}{2}\right]\right]$$

Solving an LP problem in matrix form

To enter a problem in matrix form:

- `lpsolve` takes
  - *obj*, a vector of coefficients representing the objective function.
  - *constr*, a list $[A, b, A_{\text{eq}}, b_{\text{eq}}]$ such that objective function $obj^T \cdot x$ is to be minimized/maximized subject to constraints $Ax \leq b$ and $A_{\text{eq}}x = b_{\text{eq}}$.

    If the problem does not contain equality constraints, A_{eq} and b_{eq} may be omitted. For a problem that does not contain inequality constraints, empty lists must be entered in place of A and b.
  - *bd*, a list of two vectors $[b_l, b_u]$ of the same length as c such that $b_l \leq x \leq b_u$. These vectors may contain `+infinity` or `-infinity`.
  - *opts*, as before.
- `lpsolve(`*obj*⟨, *constr*, *bd*, *opts*⟩`)` returns a list [*optimum*, *soln*] as before. *optimum* is the minimum/maximum value of the objective function and *soln* is the list of coordinates corresponding to the point at which the optimal value is attained, i.e. the optimal solution. If there is no feasible solution, an empty list is returned. When the objective function is unbounded, *optimum* is returned as `+infinity` (for maximization problems) or `-infinity` (for minimization problems). If an error is experienced while solving (terminating the process), `undef` is returned.

Examples

We minimize $-2x + y$ subject to $-x + y \leq 3$, $x + y \leq 5$, $x \geq 0$ and $y \geq 0$ by providing the parameters A and b.

```
> c:=[-2,1]:; A:=[[-1,1],[1,1],[-1,0],[0,-1]]:; b:=[3,5,0,0]:;
  lpsolve(c,[A,b])
```

$$[-10, [5, 0]]$$

You can enter variable bounds as the third argument. The following example minimizes $-2x+5y-3z$ subject to $2 \leq x \leq 6$, $3 \leq y \leq 10$ and $1 \leq z \leq \frac{7}{2}$.

```
> lpsolve([-2,5,-3],[],[[2,3,1],[6,10,7/2]])
```

$$\left[-\frac{15}{2}, \left[6, 3, \frac{7}{2}\right]\right]$$

To solve problems with only equality constraints, enter empty lists in places of A and b. The following example minimizes $4x + 5y$ subject to $-x + \frac{3}{2}y = 2$ and $-3x + 2y = 3$.

```
> lpsolve([4,5],[[],[],[[-1,3/2],[-3,2]],[2,3]])
```

$$\left[\frac{26}{5}, \left[-\frac{1}{5}, \frac{6}{5}\right]\right]$$

Simplex method implementation. Only the two-phase primal simplex method is implemented for `lpsolve` and it uses the upper-bounding technique. Basic columns are not kept in the simplex tableau, which therefore contains only the columns of non-basic variables. To prevent cycling, an adaptation of Bland's rule is used. A preprocessing routine, helping to reduce the size of the problem, is available and enabled by default (you can disable it by typing `lp_presolve=false`). All computation is done by using arbitrary-precision integer arithmetic, which is always exact but slower than the floating-point arithmetic. Note that problem data should be rational. (To solve LP problems in floating-point arithmetic, see Section 16.1.2, p. 412.)

Cycling in simplex algorithm may happen, although it is rare in practice. Bland rule safeguarding is used by default in order to prevent cycling. However, Bland's pivoting rule is known to converge slowly; therefore it may slow down the computation in problems which would otherwise not cycle. To disable the safeguarding, use the option `acyclic=false`.

Solving MIP (Mixed Integer Programming) problems

The `lpsolve` command allows restricting (some) variables to integer values. Such problems, called (*mixed*) *integer programming problems*, are solved by applying the branch-and-bound method.

Examples

To solve pure integer programming problems, in which all variables are integers, use the option `assume=integer` or `assume=lp_integer`. For example:

```
> lpsolve(-5x-7y,[7x+y<=35,-x+3y<=6],assume=integer)
```

$$[-41, [x = 4, y = 3]]$$

Use the option `assume=lp_binary` to specify that all variables are binary, i.e. the only allowed values are 0 and 1. These usually represent `false` and `true`, respectively, giving the variable a certain meaning in a logical context. For example:

```
> lpsolve(8x1+11x2+6x3+4x4,[5x1+7x2+4x3+3x4<=14],assume=lp_binary,maximize)
```

$$[21, [x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 1]]$$

To solve mixed integer problems, where some variables are integers and some are continuous, use the option keywords `lp_integervariables` to specify integer variables and/or `lp_binaryvariables` to specify binary variables. For example:

```
> lpsolve(x+3y+3z,[x+3y+2z<=7,2x+2y+z<=11],
    assume=lp_nonnegative,lp_maximize,lp_integervariables=[x,z]
```

$$[10, [x = 1, y = 0, z = 3]]$$

Use the `assume=lp_nonnegint` or `assume=nonnegint` option to get nonnegative integer values. For example:

```
> lpsolve(2x+5y,[3x-y=1,x-y<=5],assume=nonnegint)
```

$$[12, [x = 1, y = 2]]$$

When specifying MIP problems in matrix form, the lists corresponding to the options `lp_integervariables` and `lp_binaryvariables` should contain indices of the variables. For example:

```
> c:=[2,-3,-5]:; A:=[[-5,4,-5],[2,5,7],[2,-3,4]]:; b:=[3,1,-2]:;
  lpsolve(c,[A,b],lp_integervariables=[0,2])
```

$$\left[\frac{19}{4}, \left[1, \frac{3}{4}, -1\right]\right]$$

You can also specify a range of indices instead of a list when there is too much variables. For example, `lp_binaryvariables=0..99` means that all variables x_i such that $0 \leq i \leq 99$ are binary.

Branch-and-bound implementation. The branch-and-bound algorithm generates a binary tree of subproblems, called *nodes*, by branching on integer variables with fractional values. Leaf nodes of the tree, called *active nodes*, are stored in a list. In each iteration of the algorithm, an active node is selected, branched on a particular variable into two new nodes, and subsequently removed from the list. A node in which no branching is possible represents a feasible solution. The corresponding objective value is used to fathom nodes which cannot possibly lead to a better solution. The algorithm terminates when there is no space left for improvement.

If presolving is enabled, then basic preprocessing is done at each node of the tree, except when `lp_presolve=root` is set, in which case only the root node is processed. Additionally, after a non-integer-feasible solution with better objective value than the current incumbent is obtained by solving the linear relaxation, a rounding heuristic is applied in attempt to achieve integral feasibility. The heuristic is enabled by default; you can disable it by setting `lp_heuristic` to `false`.

Node-selection strategies. A node-selection strategy can be chosen by using the `lp_nodeselect` option. Possible values are:

- `lp_bestlocalbound`, which chooses an active node having the best bound for the objective value,
- `lp_depthfirst`, which chooses the newest active node,
- `lp_hybrid`, which combines the above two strategies,
- `lp_breadthfirst`, which chooses the oldest active node,
- `lp_bestprojection`, which chooses a node with the best simple projection.

By default, the `lp_bestlocalbound` strategy is used. The `lp_hybrid` strategy works as follows: until an incumbent solution is found, the solver uses the `lp_depthfirst` strategy, "diving" into the tree as an incumbent solution is more likely to be located deeply. When an incumbent is found, the solver switches to the `lp_bestlocalbound` strategy in attempt to close the integrality gap as quickly as possible.

Variable-selection strategies. A variable-selection strategy can be chosen by using the `lp_varselect` option. Possible values are:

- `lp_firstfractional`, which chooses the first fractional variable,
- `lp_lastfractional`, which chooses the last fractional variable,
- `lp_mostfractional`, which chooses a variable with fractional part closest to 0.5,
- `lp_pseudocost`, which chooses the variable which had the greatest impact on the objective value in previous branchings.

Node selection	Variable selection	Nodes examined	Time
Best local bound	Last fractional	13102	4.8
Best projection	Most fractional	26238	11.7
Hybrid	First fractional	55046	19.5
Depth-first	Pseudocost	131466	36.2

Table 16.1: Comparison of different variable and subproblem selection strategies for `lpsolve`

By default, the `lp_pseudocost` strategy is used. However, since a pseudocost-based choice cannot be made until all integer variables have been branched upon at least one time in each direction, the `lp_mostfractional` strategy is used until that condition is fulfilled.

Using an appropriate combination of node/variable selection strategies may significantly reduce the number of subproblems needed to be examined when solving a particular MIP problem, as illustrated in the following example.

Example

Minimize $z = \mathbf{c} \cdot \mathbf{x}$ subject to $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, where $\mathbf{x} \in \mathbb{Z}_+^8$ and

$$\mathbf{A} = \begin{bmatrix} 22 & 13 & 26 & 33 & 21 & 3 & 14 & 26 \\ 39 & 16 & 22 & 28 & 26 & 30 & 23 & 24 \\ 18 & 14 & 29 & 27 & 30 & 38 & 26 & 26 \\ 41 & 26 & 28 & 36 & 18 & 38 & 16 & 26 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 7872 \\ 10466 \\ 11322 \\ 12058 \end{bmatrix}, \quad \mathbf{c} = [2, 10, 13, 17, 7, 5, 7, 3]^T.$$

The optimal solution is $z^* = 1854$, $\mathbf{x}^* = (24, 15, 19, 11, 3, 99, 4, 226)$.

In Table 16.1, different strategies are compared according to the number of examined nodes and total solving time (in seconds). Note that the above comparison is problem-specific; it does not mean that `lp_bestlocalbound` with `lp_lastfractional` is always the best strategy. Usually, one has to experiment with different combinations to find which one is optimal for the given problem. However, the strategies which use larger amounts of information generally perform better. Therefore, it is reasonable to assume that `lp_bestlocalbound` will be more appropriate than `lp_breadthfirst`, for instance.

Cutting planes. Gomory mixed integer (GMI) cuts are generated at every node of the branch-and-bound tree to improve the objective value bound. After solving the relaxed subproblem using the simplex method, GMI cuts are added to the subproblem which is subsequently reoptimized. This procedure is repeated until no useful cuts can be generated or until `lp_maxcuts` limit is reached.

Simplex reoptimizations are fast because they start with the last feasible basis; however, applying cuts makes the simplex tableau larger, which may slow the computation down. To limit the number of GMI cuts that is allowed be appended to a subproblem, use `lp_maxcuts` option, setting it either to zero (which disables the cut generation altogether) or to some positive integer. You may set it to `+infinity` as well, thus allowing any number of cuts to be applied to a node. By default, `lp_maxcuts` equals to 5.

Stopping criteria. There are several ways to force the branch-and-bound algorithm to stop prematurely when the execution takes too much time. You can set `lp_timelimit` to an integer which defines the maximum number of milliseconds allowed to find an optimal solution. Other ways are to set `lp_nodelimit` or `lp_depthlimit` to limit the number of nodes generated in the branch-and-bound tree or its depth, respectively. Finally, you can set `lp_gaptolerance` to some positive value, say $t > 0$, which terminates the algorithm after finding an incumbent solution and proving that the corresponding objective value differs from the optimum value for less than $t \cdot 100\,\%$. This is done by monitoring the

size of the integrality gap, i.e. the difference between the current incumbent objective value and the best objective value bound among active nodes.

If the branch-and-bound algorithm terminates prematurely, a warning message indicating the cause is displayed. The incumbent solution, if any, is returned as the result, else the problem is declared to be infeasible.

Displaying detailed output. Typing `lp_verbose=true` or `lp_verbose` when specifying options for `lpsolve` causes detailed messages to be printed during and after solving a LP problem. During the simplex algorithm, a status report in form

$$n \texttt{ obj: } z$$

is printed every two seconds, where n is the number of simplex iterations and z is the current objective value. If the line is prefixed with the asterisk character (`*`), it means that the solver is optimizing the given objective (Phase 2); otherwise, the solver is searching for a feasible basis (Phase 1), in which case z is a relative value in percentages (when it reaches zero, Phase 2 is initiated). Note that values of z reported in Phase 2 may not correspond to the actual values if presolving is enabled.

If the problem contains integrality constraints, then the simplex algorithm messages are not printed. Instead, during the branch-and-bound algorithm, a status report in form

$$n\texttt{: } m \texttt{ nodes active, bound: } b\langle \texttt{ gap: } g\rangle$$

is displayed every 5 seconds, where n is the number of already examined subproblems, b is the lower resp. upper bound (for minimization resp. maximization), and g is the relative integrality gap. Also, a message is printed every time the incumbent solution is found or updated, as well as when the solver switches to pseudocost-based branching. After the algorithm is finished, a summary is displayed containing the total number of examined subproblems, the number of most nodes being active at the same time, and the number of applied GMI cuts along with the respective average objective value improvement.

Example

Here we solve the MIP given above. The solver shows all progress and summary messages.

```
> A:=[[22,13,26,33,21,3,14,26],[39,16,22,28,26,30,23,24],
      [18,14,29,27,30,38,26,26],[41,26,28,36,18,38,16,26]]:;
  b:=[7872,10466,11322,12058]:; c:=[2,10,13,17,7,5,7,3]:;
  lpsolve(c,[[],[],A,b],assume=nonnegint,lp_verbose)
Constraint matrix has 4 rows, 9 columns, and 36 nonzeros
Variables: 0 continuous, 8 integer (0 binary)
Constraints: 4 equalities, 0 inequalities
Optimizing...
  Starting branch & bound...
  11310: 147 nodes active, bound: 1793.43
  12709: Incumbent solution found: 1854
  20646: 931 nodes active, bound: 1841.81, gap: 0.657343%
  23836: Tree is empty
Summary:
  * 23836 subproblem(s) examined
  * max. tree size: 1256 nodes
  * 13 GMI cut(s) applied
```

$$[1854, [24, 15, 19, 11, 3, 99, 4, 226]]$$

Solving problems in floating-point arithmetic

The `lpsolve` command provides, in addition to its own exact solver implementing the primal simplex method with the upper-bounding technique, an interface to GLPK (GNU Linear Programming Kit) library which contains LP/MIP solvers operating in floating-point arithmetic, designed to be fast and to handle larger problems. Choosing between the available solvers is done by setting `lp_method` option.

By default, `lp_method` is set to `lp_simplex`, which solves the problem by using the native solver, but only if all problem coefficients are exact. If at least one of them is approximative (a floating-point number), then the GLPK solver is used instead.

Setting `lp_method` to `exact` forces the solver to perform exact computation even when some coefficients are inexact (they are converted to rational equivalents before applying the simplex method). In this case the native solver is used.

Setting `lp_method` to `float` forces `lpsolve` to use the GLPK simplex solver. If a (mixed) integer problem is given, then the branch-and-cut algorithm in GLPK is used. The parameters can be controlled by setting the `lp_timelimit`, `lp_iterationlimit`, `lp_gaptolerance`, `lp_maxcuts`, `lp_heuristic`, `lp_verbose`, `lp_nodeselect`, and `lp_varselect` options. If `lp_varselect` is not set, then the Driebeek–Tomlin heuristic is used, and if `lp_nodeselect` is not set, then the best-local-bound selection method is used. If `lp_maxcuts` is greater than zero, then GMI/MIR cut generation is enabled, else it is disabled. If the problem contains binary variables, then cover/clique cut generation is enabled, else it is disabled. If `lp_heuristic=false`, then the simple rounding heuristic in GLPK is disabled (by default it is enabled). Finally, `lp_verbose=true` enables GLPK messages, which are useful for monitoring solver's progress.

Setting `lp_method` to `lp_interiorpoint` uses the primal-dual interior-point algorithm in GLPK. The only optional argument that affects this kind of solver is `lp_verbose`. The interior-point algorithm is faster than the simplex method for large and sparse LP problems. Note, however, that it does not support integrality constraints.

Example

We solve the following LP problem using the default settings.

$$\text{Minimize } 1.06\,x_1 + 0.56\,x_2 + 3.0\,x_3$$

subject to

$$\begin{aligned} 1.06\,x_1 + 0.015\,x_3 &\geq 729824.87 \\ 0.56\,x_2 + 0.649\,x_3 &\geq 1522188.03 \\ x_3 &\geq 1680.05 \end{aligned}$$

```
> lpsolve(1.06x1+0.56x2+3x3,
    [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05])
```

$$[2255937.4968, [x_1 = 688490.254009, x_2 = 2716245.85277, x_3 = 1680.05]]$$

If the requirement $x_k \in \mathbb{Z}$ for $k = 1, 2$, the following result is obtained from the MIP solver in GLPK.

```
> lpsolve(1.06x1+0.56x2+3x3,
    [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05],
    lp_integervariables=[0,1])
```

$$[2255938.37, [x_1 = 688491, x_2 = 2716246, x_3 = 1680.05]]$$

The solution of the original problem can also be obtained with the interior-point solver by using the `lp_method=lp_interiorpoint` option.

```
> lpsolve(1.06x1+0.56x2+3x3,
    [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05],
    lp_method=lp_interiorpoint)
```

$$[2255937.50731, [x_1 = 688490.256652, x_2 = 2716245.85608, x_3 = 1680.05195065]]$$

Loading problems from files

Linear (integer) programming problems can be loaded from MPS or CPLEX LP format files. The file name should be a string passed as the *obj* parameter. If the file name has extension `.lp`, then CPLEX LP format is assumed. If the extension is `.mps` resp. `.gz`, then MPS resp. gzipped MPS format is assumed.

For example, assume that the file `somefile.lp` is stored in the working directory and that it contains the following lines of text:

```
Maximize
  obj: 2x1+4x2+3x3
Subject To
  c1: 3x1+5x2+2x3 <= 60
  c2: 2x1+x2+2x3  <= 40
  c3: x1+3x2+2x3  <= 80
General
  x1 x3
End
```

To find an optimal solution to the linear program specified in this file, enter:

```
> lpsolve("somefile.lp")
```

$$[71.8, [x_1 = 0, x_2 = 5.2, x_3 = 17]]$$

You can provide additional variable bounds, assumptions, and options alongside the file name, as in the examples below. Note that the original constraints (those which are read from file) cannot be removed.

```
> lpsolve("somefile.lp",x2=5.4..inf,lp_method=exact)
```

$$\left[\frac{352}{5}, \left[x_1 = 0, x_2 = \frac{28}{5}, x_3 = 16\right]\right]$$

Next we force all variables to integral values.

```
> lpsolve("somefile.lp",x2=5.4..inf,assume=integer)
```

$$[69.0, [x_1 = 0, x_2 = 6, x_3 = 15]]$$

It is advisable to use only (capital) letters, digits and underscores when naming variables in an LP file, although the corresponding format allows more characters. That is because these names are converted to `giac` identifiers during the loading process.

Note that the coefficients of a problem loaded from file are always floating-point values. Therefore, to solve the problem with the native solver, you have to use the option `lp_method=exact` (see Section 16.1.2, p. 412).

16.1.3 Transportation problem

The objective of a transportation problem is to minimize the cost of distributing a product from m sources to n destinations. It is determined by three parameters:

- $\mathbf{s} = (s_1, s_2, \ldots, s_m)$, the supply vector, where $s_k \in \mathbb{Z}^+$ is the maximum number of units that can be delivered from the kth source for $k = 1, 2, \ldots, m$.
- $\mathbf{d} = (d_1, d_2, \ldots, d_n)$, the demand vector $\mathbf{d} = (d_1, d_2, \ldots, d_n)$, where $d_k \in \mathbb{Z}^+$ is the minimum number of units required by the kth destination for $k = 1, 2, \ldots, n$.
- $\mathbf{C} = [c_{ij}]_{m \times n}$, the cost matrix, where $c_{ij} \in \mathbb{R}^+$ is the cost of transporting one unit of product from the ith source to the jth destination, for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

An optimal solution is represented by the matrix $\mathbf{X}^* = (x_{ij}^*)$, where x_{ij}^* is number of units that must be transported from the ith source to the jth destination for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, n$.

The `tpsolve` command solves a transportation problem.

- `tpsolve` takes three arguments:
  - s, the supply vector.
  - d, the demand vector.
  - C, the cost matrix.
- `tpsolve`(s, d, C) returns a sequence c, X^*, where X^* is the optimal solution and $c = \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij}\, x_{ij}^*$ is the total (minimal) cost of transportation.
- If the total supply and total demand are equal, i.e. if $\sum_{i=1}^{m} s_i = \sum_{j=1}^{n} d_j$ holds, the transportation problem is said to be *closed* or *balanced*, otherwise it is said to be *unbalanced*. The excess supply/demand is covered by adding a dummy demand/supply point with zero cost of "transportation" from/to that point. The `tpsolve` command handles such cases automatically.
- Sometimes it is desirable to forbid transportation on certain routes. That is achieved by setting a very high cost, represented by a symbol (e.g. M), to these routes. All instances of the symbol in the cost matrix are automatically replaced by a practically infinite number (precisely $(mn+1)$/`epsilon`, which forces the algorithm to avoid the associated routes.

Examples

```
> s:=[12,17,11]:; d:=[10,10,10,10]:;
  C:=[[50,75,30,45],[65,80,40,60],[40,70,50,55]]:;
  tpsolve(s,d,C)
```

$$2020, \begin{bmatrix} 0 & 0 & 2 & 10 \\ 0 & 9 & 8 & 0 \\ 10 & 1 & 0 & 0 \end{bmatrix}$$

```
> s:=[7,10,8,8,9,6]:; d:=[9,6,12,8,10]:;
  C:=[[36,40,32,43,29],[28,27,29,40,38],[34,35,41,29,31],
      [41,42,35,27,36],[25,28,40,34,38],[31,30,43,38,40]]:;
  tpsolve(s,d,C)
```

$$1275, \begin{bmatrix} 0 & 0 & 2 & 0 & 5 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 8 & 0 \\ 9 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \end{bmatrix}$$

```
> s:=[95,70,165,165]:; d:=[195,150,30,45,75]:;
  C:=[[15,M,45,M,0],[12,40,M,M,0],[0,15,25,25,0],[M,0,M,12,0]]:;
  tpsolve(s,d,C)
```

$$2820, \begin{bmatrix} 20 & 0 & 0 & 0 & 75 \\ 70 & 0 & 0 & 0 & 0 \\ 105 & 0 & 30 & 30 & 0 \\ 0 & 150 & 0 & 15 & 0 \end{bmatrix}$$

16.2 Analytical methods for nonlinear optimization

16.2.1 Constrained global optimization

Functions of one variable. For univariate minimization and maximization you can use the `fMin` and `fMax` commands.

- `fMin` and `fMax` both take two arguments:
  - *expr*, a univariate differentiable expression.
  - *var*, which is either x, a variable, or x=a`..`b where a and b are two real numbers such that $a < b$ (it is allowed to enter `-inf` for a or `inf` for b).
- `fMin(`*expr*, *var*`)` (or analogously with `fMax`) returns the sequence of all points of global minimum (maximum) of *expr* in the specified interval or segment. If infinity is returned, then *expr* is unbounded.
- If `fMin` or `fMax` fails to find exact points of minima or maxima, they will be computed numerically and returned as floating-point values.

Functions of several variables. The `minimize` and `maximize` commands attempt to find, using analytical methods, the exact minimal and maximal value of a differentiable expression on a compact domain specified by (in)equality constraints.

- `minimize` and `maximize` both take two mandatory arguments and two optional arguments:
  - *obj*, a univariate or multivariate expression
  - Optionally, *constr*, list of constraints given by equalities, inequalities, and/or expressions assumed to be equal to 0. If there is only one constraint, it does not have to be a list.
  - *vars*, list of variables. If there is only one variable, it does not have to be a list. A variable can also include bounds, as in $x = a..b$.

    Variables can also be given as $x = x_0$, $y = y_0$ and so on, in which case the optimum is computed numerically by performing a local search from the specified point $(x_0, y_0, \dots)$.
  - Optionally, `point`.
- `minimize(`*obj*⟨`,` *constr*⟩`,` *vars*⟨`,` `point`⟩`)` (or analogously with `maximize`) returns the minimum (maximum) value of `obj` on the domain specified by constraints and/or bounding variables, or (if `point` is given) a list consisting of the minimum value and the list of points where the minimum is achieved.
- If the domain is not compact, the final result may be incorrect or meaningless. If the minimal value could not be found, then `undef` is returned. In some cases, unboundeness of the objective function can be detected.
- Note that `minimize` and `maximize` respect the bound constraints set to variables by the `assume` command.

Examples

Univariate optimization:

```
> fMax(x*exp(-2x^3),x=0..1)
```

$$\left(\frac{1}{6}\right)^{\frac{1}{3}}$$

```
> fMin(sinc(x),x=0..20)
```

$$4.49340945791$$

```
> minimize(sin(x),x=0..4)
```

$$\sin(4)$$

```
> minimize(x^4-x^2,x=-3..3,point)
```

$$\left[-\frac{1}{4}, \left[\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2}\right]\right]$$

`minimize` and `maximize` can handle absolute values and piecewise expressions in one or more variables. For example, we find global minimum and global maximum of the function

$$f(x,y) = \frac{|2+xy-y^2| + |2x-x^2+xy+y^2|}{1+x^2+y^2}$$

on the square $[-1,1] \times [-1,1]$.

```
> f(x,y):=(abs(2+x*y-y^2)+abs(2x-x^2+x*y+y^2))/(1+x^2+y^2):;
  minimize(f(x,y),[x=-1..1,y=-1..1]);
  maximize(f(x,y),[x=-1..1,y=-1..1]);
```

$$\frac{1}{3}, \frac{\sqrt{5}+3}{2}$$

As another example, input:

```
> obj:=piecewise(x<=-2,x+6,x<=1,x^2,3/2-x/2):;
  maximize(obj,x=-3..2)
```

$$4$$

Each constraint can be either an equality or a non-strict inequality.

```
> obj:=sqrt(x^2+y^2)-z:;
  constr:=[x^2+y^2<=16,x+y+z=10]:;
  minimize(obj,constr,[x,y,z])
```

$$-4\sqrt{2}-6$$

```
> minimize(x^2*(y+1)-2y,[y<=2,sqrt(1+x^2)<=y],[x,y])
```

$$-4$$

`minimize` and `maximize` are aware of implied constraints that restrict the natural domain of the objective function. In the following example, the constraint $x^2+y^2 \leq 1$ is implied, and the minimum corresponds to any point on the unit circle.

```
> minimize(sqrt(1-x^2-y^2),[x,y],point)
```

$$\left[0, \begin{bmatrix} \sqrt{-y^2+1} & y \\ -\sqrt{-y^2+1} & y \\ x & \sqrt{-x^2+1} \\ x & -\sqrt{-x^2+1} \end{bmatrix}\right]$$

As another example, it is known that the natural domain of arc sine is the segment $[-1, 1]$.

```
> minimize(asin(x),x)
```

$$-\frac{\pi}{2}$$

Constraints can be built from simpler ones by using logical conjunction and disjunction. The latter is suitable for specifying disconnected domains, as in the following example.

```
> minimize(x*y,[x^2+y^2<=1,x<=-3/4 or x>=5/6],[x,y],point)
```

$$\left[-\frac{3\sqrt{7}}{16}, \begin{bmatrix} -\frac{3}{4} & \frac{\sqrt{7}}{4} \end{bmatrix}\right]$$

Symbolic constants are allowed in the objective and constraints. Note that they have to be either implied or assumed real numbers.

```
> assume(a>0):;
  maximize(x^2*y^2*z^2,x^2+y^2+z^2=a^2,[x,y,z])
```

$$\frac{a^6}{27}$$

In the following example, we use `minimize` to obtain the formula for computing the distance d between the point $P = (a, b, c)$ and the plane π given by the equation $Ax + By + Cz + D = 0$. We accomplish this by finding a point $(x_0, y_0, z_0) \in \pi$ which is closest to P. Since we are going to minimize $d^2 = (x-a)^2 + (y-b)^2 + (z-c)^2$, the distance is equal to the square root of the minimum.

```
> d:=minimize((x-a)^2+(y-b)^2+(z-c)^2,A*x+B*y+C*z+D=0,[x,y,z]):;
  simplify(sqrt(d))
```

$$\frac{|Aa + Bb + Cc + D|}{\sqrt{A^2 + B^2 + C^2}}$$

Note that the coordinates x_0, y_0 and z_0 can be obtained by passing `point` as the fourth argument in `minimize`.

16.2.2 Local extrema

The `extrema` command attempts to find local extrema of a univariate/multivariate differentiable expression, possibly subject to equality constraints, by using analytical methods.

- `extrema` takes two mandatory arguments and two optional arguments:
  - *expr*, a differentiable expression.
  - Optionally, *constr*, a list of equality constraints, where each constraint is an equality or an expression assumed to be equal to zero, and the number of constraints must be strictly less than the number of variables. If there is only one constraint, it does not have to be a list.

    Additionally, the Jacobian matrix of the constraints must be full rank (i.e., denoting the kth constraint by $g_k(x_1, x_2, \ldots, x_n) = 0$ for $k = 1, 2, \ldots, m$ and letting $\mathbf{g} = (g_1, g_2, \ldots, g_m)$, the Jacobian matrix of $\mathbf{g}$ must be equal to m).

  - *vars*, a list of variables. A variable can be specified with bounds, $x = a..b$, where a or b is allowed to be `-infinity` or `infinity`. Note that `extrema` respects the bound constraints set to variables by the `assume` command.

    If there is only one variable, it does not have to be enclosed in square brackets.

    The parameter `vars` can also be entered as a list of values of the variables; e.g. $[x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n]$, in which case the critical point close to $a = (a_1, a_2, \ldots, a_n)$ is computed numerically by applying an iterative method with initial point a.
  - Optionally, *opt*, which can be either
    * `order=`n, to make n the upper bound for the order of derivatives examined in the process (so if $n = 1$ only critical points are found, by default $n = 5$), or
    * `lagrange` to specify the method of Lagrange multipliers.

- `extrema(`*expr* ⟨, *constr*⟩, *vars* ⟨, *opt*⟩`)` returns a list [*min*,*max*], where *min* is the list of local minima and *max* is the list of local maxima of `expr`. Saddle, unclassified and indeterminate points are reported in the message area. If `lagrange` is passed as an optional last argument, the method of Lagrange multipliers is used. Else, the problem is reduced to an unconstrained one by applying implicit differentiation.

- If some critical points are left unclassified, you might consider repeating the process with larger values of the parameter `order` (e.g. for $f(x) = x^8$, `extrema` requires $n \geq 8$ in order to find the minimum), although the success is not guaranteed. Indeterminate critical points are unclassifiable.

Examples

```
> extrema(-2*cos(x)-cos(x)^2,x)
```

$$\begin{bmatrix} 0 \\ \pi \end{bmatrix}$$

```
> extrema(x/2-2*sin(x/2),x=-12..12)
```

$$\begin{bmatrix} -\frac{10}{3}\pi & \frac{2}{3}\pi \\ -\frac{2}{3}\pi & \frac{10}{3}\pi \end{bmatrix}$$

```
> assume(a>=0):; extrema(x^2+a*x,x)
```

$$\left[\left[-\frac{1}{2}a\right], []\right]$$

```
> extrema(exp(x^2-2x)*ln(x)*ln(1-x),x=0.5)
```

$$[[], [0.277769149124]]$$

```
> extrema(x^3-2x*y+3y^4,[x,y])
```

[x=0,y=0]: saddle point

$$\left[\left[\frac{12^{\frac{1}{5}}}{3} \quad \frac{\left(12^{\frac{1}{5}}\right)^2}{6}\right], []\right]$$

```
> assume(a>0):; extrema(x/a^2+a*y^2,x+y=a,[x,y],lagrange)
```

$$\left[\left[\frac{2a^4-1}{2a^3} \quad \frac{1}{2a^3}\right], []\right]$$

```
> extrema(x^2+y^2,x*y=1,[x=0..inf,y=0..inf])
```

$$\left[\left[1 \quad 1\right], []\right]$$

To find only the critical points of $f(x, y, z) = xyz$ subject to $x + y + z = 1$:

```
> extrema(x*y*z,x+y+z=1,[x,y,z],order=1)
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

The Peano surface $z = (2x^2 - y)(y - x^2)$ was proposed by Giuseppe Peano in 1899 as a counter-example to a criterion for the existence of maxima and minima which was conjectured at the time. Indeed, it is not possible to classify its only critical point $(0, 0)$, which turns out to be a saddle, not a maximum:

```
> extrema((2x^2-y)*(y-x^2),[x,y])
```

```
[x=0,y=0]: indeterminate critical point
```

$$[[], []]$$

16.3 Numerical methods for nonlinear optimization

16.3.1 Univariate global minimization on a segment

The `find_minimum` command is used for global minimization of a continuous function on a segment.

- `find_minimum` takes between three and five arguments:
  - f, a continuous univariate function or expression.
  - a, a real number.
  - b, a real number such that $a < b$.
  - Optionally, a sequence of one or two numbers, each of which being either:
    - $\varepsilon \in \langle 0, 1 \rangle$, the optimality tolerance (by default, $\varepsilon =$ `epsilon()`$^{2/3}$), or
    - $n \in \mathbb{Z}$, the maximum number of iterations ($n \geq 1$ is required) for Brent's method subroutine (by default, $n = 100$).
- `find_minimum(`$f, a, b\,\langle, \varepsilon, n\,\rangle$`)` returns the location of the smallest local minimum found.
- `find_minimum` applies a modification of Brent's algorithm which usually finds a global minimum, even when f has several local extrema. The original Brent's method, which searches for a local minimum, is used as a subroutine which is applied recursively by partitioning the search interval. Each instance is allowed the maximum of n iterations.

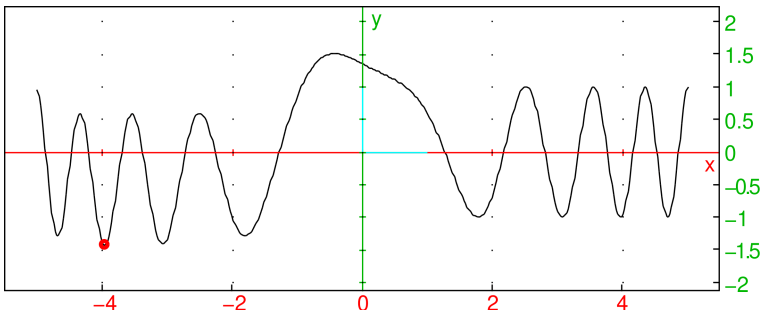
Examples

We minimize a function which has several local extrema, one of which is global.

```
> f(x):=Airy_Ai(x+sin(x))+cos(x^2):;
  x0:=find_minimum(f(x),-5,5)
```

$$-3.96344565209$$

```
> plot(f(x),x=-5..5); point(x0,f(x0),display=point_width_3+point_point+red)
```

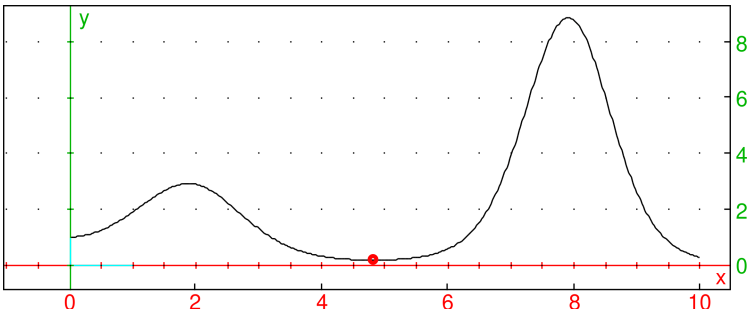


The objective function in the example below is unimodal, with a single local minimum which is also the global minimum.

```
> f(x):=besselJ(x,2)/Gamma(x+1)+(x+1)^sin(x):;
  x0:=find_minimum(f(x),0,10)
```

$$4.82843934915$$

```
> plot(f(x),x=0..10); point(x0,f(x0),display=point_width_3+point_point+red)
```

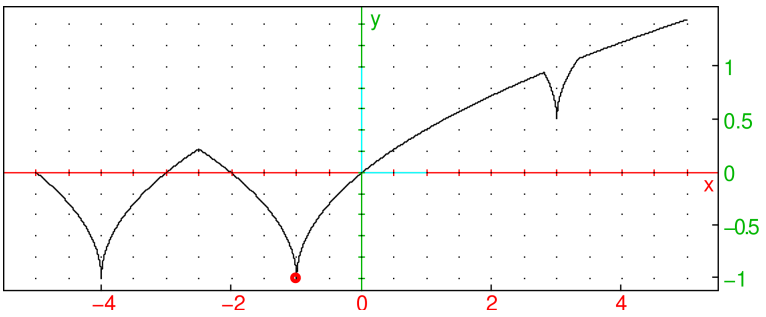


The function f defined below is continuous but not differentiable. It has three local and only one global minimum at $x_0 = -1$. Note that we are passing the function itself as the first argument. Also, the parameters ε and n are set to 10^{-5} and 30, respectively.

```
> f(x):=min(sqrt(abs(x+4))-1,sqrt(abs(x+1))-1005/1000,sqrt(abs(x-3))+1/2):;
  x0:=find_minimum(f,-5,5,1e-5,30)
```

$$-0.999998230068$$

```
> plot(f(x),x=-5..5,xstep=5e-4);
  point(x0,f(x0),display=point_width_3+point_point+red)
```



16.3.2 Minimization on bounded convex polyhedra

The `frank_wolfe` command finds the minimum of a (twice) differentiable convex function $f : \mathbb{R}^n \to \mathbb{R}$ on a bounded convex polyhedron defined by the set $A \subset \mathbb{R}^n$ of its vertices by using the method of Frank and Wolfe with backtracking. The problem can be written as:

$$\min_{\mathbf{x} \in \operatorname{conv} A} f(\mathbf{x}), \tag{16.2}$$

where $\operatorname{conv} A$ denotes the convex hull of A which coincides with the given polyhedron.

Note that (16.2) can be solved efficiently for $n = 1$ by using the `find_minimum` command (see Section 16.3.1, p. 419). Therefore, `frank_wolfe` requires that $n \geq 2$.

- `frank_wolfe` takes two mandatory arguments and a sequence of optional arguments:
  - *obj*, which is either an **expression** $f(\mathbf{x})$ (with at least two variables) or a vector containing two or three **functions** with arguments $x_1, x_2, \ldots, x_n$, the first one returning $f(\mathbf{x})$ and the second one returning $\nabla f(\mathbf{x})$. If the `rand` option is used (see below), then the third function should compute the Hessian matrix $\nabla^2 f(\mathbf{x})$, otherwise it can be omitted.
  - A, a finite set of $m \geq 2$ points in $\mathbb{R}^n$ given as a $m \times n$ matrix in which the ith row corresponds to the ith point.
  - Optionally, *opts*, a sequence of optional arguments each of which is one of:
    - ε, a positive floating-point number specifying the tolerance (by default, $\varepsilon = 0.001$).
    - N, a positive integer specifying the maximal number of iterations (by default, $N = 10^5$).
    - *initp*, a point in $\mathbb{R}^n$ given as a vector of n real numbers, specifying the initial point $\mathbf{x}_0$ (by default, $\mathbf{x_0}$ is the mean of points in A, which is always feasible).
    - `exact` or `exact=`*niter*, a symbol specifying that the exact line search using Brent's method should be performed. By default, the `step_size` subroutine by Pedregosa et al. (2020) is used, which is faster. The parameter *niter* specifies the maximal number of iterations for the exact line-search and its default value is 100. If *niter* is not specified, then the globally optimal step size in $[0, 1]$ is searched for, otherwise a locally optimal one is returned.
    - `rand` or `rand=`*npts*, a symbol specifying that the minimization should be initiated from several random points instead of a single point to avoid being stuck in a local minimum. Initial points are found by generating *npts* (by default 1000) random points in $\operatorname{conv} A$ at which the objective function is convex. The generated points are clustered and the Frank and Wolfe algorithm is run from the center of each cluster.
- `frank_wolfe(`*obj*, A, $\langle$*opts*$\rangle$`)` returns a feasible point $\mathbf{x}$ such that $|f(\mathbf{x}^*) - f^*| \leq \varepsilon$, where f^* is the actual optimal objective function value in (16.2) (note that local convexity of f is required to guarantee this). If in some iteration, say kth, the step-size is zero to working precision, or the maximal number of iterations is exceeded, the return value will be a vector of two elements: a string containing the warning message and the last point $\mathbf{x}_k$.
- If *obj* is given as an expression, then the gradient and Hessian information are computed automatically.
- If the algorithm fails to minimize to the given precision, it is a good idea to call `frank_wolfe` again with the last point as the initial point and watch for a possible improvement of the objective function value.
- `frank_wolfe` behaves well with optimal solution(s) lying at the boundary of the feasible set, since the Frank and Wolfe method is automatically feasible.

- Exact line search is several times slower than the default method but usually makes the total number of the main-loop iterations smaller.
- The `rand` option helps handling problems with non-convex objective functions. The algorithm finds an appropriate number of initial points in the set $\{\mathbf{x} \in \mathrm{conv} A : \nabla^2 f(\mathbf{x})$ is positive definite$\}$ and (re)starts from each of them, thereby finding the global minimum in most cases. If this approach fails, a warning message is printed an the provided initial point is used instead, or one is generated as described above.

Examples

The Lasso Problem. Solve the following ℓ^1-constrained problem in n dimensions:

$$\min \frac{1}{2}\|A\mathbf{x} - \mathbf{b}\|^2, \quad \|\mathbf{x}\| \le 1,$$

where A is a $m \times n$ matrix with real coefficients, $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{b} \in \mathbb{R}^m$. It can be shown that vertices of the polyhedron defined by $\|\mathbf{x}\|_1 \le 1$ are defined by the orthonormal basis vectors and their counterparts ($2n$ vertices in total). Hence the set of vertices V can be represented by a matrix with $2n$ rows and n columns which is obtained by gluing the identity matrix I_n and its negative $-I_n$ together:

$$V = \begin{bmatrix} I_n \\ -I_n \end{bmatrix}.$$

Since the objective function is convex, the problem can be solved by the Frank-Wolfe method. Let $n = 10$ and A be a randomly generated square matrix with coefficients in $[-1, 1]$.

```
> n:=10:; A:=ranm(n,n,uniformd(-1,1)):;
```

Now let $\mathbf{y} \in \mathbb{R}^n$ be a random point on the sphere with radius $r = 1.1$. Then define $\mathbf{b} = A\mathbf{y}$. Thus the global minimum of f is located near the polyhedron, but outside it.

```
> y:=ranv(n,randnorm):; y:=1.1*y/l2norm(y):; b:=y*tran(A):;
```

Define V, $\mathbf{x}$, f and call `frank_wolfe`. You get a vector which depends on the choices of A and $\mathbf{y}$.

```
> V:=tran(augment(idn(n),-idn(n))):;
  x:=symbol_array("x",n,purge):;
  f:=simplify(1/2*l2norm(x*tran(A)-b)^2):;
  fw:=frank_wolfe(f,V)
```

$$[0.0, 0.0, 0.08905128279, 0.0853981396774, 0.0, 0.0, 0.469342995518, 0.0, -0.355655381935, 0.0]$$

The optimal value of the objective function is:

```
> subs(f,x,fw)
```

$$1.0435553106$$

COBYLA, for instance, returns a weaker solution (see Section 16.3.3, p. 426), which is expected since it does not benefit from gradient information:

```
> cb:=fMin(f,[l1norm(x)<=1],x,[0.01$n],1e-4):;
  subs(f,x,cb)
```

$$1.16374646004$$

The strength of the Frank-Wolfe method, besides automatic feasibility, is its ability to detect whether the objective function value is sufficiently close to the (unknown) otimal value.

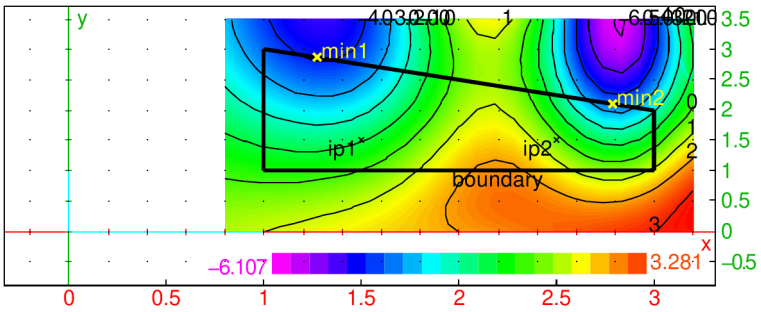
Non-convex objectives. Suppose that you want to minimize the function $f(x,y) = x\cos(y) - y\sin(x^2)$ inside the quadrilateral Q with vertices $(1,1)$, $(1,3)$, $(3,2)$ and $(3,1)$. The function f has two local minima in Q, one of which is global. To run `frank_wolfe` from the points $(\frac{3}{2},\frac{3}{2})$ and $(\frac{5}{2},\frac{3}{2})$, enter:

```
> purge(x,y):;
  f,Q:=x*cos(y)-y*sin(x^2),[[1,1],[1,3],[3,2],[3,1]]:;
  ip1,ip2:=[1.5,1.5],[2.5,1.5]:;
  min1,min2:=frank_wolfe(f,Q,ip1),frank_wolfe(f,Q,ip2)
```

$$[1.26907401345, 2.86473876147], [2.78442772106, 2.10749634066]$$

To plot the function f, quadrilateral Q, initial points and obtained minima, enter:

```
> plotdensity(f,[x=0.8..3.2,y=0.0..3.5]);
  polygon(op(B),color=line_width_3+quadrant4,legend="boundary");
  point(min1,color=yellow+point_width_2,legend="min1");
  point(min2,color=yellow+point_width_2,legend="min2");
  point(ip1,legend="ip1"); point(ip2,legend="ip2")
```



Points `min1` and `min2` are the global and a local minimum, respectively:

```
> subs(f,[x,y],min1),subs(f,[x,y],min2)
```

$$-4.08322337291, -3.52045371621$$

The algorithm gets stuck in the local minimum `min2` when starting from ip_2 because it is not allowed to go uphill. To find the global minimum one must apply multi-start, which you can select by using the `rand` option. Indeed:

```
> frank_wolfe(f,Q,rand)
```

$$[1.26907407408, 2.86474054616]$$

As another example, `frank_wolfe` is used to find the global minimum of the *six-hump camel function*:

$$f(x,y) = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + (-4 + 4y^2)y^2$$

in the rectangle $[-3,3] \times [-2,2]$ (see here). The function has six local minima, two of which are global: $(0.0898, -0.7126)$ and $(-0.0898, 0.7126)$. Indeed:

```
> f,R:=(4-2.1*x^2+x^4/3)*x^2+x*y+(-4+4*y^2)*y^2,[[-3,-2],[-3,2],[3,2],[3,-2]]:;
  frank_wolfe(f,R,rand=10000)
```

$$[0.0898388758644, -0.712673226867]$$

By evaluating the above commandline several times in a row, you always get one of the two global minima.

Boundary discretization. When minimizing on a convex domain, you can simplify the problem by discretizing the domain boundary. For example, minimize

$$f(x,y) = 4\mathrm{e}^x + 5\cos y - x^2 y + 4x + 5y - 25\mathrm{e}^{-(x-1)^2-(y-2)^2}$$

in the convex set $D \subset \mathbb{R}^2$ bounded by the closed parametric curve C defined by

$$x(t) = \left(2 + \cos\left(t - \frac{\pi}{6}\right)\right)\sin(\sin t), \quad y(t) = 2(2+\cos t)\cos\left(t + \frac{\pi}{6}\right), \quad 0 \le t \le 2\pi.$$

Let $A = \left\{(x(t_k), y(t_k)) : t_k = \frac{2k\pi}{n}, k = 1, 2, \ldots, n\right\}$ for some $n \ge 2$. Now the convex hull of A can be used as an approximation of D. To define f and the C (with u and v instead of x and y), enter:

```
> f:=4*exp(x)+5*cos(y)-x^2*y+4x+5y-25*exp(-(x-1)^2-(y-2)^2):;
  u(t):=(2+cos(t-pi/6))*sin(sin(t)):;
  v(t):=2*(2+cos(t))*cos(t+pi/6):;
```

Now generate the hull vertices with, say, $n = 1000$. Since `linspace` includes both 0 and 2π in `s`, the former is discarded by using `tail`.

```
> s:=evalf(tail(linspace(0,2*pi,1000))):;
  A:=tran(apply(u,s),apply(v,s)):;
```

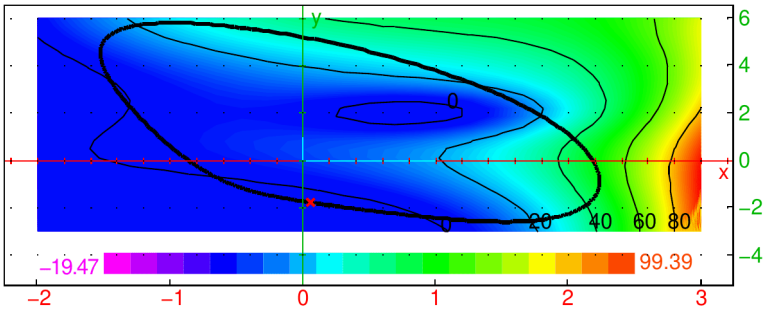
Use the `rand` option when calling `frank_wolfe` because the function f is not convex has several local minima in D.

```
> p:=frank_wolfe(f,A,rand)
```

$$[0.0565029231227, -1.78083520026]$$

To plot the function f, curve C and point p, enter:

```
> densityplot(f,[x=-2..3,y=-3..6]);
  paramplot([u(t),v(t)],t=0..2pi,display=line_width_3);
  point(p,display=point_width_2+red)
```

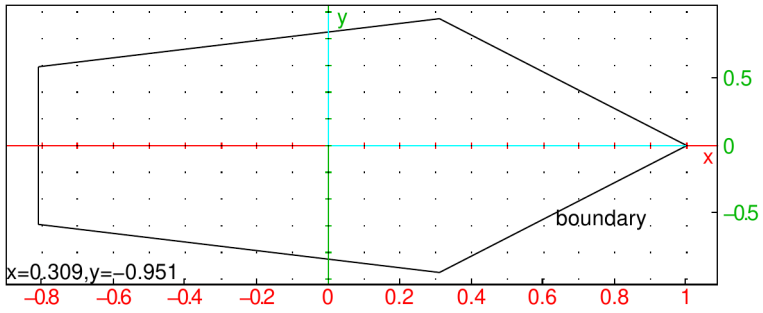


Input with functions. In some problems, computing the objective function value and/or its derivative may be too complex for automatic handling. In such cases you should provide `giac` functions which compute these values. For example, suppose you wish to minimize the function

$$f(x,y) = -\mathrm{Ai}(1+x)\cos\left(y - \frac{\pi}{8}\right), \quad x \ge -1,$$

where Ai is the Airy function of the first kind, on the domain enclosed by the pentagon with vertices corresponding to complex solutions of the equation $z^5 = 1$. To define the boundary, enter:

```
> boundary:=polygon(op(cZeros(z^5=1,z)),display=quadrant4)
```



To define the objective function, enter:

```
> f(x,y):=-Airy_Ai(1+x)*cos(y-pi/8):;
```

Computing the gradient of f automatically is difficult because the `grad` command does not know how to compute the derivative of Ai function, so you will have to provide the gradient function yourself. By the relation between Airy functions and modified Bessel functions, you get:

$$\mathrm{Ai}'(z) = -\frac{z}{\pi\sqrt{3}} K_{2/3}(\zeta),$$

where $\zeta = \frac{2}{3}z^{3/2}$ and

$$K_\nu(z) = \frac{\sqrt{\pi}}{2^\nu \Gamma(\nu+1/2)} \frac{\mathrm{e}^{-z}}{\sqrt{z}} \int_0^\infty \left(2+\frac{t}{z}\right)^{\nu-1/2} t^{\nu-1/2}\mathrm{e}^{-t}\mathrm{d}t.$$

The above integral is easily approximated with Gaussian quadrature for $z > 0$ real. Therefore, the derivative of Ai can be computed for positive arguments by using the following `giac` function:

```
Ai_diff:=proc(x)
  local zeta,K,K0,t;
  zeta:=2/3*x^(3/2);
  purge(t);
  K0:=exp(-zeta)/(4^(1/3)*Gamma(7/6)*sqrt(zeta));
  K:=K0*int((2t+t^2/zeta)^(1/6)*exp(-t),t=0..inf);
  return evalf(-x/sqrt(3*pi)*K);
end
```

Now, with some elementary differentiation by hand or by the help of `grad`, we get the gradient of f:

$$\nabla f(x,y) = \begin{bmatrix} -\mathrm{Ai}'(1+x)\cos\left(y-\frac{\pi}{8}\right) \\ \mathrm{Ai}(1+x)\sin\left(y-\frac{\pi}{8}\right) \end{bmatrix},$$

which is computed by the function

```
g:=proc(x,y)
  return evalf([-Ai_diff(x+1)*cos(y-pi/8),Airy_Ai(x+1)*sin(y-pi/8)]);
end
```

To get the coordinates of pentagon vertices, enter (note the conversion to floating-point values):

```
>   A:=evalf(coordinates(vertices(boundary)))
```

$$\begin{bmatrix} 0.309016994375 & -0.951056516295 \\ -0.809016994375 & -0.587785252292 \\ -0.809016994375 & 0.587785252292 \\ 0.309016994375 & 0.951056516295 \\ 1.0 & 0.0 \end{bmatrix}$$

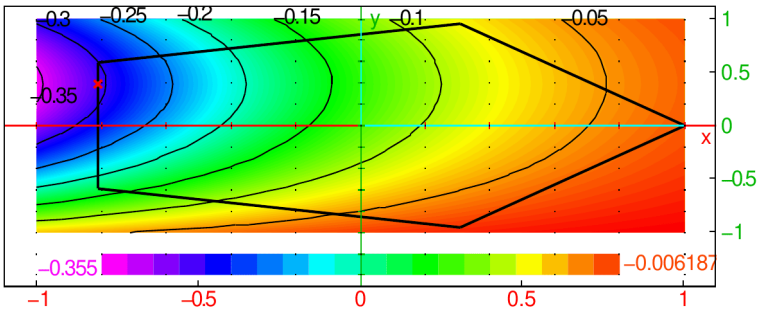
Now find the minimum of f inside the pentagon:

```
> frank_wolfe([f,g],evalf(coordinates(vertices(p))))
```

$$[-0.809016994375, 0.393184912674]$$

Indeed:

```
> plotdensity(f(x,y),[x=-1..1,y=-1..1]);
  display(boundary,line_width_2);
  point(p,display=point_width_2+red)
```



Note that using the `rand` option in this case would also require a function which computes the Hessian of f at a given point.

16.3.3 Derivative-free constrained optimization

The `fMin` and `fMax` commands find local minima and maxima of a multivariate function, possibly subject to constraints (either equalities or inequalities), by using the COBYLA algorithm.

- `fMax` and `fMin` take four mandatory and two optional arguments:
  - *expr*, an expression with several variables, which does not need to be differentiable.
  - *constr*, a list of constraints (equalities and/or inequalities).
  - *vars*, a list of the variables.
  - *init*, an initial guess, which must be a list of nonzero reals representing a starting point (not necessarily feasible).
  - Optionally, ε, the precision. If this is not given, the default epsilon value is used (see Section 2.5.7, p. 15, item 2.5.7).
  - Optionally, N, the maximum number of iterations.
- `fMin(`*expr*, *constr*, *vars*, *init* $\langle, \varepsilon, N \rangle$`)` (and analogously with `fMax`) returns a point that minimizes (maximizes) *expr* subject to *constr*.
- If there are several feasible local minima/maxima, then the return value depends on the initial point.

Examples

```
> fMax((x-2)^2+(y-1)^2,[-.25x^2-y^2+1>=0,x-2y+1=0],[x,y],[.5,.75])
```

$$[-1.82287565553, -0.411437827766]$$

```
> fMin((x-5)^2+y^2-25,[y>=x^2],[x,y],[1,1])
```

$$[1.2347728625, 1.52466402196]$$

16.3.4 Solving general nonlinear programming problems

The `nlpsolve` command computes the optimum of a multivariate objective function subject to equality and/or inequality constraints with continuous and/or integer variables.

- `nlpsolve` takes the following arguments:
  - *obj* or *fname*, an expression to optimize or a string containing the path to a problem file.
  - Optionally, *constr*, a list of equality and inequality constraints. Double-bounded constraints in form $a \leq f(x) \leq b$ can be entered as `f(x)=a..b` instead of two inequalities.
  - Optionally, *bd*, a sequence of variable boundaries `x=a..b` where $a \in \mathbb{R} \cup \{-\infty\}$ and $b \in \mathbb{R} \cup \{+\infty\}$, $b \geq a$. If several variables have the same upper and lower bounds, one can enter `[x,y,...]=a..b`. If the bounds are stored in a two-column matrix, consider using the conversion routine `box_constraints` (see Section 6.6.9).
  - Optionally, *opt*, a sequence of options in which each option may be one of:
    - `maximize=`*bool* or `nlp_maximize=`*bool*, where *bool* can be `true` or `false`. Just `maximize` or `nlp_maximize` is equivalent to `maximize=true`. (By default, `maximize=false`, i.e. the objective is minimized.)
    - `nlp_initialpoint=`*pt*, where *pt* is a starting point for the solver or a list of starting points, each of which is given in the form $[x = x_0, y = y_0, \dots]$. Alternatively, *pt* may be an initial search rectangle specified by $[x = x_{\min}..x_{\max}, y = y_{\min}..y_{\max}, \dots]$ which serves as a frame for automatic generation of starting points.
    - `nlp_method=`*meth*, which is a string or a list specifying the method of optimization. Available methods are:
      - `nelder-mead` or `nm`, the simplex method of Nelder and Mead, modified so that it can handle variable bounds and (in)equality constraints; see Luersen, Riche, and Guyon (2003).
      - `differential-evolution` or `de`, the method of differential evolution, modified so it can handle bounds, (in)equalities and integrality constraints; see Lampinen and Zelinka (2000).
      - `interior-point` or `ipt`, the interior-point method which requires the objective and constraint derivatives, and can handle (mixed) integer nonlinear problems. If GSL routines are available, then unconstrained problems are solved by BGFS method which is faster.
      - `cobyla` or `cbl`, the (derivative-free) COBYLA algorithm.

      Note that these names must be entered with double quotes, i.e. as strings. Some parameters can be set for individual methods; to do so, *meth* should be a list in which the leading element is the method specification, followed by one or more entries of the form "*param*"=*value*, where *param* can be one of the following (see below for detailed explanations on values):
      - `cross-probability`, which specifies the probability of mutation in differential evolution, where *value* $= p \in \langle 0, 1 \rangle$ (by default, $p = 0.5$).
      - `scale`, which specifies the scaling factor for differential evolution (by default, it is computed automatically from `cross-probability` by using the formula of Zaharie (2002)).
      - `reflect-ratio`, which specifies the simplex reflection factor in Nelder-Mead method (by default 1).

- `expand-ratio`, which specifies the simplex expansion factor in Nelder-Mead method (by default 2).
- `contract-ratio`, which specifies the simplex contraction factor in Nelder-Mead method (by default 0.5).
- `shrink-ratio`, which specifies the simplex shrinking factor in Nelder-Mead method (by default 0.5).
- `size`, which specifies the initial simplex size for Nelder-Mead method (by default, it is computed automatically by using the variable bounds).
- `step`, which specifies initial step for BGFS method or adaptive penalty step for Nelder-Mead method (by default, it is set to 1 for BGFS and to 0.01 for Nelder-Mead).
- `search-points`, which specifies the number of agents in differential evolution (by default $10n$, where n is the number of variables) or the number of points for determining PDH clusters (by default $1000\sqrt{n}$).
- `postprocess`, which specifies whether the result obtained by differential evolution or Nelder-Mead method should be refined by a local method, where *value* is a boolean value (by default, *value* = `true`).

The optimization method is determined automatically if *meth* is not set (see below).

* `nlp_precision=`ε for some $\varepsilon > 0$, which sets the optimality tolerance for the solver (by default, $\varepsilon =$ `epsilon()`$^{2/3}$).
* `nlp_tolerance=`δ for some $\delta > 0$, which sets the feasibility tolerance for the solver (by default, $\delta = 10^{-5}$).
* `nlp_presolve=`*bool*, where *bool* can be either `true` or `false`. This option enables/disables the preprocessing step which attempts to reduce the number of decision variables by solving subsets of equality constraints which are all linear in certain subsets of variables, substituting the results, and repeating the process until no further reduction can be made. (In particular, fixed variables are removed.) After solving the presolved problem, the removed variables are substituted back to the solution. (By default, `nlp_presolve=true`.)
* `border=`*bool*, where *bool* is a boolean value. This enables or disables finding variable bounds automatically. (By default, `border=true`.)
* `nlp_integervariables=`*lst*, where *lst* is a list of problem variables that are supposed to take integral values (empty by default).
* `nlp_binaryvariables=`*lst*, where *lst* is a list of problem variables that are supposed to take binary (0-1) values (empty by default).
* `nlp_iterationlimit=`N for a positive integer N, which sets the maximum number of iterations allowed for the solver (by default, $N = 250$).
* `nlp_verbose=`v, where v is an integer specifying the verbosity level: 0 – no messages, 1 – only errors, 2 – errors and warnings, 3 – all messages (by default, $v = 1$).
* `cluster=`k or *bool*, where $k \geq 2$ is an integer and *bool* is either `true` or `false`, which specifies whether the initial points should be determined by k-means clustering a large number of random points at which the objective Hessian is positive definite (this works only if the interior-point method is used), called PDH (Positive Definite Hessian) points. The number of generated points can be specified by setting the `search-points` parameter. If `cluster=true`, the number of clusters is determined automatically using

the Hartigan criterion. If k is set, then k clusters are generated. Cluster centers are subsequently used as the initial points. (By default, `cluster=false`.)

  - `convex` or `convex=`*bool*, where *bool* can be either `true` or `false`. This specifies whether the input problem should be treated as being convex. If this option is set, then the convexity check is not performed, speeding up the execution. (By default, convexity check is performed only if it is relevant for choosing a suitable optimization method.)
  - `assume=`*asmp*, where *asmp* may be one of the following:
    - `nlp_nonnegative`, which restricts all variables to nonnegative values (existing positive lower bounds are kept, however),
    - `nlp_nonnegint` or `nonnegint`, which makes all variables take nonnegative integer values,
    - `nlp_integer` or `integer`, which forces all variables to take integer values,
    - `nlp_binary`, which forces all variables to take 0-1 values.

- `nlpsolve(`*obj*⟨, *constr*, *bd*, *opt*⟩`)` returns a list [*optobj*,*optdec*], where *optobj* is the optimal value of *obj* and *optdec* is a list of of optimal values of the decision variables. If the optimization fails, then *optobj* is the error message and *optdec* is the last point on the path the solver takes in attempt to find the optimal objective value. If the problem is infeasible or if the solver fails to find a feasible point, then an error is returned.
- `nlpsolve(`*fname*⟨, *opt*⟩`)` solves the problem written in the file pointed to by *fname*, which must be written in the AMPL modeling language (file extension `.mod`). `nlpsolve` contains a very basic parser for such files and is able to read problem variables, objective and constraints (this works with problems from MINLPLib). Solver parameters can be specified in *opt*.

Automatic selection of the optimization method. If objective and constraints are twice differentiable, then interior-point method is used. If no initial point is provided and there are only box constraints, then differential evolution is applied first to find a good starting point. In the non-differentiable case, `nlpsolve` applies the Nelder-Mead method and switches to differential evolution in the case of failure. In case of integrality constraints, as well as in the case of a box-constrained problem without additional constraints and initial point(s), differential evolution is selected. Linear problems are solved by the `lpsolve` command, while univariate optimization on a segment is performed by using Brent's algorithm.

Local vs. global optimization. When an initial point is provided, `nlpsolve` acts like a local optimizer. Without initial point(s), it uses a multistart technique in attempt to find the global minimum: starting points are automatically generated by using the probabilistic restart technique described by Luersen, Riche, and Guyon (2003) until the maximum number of iterations is reached. This technique avoids previously generated points and also the points of convergence, while keeping relatively close to them; thus, the search space is examined in a probabilistic but systematic way. An exception is the method of differential evolution, which is a "global" method by design (however not a true one); the initial population is generated uniformly if the search space is compact.

Automatic detection of variable bounds. If the constraint derivatives are available, `nlpsolve` determines rough bounds of the decision variables, and is thus able to detect whether the search domain is compact. In particular, this information is used by the probabilistic restart technique.

(Mixed) integer optimization. When integrality constraints are provided, `nlpsolve` applies a suitable method for finding integer-feasible solutions. For differentiable input, the branch&bound method is used, unless the problem is convex, in which case the outer-approximation method is used. In the non-differentiable case, differential evolution is applied.

Examples

The continuous function f defined by

$$f(x) = \min\left\{\sqrt{|x+4|}-1, \sqrt{|x+1|}-1.005, \sqrt{|x-3|}+0.5\right\},$$

has a unique global minimum in $[-5,5]$ at $x=-1$, with $f(-1)=1.005$.

```
> f:=min(sqrt(abs(x+4))-1,sqrt(abs(x+1))-1.005,sqrt(abs(x-3))+0.5):;
  nlpsolve(f,x=-5..5)
```

$$[-1.0049992998, [x = -1.0]]$$

Minimize $z = x_1x_4(x_1+x_2+x_3)+x_3$ subject to

$$x_1x_2x_3x_4 \geq 25,$$
$$x_1^2+x_2^2+x_3^2 = 40,$$

where $1 \leq x_i \leq 5$, $i = 1,2,3,4$.

```
> nlpsolve(x1*x4*(x1+x2+x3)+x3,
    [x1*x2*x3*x4-25>=0,x1^2+x2^2+x3^2+x4^2-40=0],[x1,x2,x3,x4]=1..5)
```

$$[17.0140172892, [x_1 = 1.0, x_2 = 4.74299973362, x_3 = 3.82114985829, x_4 = 1.3794083106]]$$

Minimize $z = (x_1-1)^2+(x_1-x_2)^2+(x_3-1)^2+(x_4-1)^4+(x_5-1)^6$ subject to

$$x_1^2x_4+\sin(x_4-x_5) = 2\sqrt{2},$$
$$x_2+x_3^4x_4^2 = 8+\sqrt{2}.$$

```
> nlpsolve((x1-1)^2+(x1-x2)^2+(x3-1)^2+(x4-1)^4+(x5-1)^6,
    [x1^2*x4+sin(x4-x5)=2*sqrt(2),x2+x3^4*x4^2=8+sqrt(2)])
```

$$[0.241505128809, [x_1 = 1.16617119669, x_2 = 1.18211040318, x_3 = 1.3802572722,$$
$$x_4 = 1.50603586392, x_5 = 0.610913318325]]$$

Maximize $z = 3x_1x_2-x_1+6x_2$ subject to $5x_1x_2-4x_1-4.5x_2 \leq 32$, where $1 \leq x_1, x_2 \leq 5$ are integers.

```
> nlpsolve(3x1*x2-x1+6x2,[5x1*x2-4x1-4.5x2<=32],[x1,x2]=1..5,assume=integer,maximize)
```

$$[58.0, [x_1 = 2.0, x_2 = 5.0]]$$

Maximize $z = 2x_1+3x_2+6x_3-2x_1x_2-x_1x_3-4x_2x_3$ where all variables are binary.

```
> nlpsolve(2x1+3x2+6x3-2x1*x2-x1*x3-4x2*x3,assume=nlp_binary,maximize)
```

$$[7.0, [x_1 = 1.0, x_2 = 0, x_3 = 1.0]]$$

Minimize $z = 5y-2\ln(x+1)$ subject to

$$e^{x/2}-\frac{1}{2}\sqrt{y} \leq 1,$$
$$-2\ln(x+1)-y+2.5 \leq 0,$$
$$x+y \leq 4,$$

where $x \in [0,2]$ and $y \in [1,3]$ is integer.

```
> nlpsolve(5y-2*ln(x+1),[exp(x/2)-sqrt(y)/2-1<=0,-2*ln(x+1)-y+2.5<=0,x+y-4<=0],
    x=0..2,y=1..3,nlp_integervariables=[y])
```

$$[8.54528930252, [x = 1.06959999348, y = 2.0]]$$

Minimize $z = x_1^2 + x_2^2 + 3x_3^2 + 4x_4^2 + 2x_5^2 - 8x_1 - 2x_2 - 3x_3 - x_4 - 2x_5$ subject to

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 &\le 400, \\ x_1 + 2x_2 + 2x_3 + x_4 + 6x_5 &\le 800, \\ 2x_1 + x_2 + 6x_3 &\le 200, \\ x_3 + x_4 + 5x_5 &\le 200, \\ x_1 + x_2 + x_3 + x_4 + x_5 &\ge 55, \\ x_1 + x_2 + x_3 + x_4 &\ge 48, \\ x_2 + x_4 + x_5 &\ge 34, \\ 6x_1 + 7x_5 &\ge 104, \end{aligned}$$

where $0 \le x_i \le 99$ are integers for $i = 1, \ldots, 5$.

```
> obj:=x1^2+x2^2+3x3^2+4x4^2+2x5^2-8x1-2x2-3x3-x4-2x5;
  constr:=[x1+x2+x3+x4+x5<=400,x1+2x2+2x3+x4+6x5<=800,
           2x1+x2+6x3<=200,    x3+x4+5x5<=200,
           x1+x2+x3+x4+x5>=55, x1+x2+x3+x4>=48,
           x2+x4+x5>=34,       6x1+7x5>=104];
  nlpsolve(obj,constr,[x1,x2,x3,x4,x5]=0..99,assume=integer)
```

$$[807.0, [x_1 = 16.0, x_2 = 22.0, x_3 = 5.0, x_4 = 5.0, x_5 = 7.0]]$$

Minimize the function

$$f(x_1, x_2) = \begin{cases} \int_{x_1}^{x_2} \mathrm{e}^{-t} t^a \sin(t + x_1) \cos(2t - x_2) \mathrm{d}t, & x_1 < x_2, \\ 0, & x_1 \ge x_2 \end{cases}$$

for $0 \le x_1, x_2 \le 10$ and $a = 2$.

With the following program compiled in XCAS:

```
f:=proc(x1,x2,a)
  local t;
  purge(t);
  if x1<x2 then
    return approx(int(exp(-t)*t^a*sin(t+x1)*cos(2t-x2),t=x1..x2));
  else return 0; fi;
end:;
```

input:

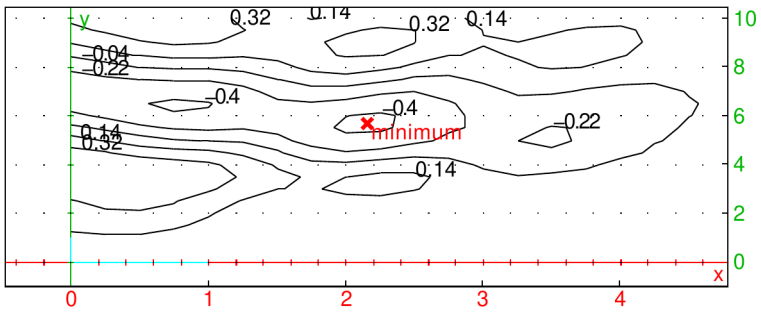
```
> sol:=nlpsolve(quote(f(x1,x2,2)),[x1,x2]=0..10)
```

$$[-0.445124159544, [x_1 = 2.15550857079, x_2 = 5.66106529292]]$$

Since `f` is programmatic, it cannot be differentiated symbolically in XCAS. Differential evolution is thus selected for minimization since the problem is box-constrained and no initial points are provided. Because `f` has three input arguments, the first argument in `nlpsolve` has to be quoted in order to fix the parameter a to 2. This technique should be used for "black box" objectives possibly requiring parameter specifications.

To see a contour plot of f, enter:

```
> contourplot(quote(f(x,y,2)),[x=0..5,y=0..10],linspace(-0.4,0.5,6));
  point(sol[1],display=red+point_width_3+quadrant4,legend="minimum")
```



16.3.5 Simulated annealing minimization

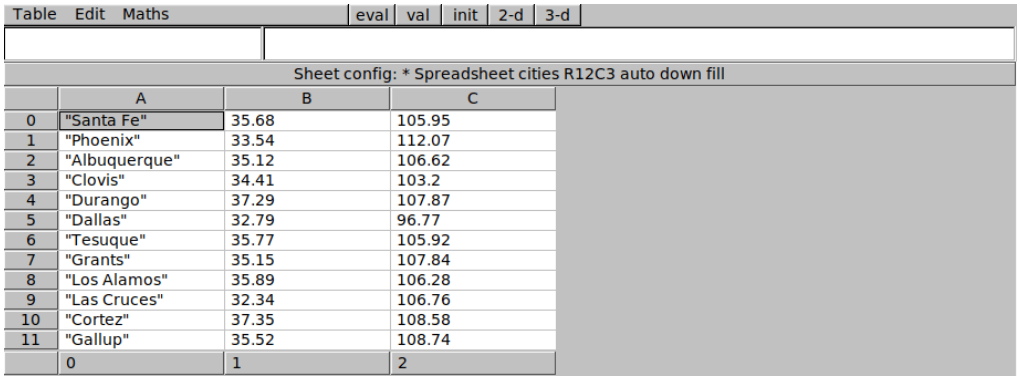
The `simulated_annealing` command attempts to find a point in a search space for which a real-valued cost function is minimized, by using the method of simulated annealing.

`simulated_annealing` requires that `giac` is linked to GSL.

- `simulated_annealing` takes five mandatory arguments and one or two optional arguments:
  - x_0, an initial configuration, which may be any object.
  - *costfunc*, the *cost function* which takes a configuration and returns a real number.
  - *distfunc*, the *distance function* which takes two configurations and returns the distance between them as a real number.
  - *stepfunc*, the *step function* which takes a configuration c and optionally a real number *maxstep* and returns a random configuration c' which is a neighbor of c. If *stepfunc* accepts the *maxstep* argument, it can be used to limit the distance between c and c'.
  - *csparam*, a vector $[k, T_0, \mu, T_{\min}]$ which defines the *cooling schedule* parameters, where k is a Boltzmann constant, T_0 is the initial temperature, μ is the damping factor for temperature and $T_{\min}$ is the minimal temperature.
  - Optionally, *iterparam*, a vector $[n, N]$ of positive integers where n is the number of points tried before stepping and N is the number of iterations for each value of T. By default, $n = 10$ and $N = 100$.
  - Optionally, *maxstep*, a real number which limits the step size from above. If this parameter is set, then *stepfunc* must accept two arguments and *maxstep* will be passed as the second argument whenever calling *stepfunc*. By default, this parameter is unset.
- `simulated_annealing(`x_0, *costfunc*, *distfunc*, *stepfunc*, *csparam* ⟨, *iterparam*, *maxstep* ⟩`)` returns the best configuration found in the context of cost function minimization. The method of simulated annealing gives good results in avoiding local minima, so you usually get the global cost minimum.
- Simulated annealing is essentially a random walk in the search space. The probability of taking a step is determined by the Boltzmann distribution

$$p = \mathrm{e}^{-\frac{E_{i+1}-E_i}{kT}}$$

if $E_{i+1} > E_i$, and $p = 1$ when $E_{i+1} \leq E_i$. Here, E_i and E_{i+1} are the configuration energies (costs). If the new energy is higher, a step is always taken, but it may also be taken if the new energy is lower then E_i (the probability of that event drops with T). The temperature T is initally set to a high value and lowered very slightly after the prescribed number of iterations N by using the formula $T \leftarrow T/\mu$. Here μ is usually slightly larger than 1. The cooling process stops when T reaches its minimum $T_{\min}$.



	A	B	C
0	"Santa Fe"	35.68	105.95
1	"Phoenix"	33.54	112.07
2	"Albuquerque"	35.12	106.62
3	"Clovis"	34.41	103.2
4	"Durango"	37.29	107.87
5	"Dallas"	32.79	96.77
6	"Tesuque"	35.77	105.92
7	"Grants"	35.15	107.84
8	"Los Alamos"	35.89	106.28
9	"Las Cruces"	32.34	106.76
10	"Cortez"	37.35	108.58
11	"Gallup"	35.52	108.74
	0	1	2

Figure 16.1: Cities used in the TSP example

- The functions *costfunc*, *distfunc* and *stepfunc* should return `undef` to raise an error. In that case, `simulated_annealing` will terminate immediately with an error message.
- Simulated annealing is frequently used to solve combinatorial optimization problems.

Examples

(The following examples are adapted from GSL documentation, see here.) In the first example we find the global minimum of a damped sine wave function $f(x) = \mathrm{e}^{-(x-1)^2}\sin(8x)$. This function has many local minima but only a single global minimum in the interval $[1.0, 1.5]$. We start from the point $x_0 = 15.5$, which is several local minima away from the solution. The following parameter values are used: $k = 1$, $T_0 = 0.008$, $\mu = 1.003$, $T_{\min} = 2 \times 10^{-6}$, $n = 20$, $N = 100$ and $maxstep = 1$.

```
> simulated_annealing(15.5,x->exp(-(x-1)^2)*sin(8x),(x,y)->abs(x-y),(x,m)->rand(x-m,x+m),
    [1.0,0.008,1.003,2.0e-6],[20,100],1.0)
```

1.36313090008

In the second example we solve an instance of traveling salesman problem with 12 cities.We enter the cities alongside their latitude and longitude coordinates in an XCAS table cell which we associate with the variable `cities` (see Figure 16.1).

We attempt to find the shortest tour which visits each city exactly once. A tour is represented by a permutation of cities. In total, there are $12! = 479001600$ possible configurations.

First we need to create a distance matrix which stores the distances between each two cities because each distance may be called for multiple times. We do so by applying the function `dist_km` which computes the shortest distance (in kilometres) between two cities on a spherical model of Earth of radius $R = 6375$ km.

```
dist_km:=proc(latitude1,longitude1,latitude2,longitude2)
  local sla1,cla1,slo1,clo1,sla2,cla2,slo2,clo2,x1,x2,y1,y2,z1,z2;
  sla1,cla1:=sin(latitude1*pi/180),cos(latitude1*pi/180);
  slo1,clo1:=sin(longitude1*pi/180),cos(longitude1*pi/180);
  sla2,cla2:=sin(latitude2*pi/180),cos(latitude2*pi/180);
  slo2,clo2:=sin(longitude2*pi/180),cos(longitude2*pi/180);
  x1,y1,z1:=cla1*clo1,cla1*slo1,sla1;
  x2,y2,z2:=cla2*clo2,cla2*slo2,sla2;
  return evalf(6375*acos(x1*x2+y1*y2+z1*z2));
end:;
```

Now we create the distance matrix by using the following commands.

```
> nc:=length(cities):;
  dist_mat:=matrix(nc,nc,(j,k)->dist_km(cities[j,1],cities[j,2],cities[k,1],cities[k,2])):;
```

We generate a neighbor of a given tour by exchanging two random consecutive cities. The distance between two tours (configurations) is simply the Hamming distance. The cost of a tour is equal to its length (in kilometres). The three functions we require are thus entered as follows:

```
costf:=proc(xp)
  local j,d;
  assume(j,symbol);
  d:=0;
  for j from 0 to nc-1 do
    d+=dist_mat[xp[j],xp[irem(j+1,nc)]];
  od;
  return d;
end:;

stepf:=proc(xp)
  local j,jnext,tmp,xpmod;
  j:=rand(nc);
  jnext:=irem(j+1,nc);
  tmp:=xp[j];
  xpmod:=xp;
  xpmod[j]:=xpmod[jnext];
  xpmod[jnext]:=tmp;
  return xpmod;
end:;

distf:=proc(xp,yp)
  return hamdist(xp,yp);
end:;
```

The initial tour is a random permutation of cities.

```
> x0:=randperm(nc):;
```

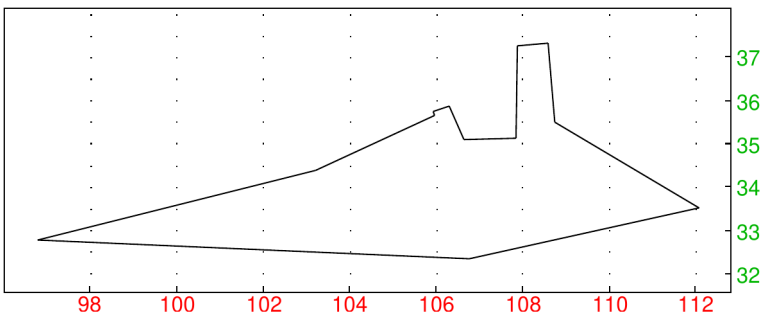
A good tour can be found by the following command (we use the default values of n and N).

```
> best:=simulated_annealing(x0,costf,distf,stepf,[1.0,5000.0,1.005,5.0e-1])
```

$$[7, 2, 8, 6, 0, 3, 5, 9, 1, 11, 10, 4]$$

Finally, we visualize the tour. We swap latitudes and longitudes for a more appropriate layout.

```
> polygon(sortperm(colswap(subMat(cities,0,1,11,2),0,1),best))
```



17 Interpolation and fitting

17.1 Interpolation

17.1.1 Lagrange polynomial

The `lagrange` command finds the Lagrange polynomial which interpolates given data.

- `lagrange` takes two mandatory arguments and one optional argument:
  - l_1 and l_2, two lists of the same size. These can be given as a matrix with two rows.

    The first list (resp. row) corresponds to the abscissa values x_k $(k = 1, \ldots, n)$, and the second list (resp. row) corresponds to ordinate values y_k $(k = 1, \ldots, n)$.
  - Optionally x, the name of a variable (by default `x`).
- `lagrange`$(l_1, l_2\,\langle, x\rangle)$ returns a polynomial expression $P(x)$ of degree $n-1$ such that $P(x_i) = y_i$.

You can use the `interp` command as a synonym for `lagrange`.

Examples

```
> lagrange([[1,3],[0,1]])
```

or:

```
> lagrange([1,3],[0,1])
```

$$\frac{x-1}{2}$$

since $\frac{x-1}{2} = 0$ for $x = 1$ and $\frac{x-1}{2} = 1$ for $x = 3$.

```
> factor(lagrange([1,2,3,4],[1,a,-2a,0],t))
```

$$\frac{(t-4)\,(9at^2 - 30at + 21a - t^2 + 5t - 6)}{6}$$

Remark. An attempted function definition such as `f:=lagrange([1,2],[3,4],y)` does not return a function but an expression with respect to y. To define f as a function, input:

```
> f:=unapply(lagrange([1,2],[3,4],x),x)
```

Avoid `f(x):=lagrange([1,2],[3,4],x)` since then the Lagrange polynomial would be computed each time `f` is called (indeed in a function definition, the second member of the assignment is not evaluated). Note also that `g(x):=lagrange([1,2],[3,4])` would not work since the default argument of `lagrange` would be global, hence not the same as the local variable used for the definition of `g`.

17.1.2 Spline interpolation

Spline interpolation is a method of piecewise polynomial interpolation in which smaller, low-degree pieces are glued together at given points under suitable boundary conditions. With Xcas you can do both symbolic and numerical spline interpolation.

Theoretical background. Let σ_n be a subdivision of a real interval $[a, b]$:

$$a = x_0, x_1, \ldots, x_n = b.$$

The function s is a *spline* function of degree l if s is a function from $[a, b]$ to $\mathbb{R}$ such that:

- s has continuous derivatives up to the order $l - 1$,
- on each interval of the subdivision σ_n, s is a polynomial of degree less or equal than l.

Theorem 5. *The set of spline functions of degree l on σ_n is an $\mathbb{R}$-vector space of dimension $n + l$.*

Proof. Let s be a spline function of degree l on σ_n. For $x \in [a, x_1]$, s is a polynomial A of degree less or equal to l, hence on $[a, x_1]$, $s = A(x) = a_0 + a_1x + \cdots + a_lx^l$ and A is a linear combination of $1, x, \ldots, x^l$.

For $x \in [x_1, x_2]$, s is a polynomial B of degree less or equal to l, hence on $[x_1, x_2]$, $s = B(x) = b_0 + b_1x + \cdots + b_lx^l$. Since s has continuous derivatives up to order $l-1$, it follows $B^{(j)}(x_1) - A^{(j)}(x_1) = 0$ for $0 \leq j < l$, and therefore $B(x) - A(x) = \alpha_1(x - x_1)^l$, i.e. $B(x) = A(x) + \alpha_1(x - x_1)^l$, for some α_1. Define the function:

$$q_1(x) = \begin{cases} 0, & a \leq x \leq x_1, \\ (x - x_1)^l, & x_1 \leq x \leq b. \end{cases}$$

Now $s|_{[a,x_2]} = a_0 + a_1x + \cdots + a_lx^l + \alpha_1 q_1(x)$.

For $x \in [x_2, x_3]$, s is a polynomial C of degree less or equal than l, hence on $[x_2, x_3]$, $s = C(x) = c_0 + c_1x + \cdots + c_lx^l$. Since s has continuous derivatives up to order $l-1$, it follows $C^{(j)}(x_2) - B^{(j)}(x_2) = 0$ for $0 \leq j < l$, and therefore $C(x) - B(x) = \alpha_2(x - x_2)^n$ or $C(x) = B(x) + \alpha_2(x - x_2)^n$ for some $\alpha_2 \in \mathbb{R}$. Define the function:

$$q_2(x) = \begin{cases} 0, & a \leq x \leq x_2, \\ (x - x_2)^l, & x_2 \leq x \leq b. \end{cases}$$

Now $s|_{[a,x_3]} = a_0 + a_1x + \cdots + a_lx^l + \alpha_1 q_1(x) + \alpha_2 q_2(x)$.

Continuing, define the functions q_j for all $1 \leq j < n$:

$$q_j(x) = \begin{cases} 0, & a \leq x \leq x_j, \\ (x - x_j)^l, & x_j \leq x \leq b. \end{cases}$$

You obtain $s|_{[a,b]} = a_0 + a_1x + \cdots + a_lx^l + \alpha_1 q_1(x) + \cdots + \alpha_{n-1} q_{n-1}(x)$, meaning that s is a linear combination of $n+l$ independent functions $1, x, \ldots, x^l, q_1, \ldots, q_{n-1}$. It follows that the set of all possible s is a real vector space of dimension $n + l$. □

Types of spline functions. If you want to interpolate a function f on σ_n by a spline function s of degree l, then s must satisfy $s(x_k) = y_k = f(x_k)$ for all $0 \leq k \leq n$. This gives $n + 1$ conditions, leaving $l - 1$ degrees of freedom. You can therefore add $l - 1$ conditions, these conditions are on the derivatives of s at a and b.

Hermite interpolation, natural interpolation and periodic interpolation are three kinds of interpolation obtained by specifying three kinds of constraints. The uniqueness of the solution of the interpolation problem can be proved for each kind of constraints.

If l is odd (i.e. $l = 2m - 1$), there are $2m - 2$ degrees of freedom. The constraints are defined as follows.

- Hermite interpolation: $s^{(j)}(a) = f^{(j)}(a)$ and $s^{(j)}(b) = f^{(j)}(b)$, $1 \leq j \leq m - 1$.
- Natural interpolation: $s^{(j)}(a) = s^{(j)}(b) = 0$, $m \leq j \leq 2m - 2$.
- Periodic interpolation: $s^{(j)}(a) = s^{(j)}(b)$, $1 \leq j \leq 2m - 2$.

If l is even ($l = 2m$), there are $2m - 1$ degrees of freedom. The constraints are defined as follows.

- Hermite interpolation: $s^{(j)}(a) = f^{(j)}(a)$, $s^{(j)}(b) = f^{(j)}(b)$ and $s^{(m)}(a) = f^{(m)}(a)$, $1 \le j \le m - 1$.
- Natural interpolation: $s^{(j)}(a) = s^{(j)}(b) = 0$ and $s^{(2m-1)}(a) = 0$, $m \le j \le 2m - 2$.
- Periodic interpolation: $s^{(j)}(a) = s^{(j)}(b)$, $1 \le j \le 2m - 1$.

Natural symbolic interpolation. The `spline` command finds the natural spline.

- `spline` takes two mandatory arguments and up to three optional arguments:
  - L_x, a list of abscissas (in increasing order).
  - L_y, a list of ordinates (the same length as L_x).
  - Optionally, x, either a symbolic variable, real number, or list of reals (by default `x`).
  - Optionally, l, a positive integer for the degree (by default, $l = 3$).
  - Optionally, `piecewise`, the symbol.
- If x is a variable, then `spline(`$L_x, L_y \langle, x, l \rangle \langle,$`piecewise`$\rangle$`)` returns the natural spline function s of degree l, where $s(L_{x,j}) = L_{y,j}$ for $j = 0, \ldots,$ `length(`L_x`)`, as a list of polynomials, each polynomial being valid on the corresponding interval determined by L_x, or, if `piecewise` is given, a piecewise expression representing the entire spline for $\min L_x \le x \le \max L_x$.
- If x is a real number (resp. a list of numbers), then `spline(`$L_x, L_y, x \langle, l \rangle$`)` returns the spline value at x (resp. the list of spline values at individual components of x).

Examples

Find the natural spline of degree 3, crossing through the points $(0, 1)$, $(1, 3)$ and $(2, 0)$:

```
> spline([0,1,2],[1,3,0])
```

or:

```
> spline([0,1,2],[1,3,0],x,3)
```

$$\left[-\frac{5}{4}x^3 + \frac{13}{4}x + 1, \frac{5}{4}(x-1)^3 - \frac{15}{4}(x-1)^2 - \frac{x-1}{2} + 3\right]$$

The first polynomial, $-\frac{5}{4}x^3 + \frac{13}{4}x + 1$, is defined on the interval $[0, 1]$ (the first interval defined by the list $[0, 1, 2]$) and the second polynomial $\frac{5}{4}(x-1)^3 - \frac{15}{4}(x-1)^2 - \frac{x-1}{2} + 3$ is defined on the interval $[1, 2]$, the second interval defined by the list $[0, 1, 2]$. To obtain the result in a piecewise form, enter:

```
> spline([0,1,2],[1,3,0],x,3,piecewise)
```

$$\begin{cases} -\frac{5}{4}x^3 + \frac{13}{4}x + 1, & x \ge 0 \wedge 1 > x \\ \frac{5}{4}(x-1)^3 - \frac{15}{4}(x-1)^2 - \frac{x-1}{2} + 3, & 2 \ge x \end{cases}$$

Find the natural spline of degree 4, crossing through the points $(0, 1)$, $(1, 3)$, $(2, 0)$ and $(3, -1)$:

```
> spline([0,1,2,3],[1,3,0,-1],x,4)
```

$$\left[-\frac{62}{121}x^4 + \frac{304}{121}x + 1,\right.$$
$$\frac{201}{121}(x-1)^4 - \frac{248}{121}(x-1)^3 - \frac{372}{121}(x-1)^2 + \frac{56}{121}(x-1) + 3,$$
$$\left.-\frac{139}{121}(x-2)^4 + \frac{556}{121}(x-2)^3 + \frac{90}{121}(x-2)^2 - \frac{628}{121}(x-2)\right]$$

The output is a list of three polynomial functions of x, defined on the intervals $[0,1]$, $[1,2]$ and $[2,3]$, respectively. To compute the spline value for points $x = \frac{1}{2}, \frac{4}{3}, \frac{9}{4}$, enter:

```
> spline([0,1,2,3],[1,3,0,-1],[1/2,4/3,9/4],4)
```

$$\left[\frac{2153}{968}, \frac{9008}{3267}, -\frac{36667}{30976}\right]$$

Find the natural spline interpolation of the cosine function with abscissas $\left[0, \frac{\pi}{2}, \frac{3\pi}{2}\right]$:

```
> t:=[0,pi/2,3*pi/2]:; spline(t,cos(t))
```

$$\left[\frac{4x^3}{3\pi^3} - \frac{7x}{3\pi} + 1, -\frac{2\left(x-\frac{\pi}{2}\right)^3}{3\pi^3} + \frac{2\left(x-\frac{\pi}{2}\right)^2}{\pi^2} - \frac{4\left(x-\frac{\pi}{2}\right)}{3\pi}\right]$$

Numerical spline interpolation. The `interp` command interpolates a discretized function at a list of given points by using fast spline interpolation routines provided by GSL. You should use this command when you do not need the interpolated function itself but only its values at a sequence of points.

- `interp` takes two or three mandatory arguments and a sequence of optional arguments:
  - *data*, a sequence or list of two vectors x and y of the same length.
  - v, a vector of x-values at which the spline is evaluated.
  - Optionally, *opts*, a sequence of options each of which is one of:
    - "*stype*", a string specifying spline type. Allowed values for *stype* are `cubic`, `akima` and `steffen` (by default, *stype* = `cubic`). See below for descriptions of these types.
    - `periodic`, the symbol specifying that interpolation should be periodic (by default, interpolation is not periodic). This argument gets ignored if *stype* = `steffen`.
    - d, a nonnegative integer specifying the order of the derivative of interpolated function to be evaluated. Allowed values are 0, 1 and 2 (by default, $d = 0$).
- `interp([`x, y`],` v `⟨,` *opts*`⟩)` or `interp(`x, y, v `⟨,` *opts*`⟩)` returns the list z of spline (derivative) values at points in v.
- Note that *data* and v must be entirely numeric and *data* should also be inexact.
- The elements of x must be strictly ascending. The minimal number of data points is 3.
- The elements of t must be between $\min x$ and $\max x$, otherwise the output list will contain undefined elements (as an effect, plotting would fail).

The description of spline types below is quoted from GSL documentation.

cubic Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The second derivative is chosen to be zero at the first point and last point.

periodic cubic Cubic spline with periodic boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The derivatives at the first and last points are also matched. Note that the last point in the data must have the same y-value as the first point, otherwise the resulting periodic interpolation will have a discontinuity at the boundary.

(periodic) akima Non-rounded Akima spline with natural (periodic) boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

steffen Steffen's method guarantees the monotonicity of the interpolating function between the given data points. Therefore, minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The interpolated function is piecewise cubic in each interval. The resulting curve and its first derivative are guaranteed to be continuous, but the second derivative may be discontinuous.

Note that Steffen interpolation is available only if `giac` is compiled with GSL version 2.2 or later.

Example

This example is adapted from GSL documentation. To define some synthetic data, enter:

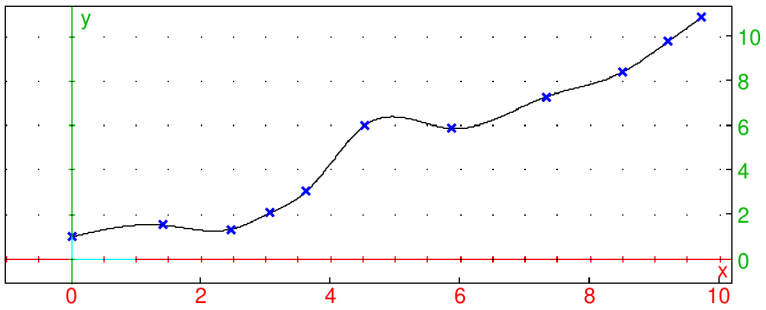
```
> x,y:=makelist(k->k+0.5*sin(k),0,10),makelist(k->k+cos(k^2),0,10):;
```

This gives you 11 data points. To interpolate the function at intermediate points, enter:

```
> t:=linspace(min(x),max(x),1000):; z:=interp(x,y,t):;
```

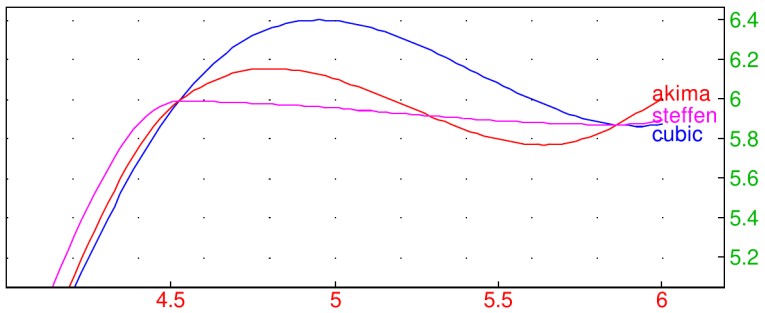
Now plot the data and the interpolated values together:

```
> listplot(tran([t,z]));
  scatterplot(tran([x,y]),display=blue+point_width_2)
```



To see the difference between the supported spline types, interpolate in the segment $[4, 6]$. Enter:

```
> t:=linspace(4,6):;
  c,a,s:=interp(x,y,t,"cubic"),interp(x,y,t,"akima"),interp(x,y,t,"steffen"):;
  listplot(tran([t,c]),color=blue+quadrant4,legend="cubic");
  listplot(tran([t,a]),color=red,legend="akima");
  listplot(tran([t,s]),color=magenta,legend="steffen");
```



17.1.3 Rational interpolation

Thiele interpolation. The `thiele` command finds the rational interpolation given a tabulated function.

- `thiele` takes two or three arguments:
  - *data*, a matrix with two columns. The first column contains the x coordinates and the second column contains the corresponding y coordinates.

Instead of a single matrix, data can be given as sequence containing the vector of x coordinates and the vector of y coordinates (in which case the call to `thiele` has three arguments).

– v, an identifier, number or symbolic expression (default: x).

- `thiele(`*data*, v`)` returns $R(v)$ where R is the Thiele rational interpolant.
- Note that the obtained interpolant may have singularities in the range $[\min \mathbf{x}, \max \mathbf{x}]$.

Examples

```
> thiele([[1,3],[2,4],[4,5],[5,8]],x)
```

Warning: interpolant has singularities in [4,5]

$$\frac{19x^2 - 45x - 154}{18x - 78}$$

```
> assume(a>1):; f:=unapply(thiele([1,2,a],[3,4,5],t),t)
```

$$t \mapsto \frac{5at - 2a - 13t + 10}{at - 3t + 2}$$

Indeed:

```
> simplify(apply(f,[1,2,a]))
```

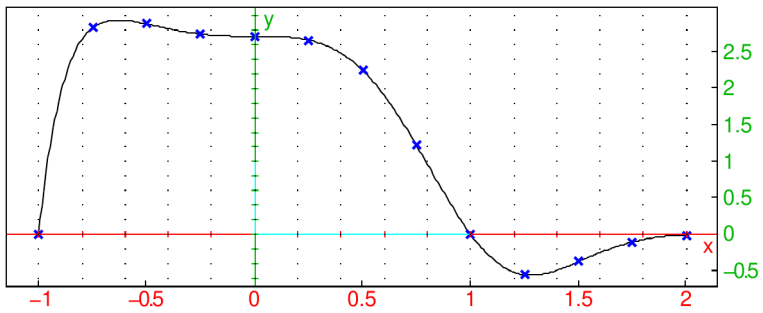
$$[3, 4, 5]$$

In the following example, data is obtained by sampling the function $f(x) = (1 - x^4)\,\mathrm{e}^{1-x^3}$ on $[-1, 2]$.

```
> xdata:=linspace(-1.0,2.0,13):;
  ydata:=apply(x->(1-x^4)*exp(1-x^3),xdata):;
  p:=thiele(xdata,ydata,x)
```

$$\begin{aligned}&(-1.55286115659x^6 + 5.87298387514x^5 - 5.4439152812x^4 + 1.68655817708x^3\\ &-2.40784868317x^2 - 7.55954205222x + 9.40462512097)/\\ &(x^6 - 1.24295718965x^5 - 1.33526268624x^4 + 4.03629272425x^3\\ &-0.885419321x^2 - 2.77913222418x + 3.45976823393)\end{aligned}$$

```
> plot(t,x=-1..2);
  scatterplot(xdata,ydata,color=blue+point_width_2)
```



Rational interpolation without poles. The `ratinterp` command computes a family of pole-free rational functions which interpolate given data. Rational interpolation usually gives better results than the classic polynomial interpolation, which may oscillate highly in some cases.

- `ratinterp` takes one mandatory argument and one or two optional arguments:
  - Matrix *data* with 2 columns with rows corresponding to points (x_k, y_k), $k = 0, 1, \ldots, n$, or the sequence of lists $data\_x = [x_0, x_1, \ldots, x_n]$ and $data\_y = [y_0, y_1, \ldots, y_n]$, where $a = x_0 < x_1 < \cdots < x_n = b$,
  - an identifier x, which may also be a number, symbolic expression or list of numbers a (by default `x`),
  - Optionally, d, an integer such that $0 \le d \le n$ (by default $\lfloor (n+1)/2 \rfloor$).
- `ratinterp(`*data*$\langle$, *var*, *d*$\rangle$`)` or `ratinterp(`*data_x*, *data_y*$\langle$, *var*, *d*$\rangle$`)` returns a rational interpolation $r(a)$ of the given points using the method of Floater and Hormann (2006). If a is a list of numbers $a_1, a_2, \ldots, a_m$, then the list $[r(a_1) \ldots, r(a_m)]$ is returned. There are at most $n+1$ distinct interpolants which can be specified by varying the parameter d, which allows you to choose the most suitable one.

Examples

```
> ratinterp([1,3,5,8],[2,-1,3,4])
```

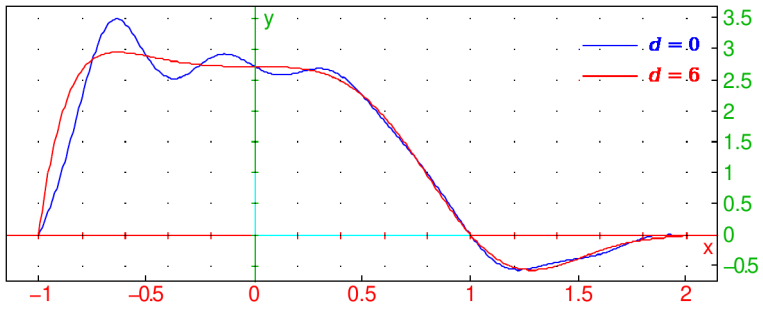
$$-\frac{29x^3}{168} + \frac{17x^2}{7} - \frac{1507x}{168} + \frac{61}{7}$$

```
> ratinterp([1,3,5,8],[2,-1,3,4],x,2)
```

$$-\frac{29x^3}{168} + \frac{17x^2}{7} - \frac{1507x}{168} + \frac{61}{7}$$

With `xdata` and `ydata` as in the third example of `thiele` above:

```
> plot(ratinterp(xdata,ydata,x,0),x=-1..2,color=blue);
  plot(ratinterp(xdata,ydata,x,6),x=-1..2,color=red);
  legend(1.5+3i,"$_____  @d = 0$",color=blue);
  legend(1.5+2.5i,"$_____  @d = 6$",color=red);
```



17.1.4 Trigonometric interpolation

The `triginterp` command computes a trigonometric polynomial which interpolates given data.

- `triginterp` takes four arguments:
  - L, a list of numbers.
  - a, a number (the beginning of an interval).

– b, a number (the end of the interval).

– x, the name of a variable.

The last three arguments can also be given as x=a..b.

- `triginterp`(L, a, b, x) or `triginterp`$(L, x$=$a..b)$ returns the trigonometric polynomial that interpolates data given in the list L. It is assumed that the list L contains ordinate components of the points with equidistant abscissa components between a and b such that the first element of L corresponds to a and the last element to b.

L may be a list of experimental measurements of some quantity taken at regular intervals, with the first observation at time $t = a$ and the last observation at time $t = b$. The resulting trigonometric polynomial has period $T = \frac{n}{n-1}(b-a)$, where $n =$ `size`(L) is the number of observations.

As an example, assume that the following data is obtained by measuring the temperature every three hours during one day:

Hour of the day	0	3	6	9	12	15	18	21
Temperature (°C)	11	10	17	24	32	26	23	19

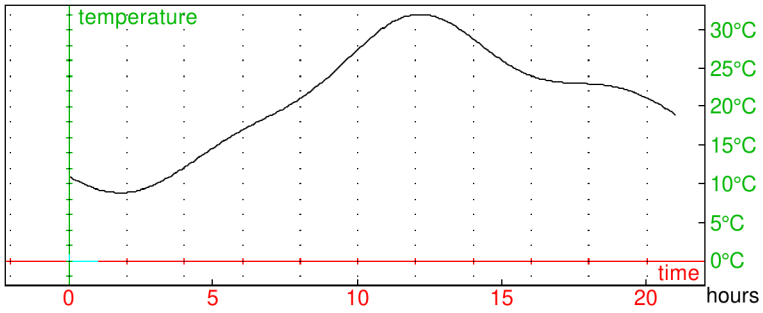
Suppose that an estimate of the temperature at 13:45 is required. To obtain a trigonometric interpolation of the data:

```
> tp:=triginterp([11,10,17,24,32,26,23,19],x=0..21)
```

$$\frac{81}{4}+\frac{1}{8}\left(-21\sqrt{2}-42\right)\cos\left(\frac{1}{12}\pi x\right)+\frac{1}{8}\left(-11\sqrt{2}-12\right)\sin\left(\frac{1}{12}\pi x\right)+\frac{3}{4}\cos\left(\frac{1}{6}\pi x\right)-$$
$$\frac{7}{4}\sin\left(\frac{1}{6}\pi x\right)+\frac{1}{8}\left(21\sqrt{2}-42\right)\cos\left(\frac{1}{4}\pi x\right)+\frac{1}{8}\left(-11\sqrt{2}+12\right)\sin\left(\frac{1}{4}\pi x\right)+\frac{\cos\left(\frac{1}{3}\pi x\right)}{2}$$

To plot the interpolant, enter:

```
> labels=["time","temperature"]; legend=["hours","°C"]; plot(tp,x=0..21)
```



The interpolant is smooth, without oscillations, and appears realistic in the given context. Now a temperature at 13:45 can be approximated by the value of `tp` for $x = 13.75$:

```
> tp | x=13.75
```

$$29.4863181684$$

17.2 Curve fitting

17.2.1 Least-squares polynomial approximation

The `fitpoly` command is used for replacing tabular data or a function by a polynomial.

Smoothing data with polynomial regression. `fitpoly` can be used for finding a polynomial that fits a given tabular data.

- `fitpoly` takes one mandatory argument and two optional arguments:
  - *data*, a two-column matrix or a sequence of two lists, representing the x- and y-values of data points $(x_0, y_0), (x_1, y_1), \ldots, (x_m, y_m)$.
  - Optionally, x`[=`a`..`b`]` or $a..b$, which is a variable x and optionally its domain $[a, b]$ or a segment $[a, b]$ (by default, unset).
  - Optionally, a sequence of options, each of which may be one of:
    - `limit=`N, a nonnegative integer, which is the maximal degree for the resulting polynomial (by default, $N = 15$).
    - `degree=`d, where d is a nonnegative integer, which is the desired degree of the resulting polynomial (by default, unset).
    - `threshold=`*tol*, a positive real number, which is used for finding an appropriate fitting degree, as described below (by default, `threshold=`0.01, ignored when d is set).
    - `length=`l, a positive integer, which is used for finding an appropriate fitting degree, as described below (by default, `length=`5, ignored when d is set).
- If d is set, then `fitpoly` returns the polynomial of degree $\min\{d, N\}$ which best fits the data in the least-squares sense.
- If d is not set, then a polynomial p_n of modest degree $n \le N$ but a good error suppression (thus representing the trend of the data) is chosen using *tol* and l such that raising the degree does not make a significant improvement to the approximation. Precisely, n is the first nonnegative integer such that
$$\frac{\text{stddev}[\sigma_n^2, \sigma_{n+1}^2, \ldots, \sigma_{n+l-1}^2]}{\text{mean}[\sigma_0^2, \sigma_1^2, \ldots, \sigma_{n+l-1}^2]} \le tol,$$
where σ_k^2 is the sum of squared residuals $\sum_{i=0}^{m}(y_i - p_k(x_i))^2$ and p_k is the best-fitting polynomial of degree k. Ideally, n is the smallest degree for which $\sigma_n^2 \approx \sigma_{n+1}^2 \approx \sigma_{n+2}^2 \approx \cdots$, meaning that higher-order polynomials would overfit on the noise.
- Lowering the parameter `length`, as well as increasing `threshold`, would make the algorithm more greedy, effectively lowering the degree of the output polynomial. Note that when `degree` is set, these two parameters are ignored.
- If no variable is specified, then `fitpoly` returns the list of polynomial coefficients.
- If a segment $[a, b]$ is given, then only the data points (x_k, y_k) for which $a \le x_k \le b$ are considered.

Example

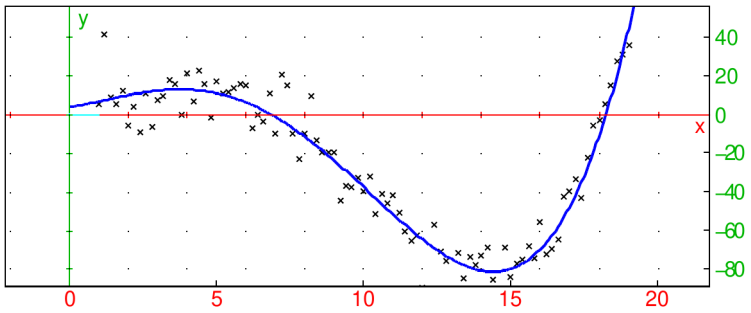
We find a polynomial which best approximates data infected by noise. Data is obtained by evaluating the polynomial $f(x) = \frac{1}{100}x^4 - \frac{1}{5}x^3 + 6x$ at 100 points between $x = 1$ and $x = 20.8$. The noise is generated by a random, normally-distributed variable.

```
> f(x):=x^4/100-x^3/5+6x:;
  X:=linspace(1.0,20.8):;
  Y:=apply(f,x)+randvector(100,randvar(normal,mean=0,stddev=10)):;
  p:=fitpoly(X,Y)
```

$$[0.011665, -0.266955, 0.844543, 2.44967, 3.94846]$$

We obtain the list of polynomial coefficients, starting with the leading coefficient. Note that the polynomial of degree 4 is returned (it has five coefficients), which is, in this case, obviously optimal for data smoothing. To visualize the fit, enter:

```
> scatterplot(X,Y); plot(r2e(p,x),x=0..20,color=blue+line_width_2)
```



When approximating only the data for $1 \leq x \leq 10$, you obtain a polynomial of 9th degree. This is called *overfitting*. Namely, data curvature at the restriction is small, which makes the noise more prominent.

```
> p1:=fitpoly(X,Y,x=1..10):; degree(p1)
```

9

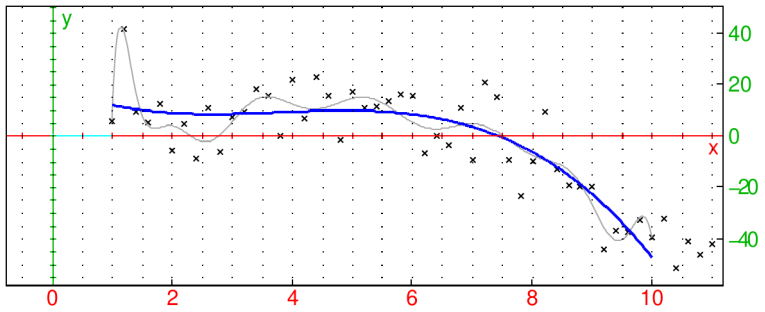
To make the approximating polynomial less sensitive to noise, you can increase the threshold value *tol*.

```
> p2:=fitpoly(X,Y,x=1..10,threshold=0.05)
```

$$-0.256298728063x^3 + 2.89166276754x^2 - 9.93723278591x + 19.4201078899$$

(Alternatively, you could set the parameter `length` to a smaller value, e.g. 3.) The command lines below show the curves `p1` and `p2` against data points (the overfitted one is drawn in grey).

```
> scatterplot(X,Y);
  plot(p1,x=1..10,color=grey);
  plot(p2,x=1..10,color=blue+line_width_2)
```



Approximating functions by polynomials. `fitpoly` can be used for approximating a continuous function $f : [a, b] \to \mathbb{R}$ by a polynomial p_n of certain degree n which minimizes the error in the sense of ℓ^2 norm.

- `fitpoly` takes two mandatory arguments and one optional argument:
  - f, an univariate symbolic expression representing the function to be approximated.
  - An interval $a..b$, which is the domain $[a, b]$ of f.
  - Optionally, *opts*, a sequence of options each of which may be one of:
    - `limit=`N, as before.

  * `degree=`d, as before.
  * ε or `threshold=`ε, which is a positive real number such that for the resulting polynomial p_n the following holds:

$$\|f - p_n\|^2 = \int_a^b (f(x) - p_n(x))^2 \mathrm{d}x \leq \varepsilon \tag{17.1}$$

    (by default, $\varepsilon = 10^{-8}$). This parameter is ignored when d is set.

- `fitpoly(`$f, a..b\langle, opts\rangle$`)` finds p_n with smallest $n \leq N$ such that (17.1) holds. If such n does not exist, then p_N is returned.
- If the parameter d is set, then p_d which minimizes $\|f - p_d\|$ among all polynomials of degree d is returned.
- Polynomial approximation is fast and robust even for large d and N resp. for small ε.

Examples

```
> fitpoly(cos(x),0..pi/2,1e-2)
```

$$-0.244320054366x^3 - 0.336364730985x^2 + 1.13004492821x - 0.0107731059645$$

```
> f:=exp(-7x)*5x:;
  g:=fitpoly(f,0..1,degree=8)
```

$$\begin{aligned} &- 21.717636069x^8 + 107.930784832x^7 - 232.831655404x^6 \\ &+ 286.778708741x^5 - 222.236631985x^4 + 111.004732684x^3 \\ &- 33.8769709361x^2 + 4.95239715728x + 0.000500886757642 \end{aligned}$$

The mean absolute error of the above approximation can be estimated as follows:

```
> sqrt(romberg((f-g)^2,x=0..1))
```

$$9.3456615562 \times 10^{-5}$$

17.2.2 Rational minimax approximation

The `minimax` command finds the rational function which approximates a continuous function $f : [a, b] \to \mathbb{R}$ most closely in the sense of ℓ^∞ norm. It operates exclusively in floating-point arithmetic.

- `minimax` takes three mandatory argument and a sequence of optional arguments:
  - *expr*, a univariate expression representing the function f.
  - x`=`$a..b$, a variable with bounds.
  - *deg*, which is either a positive integer n or a list $[n, m]$ where n is a negative and m a positive integer, specifying the degrees of the numerator (and denominator) of the approximant (by default, $m = 0$).
  - Optionally, *opts*, a sequence of options each of which is one of:
    * `limit=`*maxiter*, where *maxiter* is a positive integer specifying the number of iterations of the Remez algorithm (by default, *maxiter* $= 100$).
    * `rand`$\langle$`=`$N\rangle$, where N is a positive integer specifying the number of tries with initial node skewing (by default, $N = 1$ and no skewing is performed).

  * `threshold=`*tol*, where *tol* is a real number greater than 1 specifying the terminating criterion $\max_{x\in[a,b]} e(x) < tol \cdot \min_{x\in[a,b]} e(x)$ for the Remez algorithm, where $e(x) = |f(x) - r(x)|$ is the error function in the current iteration (by default, $tol = 1.02$).

- `minimax(`*expr*, *x*`=`*a*`..`*b*, *deg*⟨, `limit=`*maxiter*⟩`)` uses the Remez method (see e.g. here) to find the rational function $r(x) = \frac{p(x)}{q(x)}$, where p is a polynomial of degree n and q a polynomial of degree m normalized such that its trailing coefficient is equal to 1, which minimizes the ℓ^∞ error when approximating *expr* on $[a, b]$. If r has no singularities in $[a, b]$, then a list containing $\|f - r\|_\infty$ and $r(x)$ is returned. Otherwise, an approximation of $\frac{1}{b-a}\int_a^b |f(x) - r(x)|\mathrm{d}x$, computed by using Simpson's rule with $20(n + m + 2)$ steps, is returned as the first element (this may serve as a rough estimation of fitness outside poles) and a warning is printed in the message area.

- By specifying only the parameter n, you get an approximation of f by a polynomial. This is faster and the result is always well-defined, but rational approximants provide more flexibility and can be significantly more accurate even for smaller values of n and m (these should be chosen so that r has no singularities in $[a, b]$, however).

- With the `rand`⟨`=`*N*⟩ option, `minimax` skews the initial (Chebyshev) nodes randomly before proceeding to the first Remez step. This may help the algorithm to converge to a well-defined result when $m > 0$. If N is specified, then the entire computation is performed N times and the best result is selected. This will be a pole-free function when possible. The `rand` option is usually not needed when $m = 0$, i.e. when the minimax polynomial is sought.

Example

Find a function of form $r(x) = \frac{a_2x^2+a_1x+a_0}{b_2x^2+b_1x+b_0}$ which is an optimal ℓ^∞-approximation of $f(x) = \frac{\ln(x+1)}{2+\sin x}$ on the segment $[0, 7]$. Define f by entering:

```
> f:=ln(x+1)/(2+sin(x)):;
```

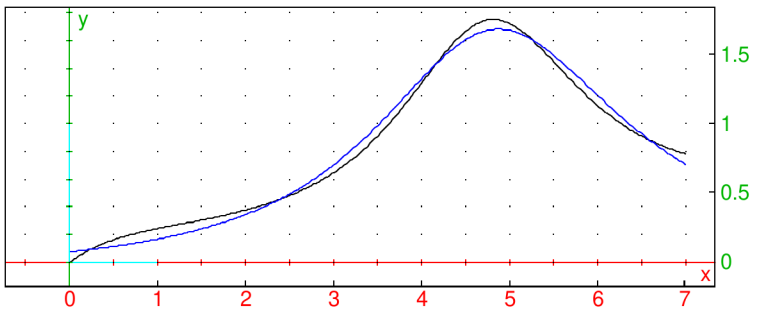
Now compute the approximation by entering:

```
> fit:=minimax(f,x=0..7,[2,2])
```

$$\left[0.0759657940359,\ \frac{-0.00264534091503x^2 + 0.0393947055011x + 0.0759192001346}{0.0386492470797x^2 - 0.368513748981x + 1}\right]$$

To display the approximation, enter:

```
> tol,r:=op(fit):;
  plot(ln(x+1)/(2+sin(x)),x=0..7);
  plot(r,x=0..7,color=blue)
```



(Here, $r(x)$ is drawn in blue.) You can conclude that $\max_{0\le x\le 7} |f(x) - r(x)| =$ `tol`. Indeed:

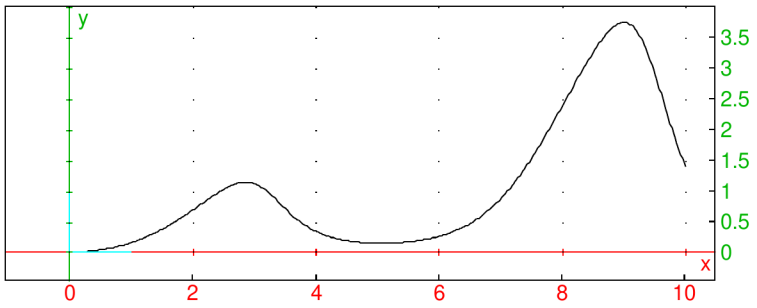
```
> maximize(abs(f-r),x=0..7)
```

$$0.0759657940359$$

The maximal absolute error is attained at $x = 3.45532968966$.

As another example, consider the function $f(x) = \frac{xe^{\sin x}}{10+7\cos x}$ for $x \in [0, 10]$. To show the graph, enter:

```
> f:=x*exp(sin(x))/(10+7*cos(x)); D:=0..10:;
  plot(f,x=D)
```



In the examples that follow, approximation functions are not output to save space.

To obtain the ℓ^∞ error of the 15th order polynomial approximation, enter:

```
> minimax(f,x=D,15)[0]
```

0.0800812074329

By requesting a rational approximant, you can achieve significantly better accuracy with polynomials of lower degrees:

```
> p:=minimax(f,x=D,[8,8])[0]
```

0.000495628620328

Sometimes you get poles in the approximant, for example:

```
> minimax(f,x=D,[10,3])[0]
```

Warning: the result is undefined at point(s) 3.03555290496

0.206238829817

By using the `rand` option, you obtain a pole-free result:

```
> p:=minimax(f,x=D,[10,3],rand=30)[0]
```

Evaluation time: 6.11

0.0364254964552

17.2.3 B-splines

Finding B-splines from control points. The command `bspline` finds a B-spline with a given list of control points and, optionally, a list of breakpoints.

- `bspline` takes one mandatory argument and up to three optional argument(s):
  - *cpts*, a list of m control points in $\mathbb{R}^d$ which may be given either as vectors of (Cartesian) coordinates, 2D or 3D points (graphic objects), or (complex) affixes of 2D points.
  - Optionally, *var*, a variable specification which can be one of:
    - x, an identifier (by default `x`).
    - x=a`..`b, where a and b are real numbers such that $a < b$ (by default, $a = 0$ and $b = 1$).

  * x=[$x_0, x_1, \ldots, x_n$], where $n = m - p + 1$ and x_i are real numbers for $i = 0, 1, \ldots, n$ given in strictly ascending order (by default, $x_k = a + \frac{k}{n}(b - a)$ for $k = 0, 1, \ldots, n$).
- Optionally, p, a positive integer corresponding to the spline degree (by default, $p = 3$, which produces a cubic B-spline).
- Optionally, `piecewise`, the symbol.

- `bspline(`*cpts* ⟨, *var*, *p* ⟩ ⟨, `piecewise` ⟩`)` returns a matrix with $n - 1$ rows and d columns in which the kth row represents a parametric definition of the B-spline with parameter x such that $x \in [x_k, x_{k+1}\rangle$, or, if `piecewise` is given, a parametric representation of the entire spline in which each component is piecewise-defined.
- The points $x_0, x_1, \ldots, x_n$ are called *breakpoints*. B-spline is a piecewise parametric function with parts joining at these points. If no breakpoints are given, a uniform subdivision of the specified segment $[a, b]$ (by default $[0, 1]$) is used. Note that the relation $2 \le n = m - p + 1$ must hold, which implies $m \ge p + 1$.
- B-splines are useful for curve-fitting and numerical differentiation of data.

Example

To define a sequence of control points, enter:

```
> c:=2i,1-2i,5-i,6-4i,8+2i,5+i:;
```

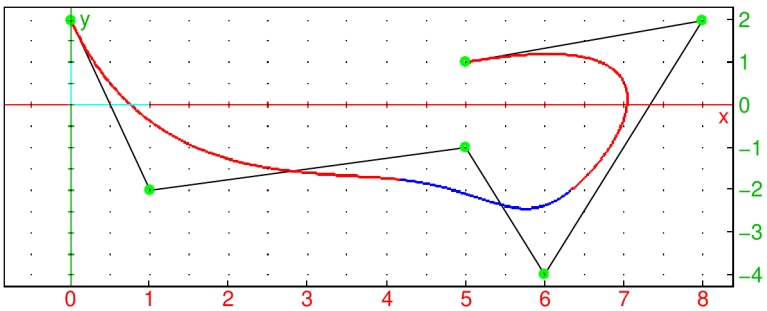
Now compute the B-spline defined by the above points and uniform knots, with parameter $t \in [0, 1]$:

```
> bs:=bspline([c],t)
```

$$\begin{bmatrix} -\frac{99t^3}{2} + 27t^2 + 9t & -\frac{567t^3}{4} + \frac{243t^2}{2} - 36t + 2 \\ \frac{63t^3}{2} - 54t^2 + 36t - 3 & \frac{297t^3}{4} - \frac{189t^2}{2} + 36t - 6 \\ -117t^3 + 243t^2 - 162t + 41 & -162t^3 + 378t^2 - 279t + 64 \end{bmatrix}$$

To plot the result with alternating red-blue color for spline pieces and green control points, enter:

```
> polygon(c,open);
  plotparam(bs[0],t=0..1/3,display=red+line_width_2);
  plotparam(bs[1],t=1/3..2/3,display=blue+line_width_2);
  plotparam(bs[2],t=2/3..1,display=red+line_width_2);
  point(c,display=point_point+point_width_3+green);
```



Sometimes it is useful to return the result in a piecewise form. Enter:

```
> bs:=bspline([c],t,piecewise)
```

$$\left[\begin{cases} -\frac{99}{2}t^3 + 27t^2 + 9t, & t \ge 0 \wedge \frac{1}{3} > t \\ \frac{63}{2}t^3 - 54t^2 + 36t - 3, & \frac{2}{3} > t \\ -117t^3 + 243t^2 - 162t + 41, & 1 \ge t \end{cases}, \begin{cases} -\frac{567}{4}t^3 + \frac{243}{2}t^2 - 36t + 2, & t \ge 0 \wedge \frac{1}{3} > t \\ \frac{297}{4}t^3 - \frac{189}{2}t^2 + 36t - 6, & \frac{2}{3} > t \\ -162t^3 + 378t^2 - 279t + 64, & 1 \ge t \end{cases}\right]$$

The result is a list of components of the spline function. Now, for example, the command line

```
> plotparam(bs,t=0..1)
```

draws the entire spline.

Fitting B-splines to data. The `fitspline` command fits B-splines to (multidimensional) time-stamped data in the least-squares sense.

- `fitspline` takes two mandatory arguments and a sequence of optional arguments:
  - *data*, a list of time-stamped locations in $\mathbb{R}^d$ given either as $[t_k, x_{k1}, x_{k2}, \ldots, x_{kd}]$ or as $[t_k, [x_{k1}, x_{k2}, \ldots, x_{kd}]]$ for $k = 1, 2, \ldots, m$, where $m \geq 2$ and $d \geq 1$.
  - *var*, which is either a variable or a vector of real numbers.
  - Optionally, *opts*, a sequence of options each of which may be one of:
    - *brkpts*, which is either a positive integer n specifying the number of spline pieces (by default, $n = \lfloor \sqrt{m} + 1/2 \rfloor$), or `breakpoint=[`$b_0, b_2, \ldots, b_n$`]` which specifies the breakpoints $b_0, \ldots, b_n \in \mathbb{R}$ explicitly (and not necessarily in ascending order). Note that $\min_i b_i \leq t_1 < t_m \leq \max_j b_j$ must hold. If $brkpts = n$, then $n + 1$ evenly spaced breakpoints are automatically generated in the segment $[t_1, t_m]$.
    - `degree=`p, where p is a positive integer (spline degree, by default $p = 3$).
    - `piecewise`, the symbol.
- `fitspline(`*data*, *var* ⟨, *opts*⟩`)` returns
  - the list of spline points at elements of *var* if the latter is a vector,
  - the spline as returned by `bspline` for the variable *var* and optionally `piecewise`, if *var* is an identifier.

  The resulting spline (of degree p) is computed from a sequence of $n + 1$ breakpoints (either automatically generated or explicitly given) and $n + p$ control points obtained by minimizing squared distances between spline points at values t_k and data points $(x_{k1}, x_{k2}, \ldots, x_{kd})$ for $k = 1, 2, \ldots, m$.
- The number m of data samples must be larger than the number $n+p$ of control points. Furthermore, it is recommended that $n + p \leq m/2$ is satisfied when uniformly spaced breakpoints are used; otherwise the spline could easily overfit on the noise, resulting in large oscillations (this is essentially a Nyquist-frequency argument). If the above inequality does not hold, a warning is printed.
- Setting explicit breakpoints can lead to a better fit in some cases, e.g. when the data curvature varies significantly. It is a good idea to provide more resp. less breakpoints for intervals in which the curvature is large resp. small.
- Note that `fitspline` operates exclusively in floating-point arithmetic. If *var* is a vector, then GSL fitting routines are used if available.

Examples

To generate some synthetic data with noise, enter:

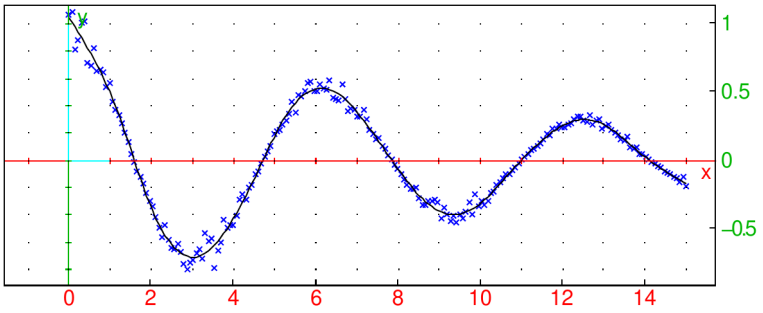
```
> x:=linspace(0,15,200):; y:=apply(t->cos(t)*exp(-0.1*t)*(1+randnorm(0,0.1)),x):;
```

The following command line computes y-values of a B-spline fit:

```
> ys:=fitspline(tran([x,y]),x)]):;
```

Essentially, `ys` is the result of "snapping" `y` to a smooth curve. Now visualize the result by entering:

```
> scatterplot(x,y,color=blue); listplot(tran([x,ys]))
```

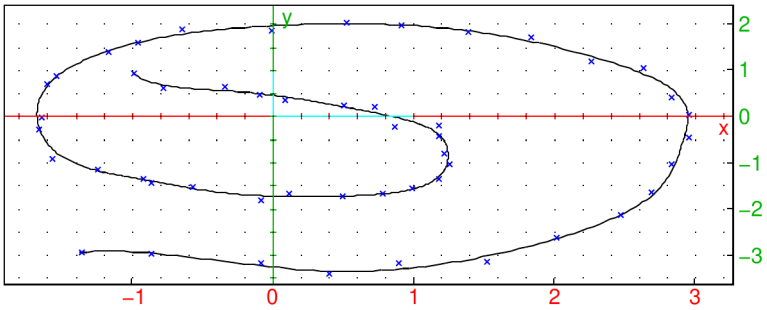


You can use B-spline to trace a sequence of points in time. For example, generate a list of 2D locations by entering:

```
> t:=linspace(0,10,50):;
  x:=apply(u->(u^2*sin(u)+u-1)/(1+u*sqrt(u))+randnorm(0,0.1),t):;
  y:=apply(u->(u^2*cos(u)-u+1)/(1+u*sqrt(u))+randnorm(0,0.1),t):;
```

The underlying smooth path of these “footsteps” can be approximated like this:

```
> t1:=linspace(0,10,500):;
  listplot(fitspline(tran([t,x,y]),t1,20)),scatterplot(x,y,color=blue)
```

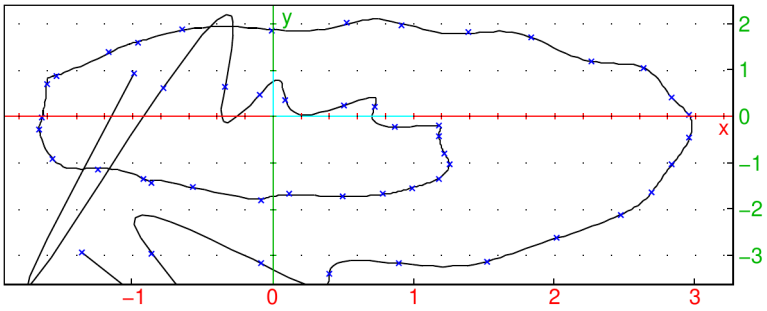


Such approximation is useful for estimating the total distance traveled or the velocity at a certain point in time, assuming that data represents a discretized trajectory of a moving object, such as e.g. a series of GPS positions would.

By setting the number of spline pieces (which is 20 in the above graph) to a value that is close to 50 (the number of samples), the resulting spline exhibits large oscillations near the beginning and end of the curve, reminiscent of Runge’s phenomenon:

```
> listplot(fitspline(tran([t,x,y]),t1,45)),scatterplot(x,y,color=blue)
```

Warning: too many control points



18 Metric properties of curves

18.1 The center of curvature

Let Γ be a curve in space parameterized by a continuously differentiable function, and M_0 be a point on the curve. The curve will have an arclength parameterization; namely, it can be parameterized by a function $M(s)$, where $M(0) = M_0$ and $|s|$ is the length of the curve from M_0 to $M(s)$, in the direction of the curve if $s > 0$ and the opposite direction if $s < 0$.

For such a Γ, the vector $T(s) = M'(s)$ will be the unit tangent to the curve at $M(s)$, and $N(s) = T'(s)$ will be perpendicular to the tangent. The circle through $M(s)$ with center at $M(s) + N(s)$ is called the *osculating circle* to Γ at $M(s)$. Informally, the osculating circle is the circle through $M(s)$ which most closely approximates Γ. The set of all centers of curvature is another curve, called the *evolute* of Γ.

The radius of the osculating circle is $|N(s)|$ and is called the *radius of curvature* of Γ at $M(s)$. The reciprocal of this is called the *curvature* of Γ at $M(s)$.

18.2 Computing the curvature and related values

A curve can be described in Xcas with a parametrization or with a curve object. Various curve objects are described in chapters 26 and 27. The commands in this section can work with curves described either way. You can get the equation of a curve object with the `equation` command (see Section 26.12.7, p. 724).

18.2.1 Curvature of a curve

The `curvature` command finds the curvature of a curve. The curve can be given as an object or by a parameterization.

- To find the curvature from a parameterization, `curvature` takes two mandatory arguments and one optional argument:
  - C, a curve.
  - t, the parameter of the curve.
  - Optionally, t_0, a value of the parameter.
- `curvature(`$C, t\,\langle, t_0\rangle$`)` returns the curvature of the curve; if t_0 is given, the curvature is given at the point it specifies, otherwise the curvature is given as a function of the parameter.
- To find the curvature from a curve object, `curvature` takes two arguments:
  - C, a curve.
  - p, a point on the curve.
- `curvature(`C, p`)` returns the curvature of C at the point p.

Examples

```
> trigsimplify(curvature([5*cos(t),5*sin(t)],t))
```

$$\frac{1}{5}$$

```
> trigsimplify(curvature([2*cos(t),3*sin(t)],t))
```

$$\frac{24\sqrt{2}\sqrt{5\cos(2t)+13}}{260\cos(2t)+25\cos(4t)+363}$$

```
> curvature([2*cos(t),3*sin(t)],t,pi/2)
```

$$\frac{3}{4}$$

```
> curvature(plot(x^2),point(1,1))
```

(see Section 19.2.3, p. 468 and Section 26.5.2, p. 685.)

$$\frac{2}{25}\sqrt{5}$$

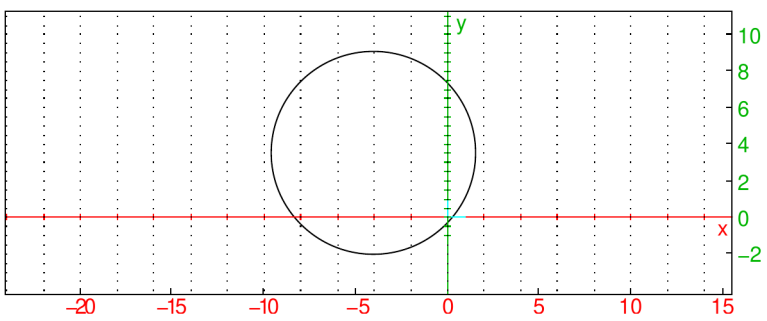
18.2.2 Osculating circle of a curve

The `osculating_circle` command finds and draws the osculating circle of a curve.

- To find the osculating circle from a parameterization, `osculating_circle` takes three arguments:
  - C, a curve.
  - t, the parameter of the curve.
  - t_0, a value of the parameter.
- `osculating_circle`(C, t, t_0) draws and returns the osculating circle of the curve at the point specified by t_0.
- To find the osculating circle from a curve object, `osculating_circle` takes two arguments:
  - C, a curve.
  - p, a point on the curve.
- `osculating_circle`(C, p) draws and returns the osculating circle of C at the point p.

Examples

```
> osculating_circle(plot(x^2),point(1,1))
```



```
> equation(osculating_circle(plot(x^2),point(1,1)))
```

$$(x+4)^2+\left(y-\frac{7}{2}\right)^2=\frac{125}{4}$$

```
> equation(osculating_circle([t^2,t^3],t,1))
```

$$\left(x+\frac{11}{2}\right)^2+\left(y-\frac{16}{3}\right)^2=\frac{2197}{36}$$

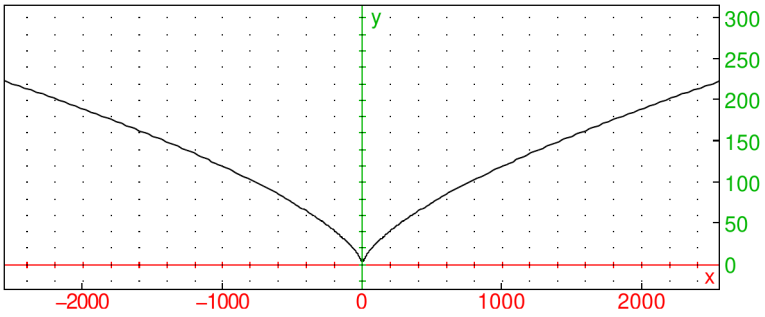
18.2.3 Evolute of a curve

The `evolute` command finds and draws the evolute of a curve.

- To find the evolute from a parameterization, `evolute` takes two arguments:
  - C, a curve.
  - t, the parameter of the curve.
- `evolute(`C,t`)` draws and returns the evolute of the curve.
- To find the evolute from a curve object, `evolute` takes C, a curve.
- `evolute(`C`)` draws and returns the evolute of C.

Examples

```
> evolute(plot(x^2))
```




```
> equation(evolute(plot(x^2)))
```

$$27x^2-16y^3+24y^2-12y+2=0$$

```
> equation(evolute([t^2,t],t))
```

$$16x^3-24x^2+12x-27y^2-2=0$$

19 Graphs

19.1 Generalities

Most graph instructions take expressions as arguments. A few exceptions (mostly MAPLE-compatibility instructions) also accept functions. Some optional arguments, like `color, thickness`, can be used as optional attributes in all graphic instructions. They are described below.

19.1.1 The graphic screen

A graphic screen, either two- or 3D as appropriate, automatically opens in response to a graphic command. Alternatively, you can open a graphic screen with its own command line with keystrokes; Alt+G for a 2D screen and Alt+H for a 3D screen. The graphic screen will have an array of buttons at the top right.

- Red arrows move the image in the x direction.
- Green arrows move the image in the y direction.
- Blue arrows zoom in and out in a 2D screen, and move the image in the z direction in a 3D screen.
- in and out buttons zoom in and out.
- ⊥ button orthonormalizes the graphic.
- ▶I button stops and restarts animations.
- auto button scales the image automatically.
- cfg button brings up the configuration screen (see Section 2.5.7, p. 15).
- M button brings up a menu. The menu has the following submenus:

  View which has entries which do the same as the buttons.

  Trace for working with traces.

  Animation for working with animations.

  3D for working with 3D graphics.

  Export/Print for exporting and printing the graphic.

The image can also be moved in the screen by clicking and dragging with the mouse. Scrolling with the mouse will zoom the images in and out.

19.1.2 Graph and geometric objects attributes

There are two kinds of attributes for graphs and geometric objects: global attributes of a graphic scene and individual attributes for the specific geometric objects in the scene.

Individual attributes

Graphic attributes are optional arguments of the form `display=`*value*. They must be given as the last argument of a graphic instruction. Attributes are ordered in several categories: color, point shape, point width, line style, line thickness, legend value, position and presence. In addition, surfaces may be filled or not, 3D surfaces may be filled with a texture, 3D objects may also have properties with respect to the light. Attributes of different categories may be combined with `+`, e.g. in:

```
> plotfunc(x^2+y^2,[x,y],display=red+line_width_3+filled)
```

You can modify the following graphic attributes:

Color which is set with `display=`*value* or `color=`*value*, where *value* is a nonnegative integer. See Section 19.1.3, p. 458 for more details on colors. The predefined colors in Xcas are: `black`, `white`, `red`, `blue`, `green`, `magenta`, `cyan`, `yellow`, `brown`, `purple`, `violet`, `pink`, `orange`, `grey`, `teal`, `olive`, `navy`, and `gold`.

Point shape which is set with `display=`*value*, where *value* can be one of: `rhombus_point`, `plus_point`, `square_point`, `cross_point`, `triangle_point`, `star_point`, `point_point` or `invisible_point`.

Point width which is set with `display=point_width_`n, where n is an integer between 1 and 7.

Line thickness which is set with `thickness=`n or `display=line_width_`n where n is an integer between 1 and 7.

Line shape which is set with `display=`*value*, where *value* can be one of: `dash_line`, `solid_line`, `dashdot_line`, `dashdotdot_line`, `cap_flat_line`, `cap_square_line` or `cap_round_line`.

Legend where the text is set with `legend="`*legendname*`"` and the position is set with `display=`*value*, where *value* is `quadrant`n for some $n \in \{1, 2, 3, 4\}$. These values correspond to the position of the legend of the object (using the trigonometric plane conventions). The legend is not displayed if the attribute `display=hidden_name` is added.

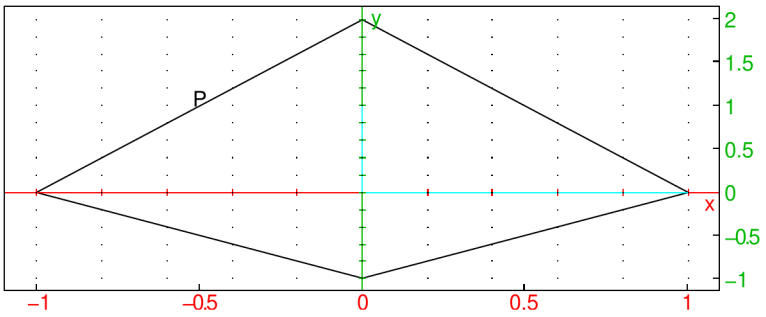
Filling which is set with `display=filled`.

Image texture which is set `gl_texture="`*picture_filename*`"` and fills a surface with a texture. See the interface manual for a more complete description and for `gl_material` options.

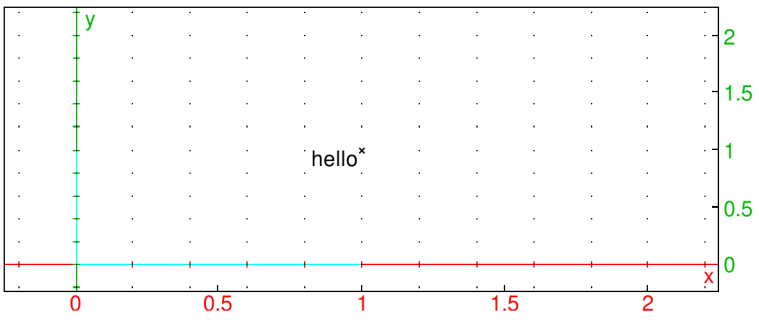
Examples

(See Section 26.9.3, p. 710, Section 26.5.2, p. 685, Section 26.3.3, p. 681 and Section 26.6.3, p. 695 for information on the commands used.)

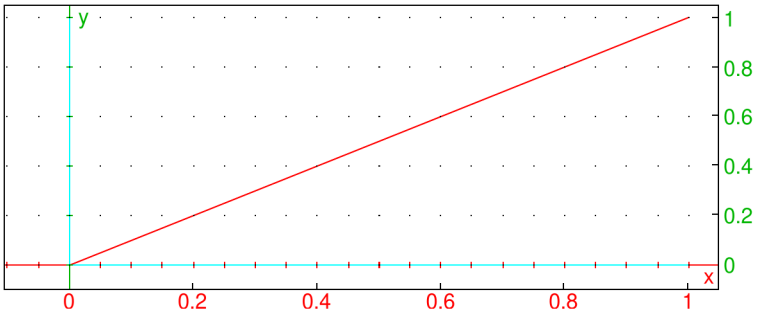
```
> polygon(-1,-i,1,2*i,legend="P")
```




```
> point(1+i,legend="hello")
```




```
> color(segment(0,1+i),red)
```



By entering

```
> segment(0,1+i,color=red)
```

we get the same result as above.

Global attributes

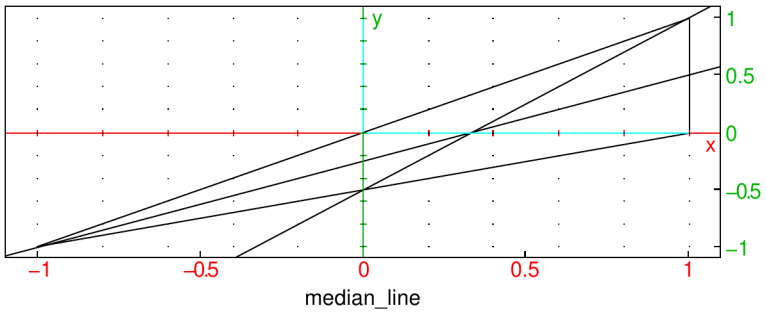
The following attributes are pertinent to the scene as a whole:

- `axes=true` or `axes=false` shows or hides the axes. Alternatively, `axes=`a can be used, where a is a nonnegative integer. The following values have effect:
  - $a = 1$: the same as `axes=true`.
  - $a = 0$: the same as `axes=false`.
  - $a = 2$: show only the relevant portion of the ordinate (used for bar plots).
  - $a = 3$: show the outer frame/ticks but not the green and red inner axes (similar to MATLAB).
  - $a = 4$: same as $a = 3$, but the ordinate ticks are not shown (used for drawing multi-channel waveforms and power spectra, where y-values are not meaningful or significant).
- `title="`*titlename*`"` sets the title.
- `labels=["`*xname*`","`*yname*`","`*zname*`"]` sets names of the x, y, z axes.
- `gl_x_axis_name="`*xname*`"`, `gl_y_axis_name="`*yname*`"`, `gl_z_axis_name="`*zname*`"` sets the names of the axes individually.
- `legend=["`*xunit*`","`*yunit*`","`*zunit*`"]` sets units for the axes.
- `gl_x_axis_unit="`*xunit*`"`, `gl_y_axis_unit="`*yunit*`"`, `gl_z_axis_unit="`*zunit*`"` sets units for the axes individually.
- `gl_texture="`*filename*`"` sets the background image to *filename*.

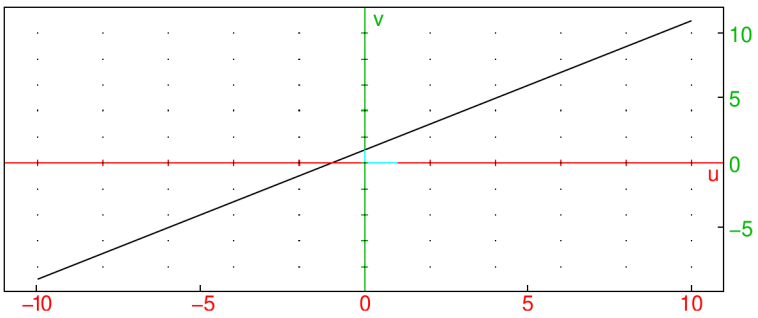
- `gl_x=`*xmin*`..`*xmax*, `gl_y=`*ymin*`..`*ymax*, `gl_z=`*zmin*`..`*zmax* sets the graphic configuration (do not use for interactive scenes)
- `gl_xtick=`*xmark*, `gl_ytick=`*ymark*, `gl_ztick=`*zmark* sets the tick marks for the axes.
- `gl_shownames=true` or `gl_shownames=false` shows or hides objects names
- `gl_rotation=[`x, y, z`]` defines the rotation axis for the animation rotation of 3D scenes.
- `gl_quaternion=[`x, y, z, t`]` defines the quaternion for the visualization in 3D scenes (do not use for interactive scenes).
- a few other OpenGL light configuration options are available but not described here.

Examples

```
> title="median_line";triangle(-1-i,1,1+i);median_line(-1-i,1,1+i);
  median_line(1,-1-i,1+i);median_line(1+i,1,-1-i)
```




```
> labels=["u","v"];plotfunc(u+1,u)
```



Formatting textual annotations

There are several formatting options for the textual attributes (`title`, `labels`, and `legend`). Note that they can be used only in 2D plots.

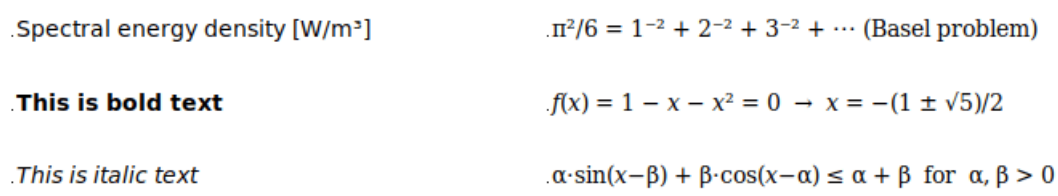
- `"*This is bold*"` draws the text in a bold sans serif font.
- `"/This is italic/"` draws the text in an italic sans serif font.
- `"$This is mathmode$"` draws the text in a serif font and enables the following mathematical typesetting options:
  - `^`*int*, where *int* is an integer, prints *int* in the exponent (also available outside math mode).
  - `@`*char*, where *char* is a letter, draws *char* with slanted font (useful for variables in formulas).

- `-` (double hyphen) prints as a minus sign − (which is different than the ordinary hyphen!).
- `<=` resp. `>=` prints as ≤ resp. ≥.
- `+-` resp. `-+` prints as ± resp. ∓.
- `==` prints as ≡.
- `!=` prints as ≠.
- `~~` prints as ≈.
- `\/` prints as $\sqrt{}$. Note that the backslash must be escaped, i.e. entered as `\\`.
- `->` prints as →.
- `=>` prints as ⇒.
- `**` prints as ⋯ (vertically centered dots).
- `*`, when preceded/followed by an alphanumeric character or parenthesis, is printed as · (the multiplication dot).
- `__` (double underscore) prints as — (em dash).
- `^C` prints as °C.
- `^F` prints as °F.

Note that changing font to bold, italic or serif typeface can only be applied to the entire text. The `@`-option uses special slanted Unicode symbols.

Example

```
> axes=0;
  legend(i,"*This is bold text*");
  legend(.5i,"/This is italic text/");
  legend(1.5i,"Spectral energy density [W/m^3]");
  legend(1+1.5i,"$pi^2/6 = 1^-2 + 2^-2 + 3^-2 + ** (Basel problem)$");
  legend(1+i,"$@f(@x) = 1 -- @x -- @x^2 = 0   =>   @x = --(1 +- \\/5)/2$");
  s1:="alpha*sin(@x--beta) + beta*cos(@x--alpha) <= alpha + beta":;
  s2:="alpha, beta > 0":;
  legend(1+.5i,"$"+s1+"  for  "+s2+"$")
```



19.1.3 Colors and color functions

Xcas uses RGB565 color encoding from FLTK and is therefore able to use at most 65536 RGB colors in graphs. In RGB565 encoding, there are only 32 values for the red and blue channels and 64 values for the green channel. The alpha channel (transparency) is not used.

Colors in Xcas are nonnegative integers with a dedicated subtype. They are evaluated to small colored squares in command line entries.

- Colors from 0 to 255 correspond to indices in the FLTK colormap (see here).
- Colors from 256 to 381 belong to the default Xcas rainbow colormap (see Section 26.3.2, p. 678).
- Values from 382 to 511 are not assigned.
- Values from 512 to 65535 are RGB colors with channel values stored in the index. The red channel is read from the upper 5 bits, the green channel is read from the middle 6 bits and the blue channel is read from the lower 5 bits.

You can use either one of the predefined colors (see Section 19.1.2, p. 454) or create color(s) by using the `rgb`, `hsv` and `colormap` commands (see below).

For example, to show all predefined colors, enter:

```
> white,grey,olive,teal,cyan,green,brown,red,orange,
  gold,yellow,pink,magenta,purple,violet,blue,navy,black
```

The above colors are portable since they are defined in the FLTK colormap.

RGB and HSV colors

The `rgb` command converts between color objects and their RGB (Red, Green, Blue) specifications.

- `rgb` can take either
  - *col*, a color object,
  - *ind*, a nonnegative integer, or
  - *rgb*, a RGB color specification given as either
    - a vector of three integers between 0 and 255,
    - a vector of three real numbers between 0 and 1,
    - a hexadecimal number `0xrrggbb`, or
    - a hexadecimal string (optionally with the leading `#` character).
- `rgb(`*col*`)` returns the RGB specification of *col* in form of a vector with three integers between 0 and 255.
- `rgb(`*ind*`)` returns the color object corresponding to the index *ind*, which may be any number between 0 and 65535 (for unassigned indices, `rgb` returns the black color).
- `rgb(`*rgb*`)` returns the color object which is closest to the given RGB specification.

The `hsv` command converts between color objects and their HSV (Hue, Saturation, Value) specifications.

- `hsv` takes either *col*, a color object, or *hsv*, a vector containing either
  - three real numbers in $[0, 1]$, or
  - three integers h, s and v such that $0 \le h \le 360$ and $0 \le s, v \le 100$ (if other values are given, then h mod 360, $\min\{100, \max\{0, s\}\}$ and $\min\{100, \max\{0, v\}\}$ are used).
- `hsv(`*col*`)` returns the HSV specification of *col* as a vector $[h, s, v]$ of integers.
- `hsv(`*hsv*`)` returns the color object closest to the given HSV specification.

You can convert from RGB to HSV and vice versa by using the `rgb2hsv` and `hsv2rgb` commands.

- `rgb2hsv` takes *rgb*, a vector of three integers between 0 and 255 or three real numbers between 0 and 1.
- `rgb2hsv(`*rgb*`)` returns the HSV specification equivalent to *rgb* as a vector $[h, s, v]$ of integers between 0 and 360 for h resp. 0 and 100 for s and v.
- `hsv2rgb` takes *hsv*, a vector of three integers between 0 and 360 for h resp. 0 and 100 for s and v, or three real numbers between 0 and 1.
- `hsv2rgb(`*hsv*`)` returns the RGB specification equivalent to *hsv* as a vector $[r, g, b]$ of integers between 0 and 255.

Examples

```
> rgb("#cf3f00"); rgb(0x00f3fc)
```

(,)

```
> b:=rgb(216); rgb(255,0,0); rgb([0.0,1.0,0.0]);
```

(, ,)

```
> spec:=rgb(b)
```

$[0, 0, 255]$

```
> rgb2hsv(spec)
```

$[240, 100, 100]$

```
> seq(hsv(15*h,100,80),h=0..20)
```

(,)

Colormaps

The `colormap` command returns built-in continuous/discrete colormaps or individual colors from the corresponding palettes. Note that the colors obtained this way are not guaranteed to be portable.

- `colormap` takes one to three arguments:
  - "*pal*", where *pal* is the colormap name. The following continous colormaps are emulated from the `pals` package in R: `magma`, `inferno`, `plasma`, `viridis`, `jet`, `parula`, `gnuplot`, `cividis`, `cubehelix`, `cyclic1`, `cyclic2`, `cyclic3`, `cyclic4`, `phase`, `isoluminant`, `cubic`, `coolwarm`, `haline`, `blues`, `greens`, `greys`, `oranges`, `reds`, and `spectral`. `cyclic1`-`cyclic4` and `phase` are cyclic palettes. Only one discrete colormap is implemented: `polychrome` (alias `discrete`), which contains 36 colors listed in Table 19.1. The default XCAS rainbow colormaps are named `default` (cyclic) and `rainbow` (non-cyclic).

    Instead of `"blues"`, `"greens"`, `"greys"`, `"oranges"` or `"reds"`, you can pass `blue`, `green`, `grey`, `orange` or `red` as the first argument, respectively.

dark purplish gray	purplish white	vivid red	vivid purple
vivid yellowish green	strong purplish blue	vivid orange yellow	vivid purplish red
brilliant green	vivid yellow green	vivid blue	brilliant purple
vivid violet	strong pink	strong blue	strong reddish orange
vivid green	light olive brown	vivid reddish purple	vivid greenish yellow
vivid yellowish green	vivid red	vivid purplish red	pale yellow
strong reddish purple	vivid violet	vivid yellow green	very light blue
strong reddish brown	very light yellowish green	very light bluish green	deep greenish blue
vivid purple	deep purple	brilliant blue	vivid violet

Table 19.1: Colors in the `polychrome` palette

  - Optionally, *modifiers*, a sequence that can contain `inv` and/or `reverse` symbols for inverting/reversing the colormap colors. This can be used only with continuous colormaps.
  - Optionally, either n, a nonnegative integer, t, a real number in $[0,1]$, or "*str*", a string. The latter can be used only with the discrete colormap `polychrome`. The string *str* is a color name or a string which is searched for among color names.

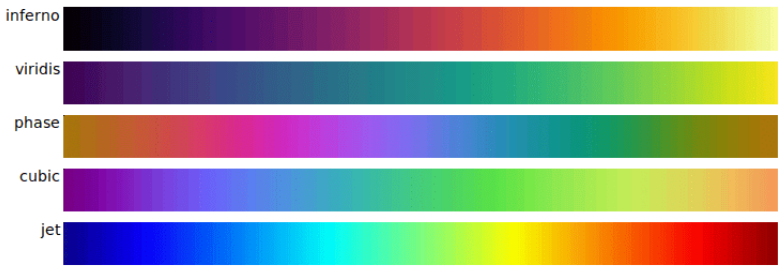
    Alternatively, if *pal* specifies a continuous colormap, the last argument can be the symbol `display`, in which case the colormap is drawn. In this case *pal* can also be a list of colormap names, resulting in the specified colormaps drawn one above another in the same picture.

- `colormap(`*pal*⟨, *modifiers*⟩`)` returns the colormap index if *pal* is a continuous colormap or the list of all colors in the palette if *pal* is a discrete colormap. The colormap index is used by commands such as `plotfunc` (see Section 19.2.1, p. 464), `plotdensity` (see Section 19.5.2, p. 479) and `hht` (see Section 21.4.9, p. 602).
- `colormap(`*pal*⟨, *modifiers*⟩, n`)` returns n equally spaced colors from a continuous colormap *pal*. If *pal* is a discrete colormap, then the nth color is returned (n is wrapped around if it exceeds the number of colors in the palette).
- `colormap(`*pal*⟨, *modifiers*⟩, t`)` returns the color of a continuous colormap *pal* which corresponds to the value t assuming that the first and last colors are mapped to 0 and 1, respectively.
- `colormap(`*pal*⟨, *modifiers*⟩, *str*`)` returns the list of colors of a discrete colormap *pal* containing *str* in their names (whole-word search is used). If *str* is a full color name, only that color is returned.

Examples

To visualize some colormaps, enter:

```
> colormap(["inferno","viridis","phase","cubic","jet"],display)
```



The above colormaps do not appear perfectly seamless; this is due to the 16-bit color coding in Xcas instead of 24-bit RGB888 (which would allow for millions of colors). Note that clamping colors to 16

bits, which results in a slight *posterization effect*, pertains only to XCAS graphic commands, while images are displayed with exact colors (see Section 28.1.4, p. 814).

You can invert and/or reverse colors in a colormap, obtaining a new one. A color is inverted by subtracting its R, G and B values from 255. For example, to obtain a "frozen" version of the `inferno` colormap, enter:

```
> colormap("inferno",inv,reverse,display)
```

To get a single color from a colormap, enter e.g.

```
> colormap("viridis",0.8)
```

To get a list of uniformly spaced colors, enter e.g.

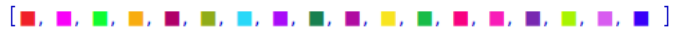
```
> colormap("blues",reverse,20)
```

To display the discrete colormap, enter:

```
> axes=0; gl_ortho=1;
  seq(rectangle(k,k+1,2,color=filled+colormap("discrete",k)),k=0..35)
```

Useful color combinations can be generated from the discrete colormap by filtering out groups of similar colors. For example:

```
> colormap("discrete","vivid")
```



```
> colormap("discrete","strong")
```

You could also try passing e.g. `"very light"`, `"brilliant"`, `"reddish"`, `"green"`, `"deep"` etc. as the second argument (see Table 19.1). These and similar color combinations can be useful for categorical data visualization techniques such as bar plots, pie charts, clustering etc.

Color interpolation and RGB to XYZ conversion

The `interp` command interpolates between given colors in the CIE 1931 XYZ color space with the sRGB gamut (see here for details). In XYZ space, the Euclidean distance between two colors is roughly proportional to their perceived difference. Such interpolation is called *perceptually uniform*.

See Section 17.1.1, p. 435, Section 17.1.2, p. 438 and Section 28.1.10, p. 821 for other uses of `interp`.

- `interp` takes two or three arguments:
  - *colors*, which may be either
    - a sequence of two color objects c_1 and c_2,

    * a list of m colors objects $c_1, \ldots, c_m$, or
    * a matrix of integers with m rows and three columns in which each row defines a RGB color.
  - Either n, a nonnegative integer, or t, a real number or a list of real numbers in $[0, 1]$.

- `interp(`*colors*$, n$`)` returns n equally spaced colors on a 3D Bézier curve defined by the XYZ color space equivalents $d_1, \ldots, d_m$ of colors $c_1, \ldots, c_m$ (see here for the details on conversion). The output does not include boundary colors c_1 and c_m. For instance, for $n = 1$ you obtain the single color at the middle of the curve, while for $n = 2$ you obtain the colors at one-third resp. two-thirds into the curve.
- `interp(`*colors*$, t$`)` returns the color(s) on the parametrized Bézier curve corresponding to value(s) of t such that $t = 0$ resp. $t = 1$ results in c_1 resp. c_m.
- The type of input matches the type of output. You cen specify the input as a list of either color objects or RGB triples. Beware that, if you input color objects, you will get color objects in the output, which are clamped to 16 bits in order to fit the RGB565 encoding. With a RGB input, `interp` returns a RGB matrix which defines a list of RGB888 colors, thus avoiding the loss of channel information.

You can access the routines for RGB$\leftrightarrow$XYZ conversion by using the `rgb2xyz` and `xyz2rgb` commands which take a list or sequence of three real numbers in $[0, 1]$ representing the RGB or XYZ triple and return the other color specification in the same form. The `rgb2xyz` command can also take color objects as inputs. These routines apply chromatic adaptation with Bradford's method to make the white color in Xcas the reference white (the default sRGB reference white is D65); thus the white color corresponds to $[1.0, 1.0, 1.0]$ and black color to $[0.0, 0.0, 0.0]$ in both RGB and XYZ.

Examples

To demonstrate the conversion between RGB and XYZ, enter:

```
> xyz:=rgb2xyz(0,255,255)
```

or:

```
> xyz:=rgb2xyz(0.0,1.0,1.0)
```

or:

```
> xyz:=rgb2xyz(cyan)
```

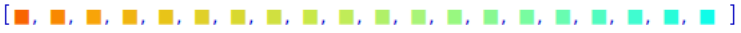
$$[0.561504215261, 0.777146370045, 0.982679191019]$$

```
> xyz2rgb(xyz)
```

$$[0, 255, 255]$$

To demonstrate color interpolation, enter:

```
> c:=[red,yellow,green,cyan]:;
  interp(c,20)
```



The above list consists of RGB565 colors. To visualize a perceptually seamless transition from red to cyan passing through yellows and greens, you should convert c to a RGB matrix, which can be done by using `apply` and `rgb`. Enter:

```
> res:=interp(apply(rgb,c),1000):;
```

The output is a 1000×3 RGB matrix from which you can create an image by entering:

```
> r,g,b:=[col(res,0)$100],[col(res,1)$100],[col(res,2)$100]:;
  img:=image(3,r,g,b);
```

an image of size 1000×100 (RGB)

To display the result with RGB888 colors, enter:

```
> display(img)
```

19.2 Graph of a function

19.2.1 2D graph

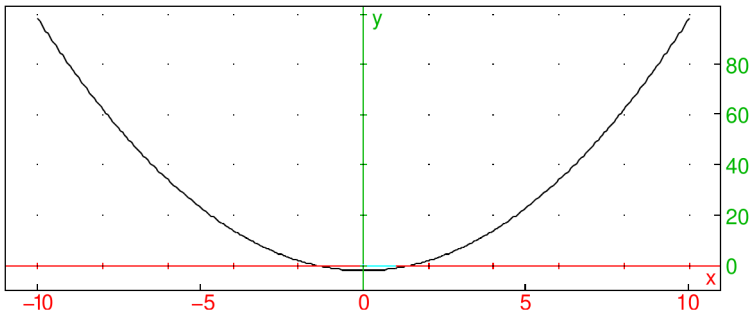
The `plotfunc` or `funcplot` command draws the graph of a function.

`plotfunc` can draw the graph of a one-variable function or a two-variable function; this section will discuss one-variable functions and the next section will discuss two-variable functions.

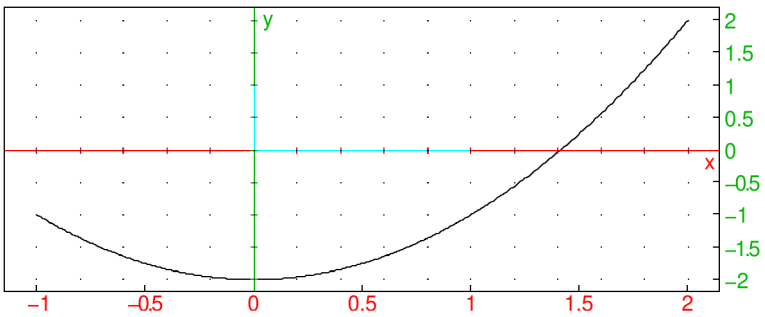
- `plotfunc` takes one mandatory argument and two optional arguments:
  - *expr*, an expression defining a function.
  - Optionally, *var*, the variable name (by default `x`) possibly with bounds. If the variable is given as $var = a..b$, the graph will be drawn from a to b, otherwise it will be graphed over the default interval (see Section 2.5.8, p. 17).
  - Optionally, *opt*, which can be `xstep=`n to specify the discretization step or `nstep=`n to specify the number of points used to graph.
- `plotfunc(`*expr*, *var*⟨, *opt*⟩`)` draws the graph.

Examples

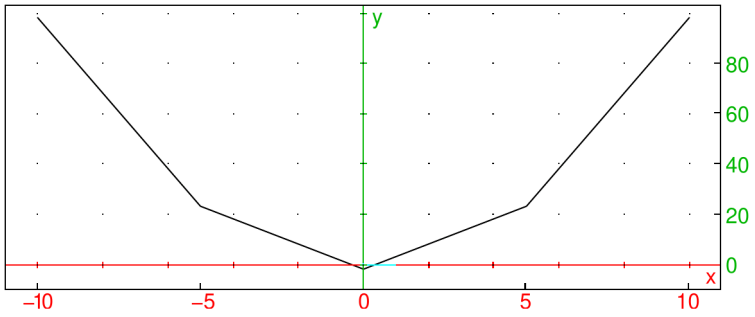
```
> plotfunc(x^2-2)
```



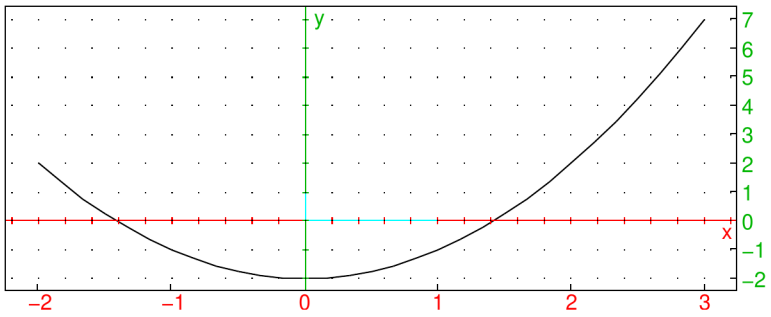

```
> plotfunc(a^2-2,a=-1..2)
```




```
> plotfunc(x^2-2,x,xstep=5)
```




```
> plotfunc(x^2-2,x=-2..3,nstep=30)
```



19.2.2 3D graph

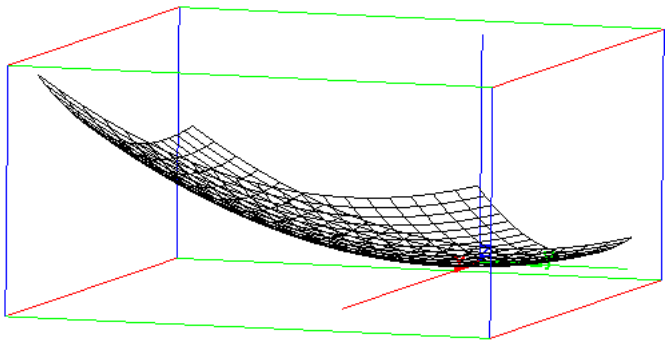
Functions of two variables

The `plotfunc` can draw the graphs of two-variable function.

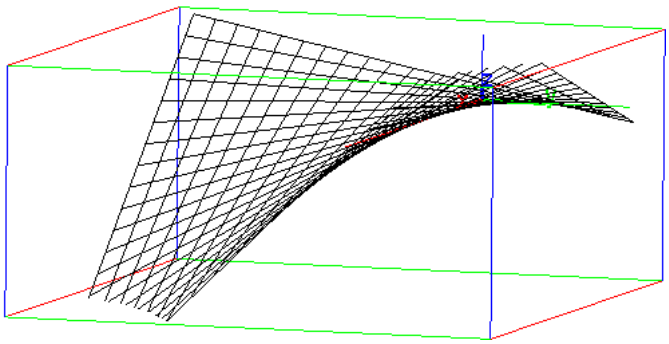
- `plotfunc` takes two mandatory argument and two optional arguments:
  - *expr*, an expression defining a function of two variables or a list of such expressions.
  - *vars*, a list of the variable names, possibly with bounds. If the variable is given as $var = a..b$, the graph will be drawn for that range of that variable, otherwise it will be graphed over the default interval (see Section 2.5.8, p. 17).
  - Optionally, *xstep*, which can be `xstep=`n to specify the discretization step in the x direction.
  - Optionally, *ystep*, which can be `ystep=`m to specify the discretization step in the y direction.
  - Instead of *xstep* and `ystep`, you could use the option `nstep=`n to specify the number of points used to graph.
- `plotfunc(`*expr*, *vars* ⟨, *xstep*, *ystep* ⟩`)` draws the graph.

Examples

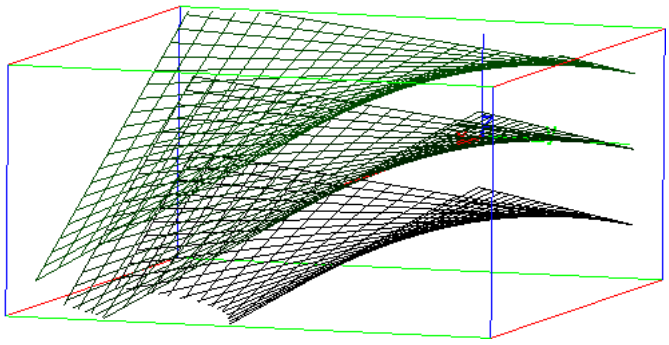
```
> plotfunc(x^2+y^2,[x,y])
```



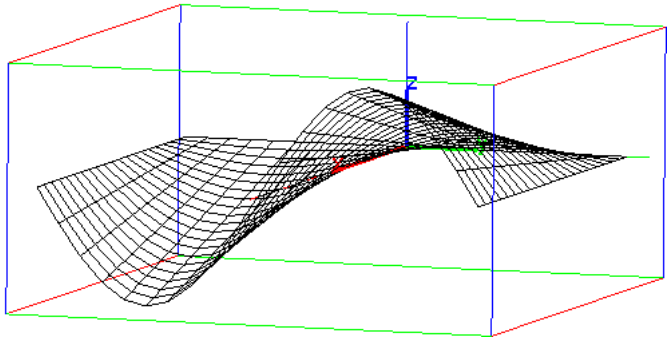
```
> plotfunc(x*y,[x,y])
```



```
> plotfunc([x*y-10,x*y,x*y+10],[x,y])
```

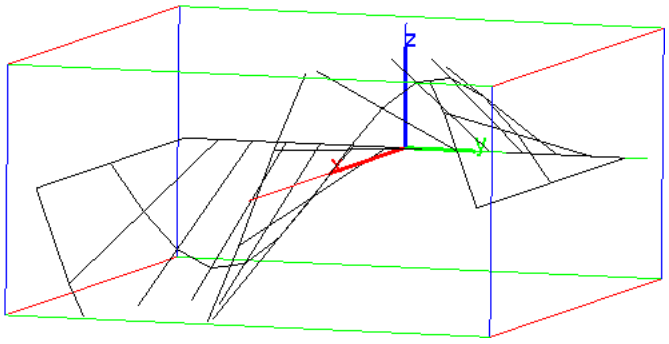


```
> plotfunc(x*sin(y),[x=0..2,y=-pi..pi])
```



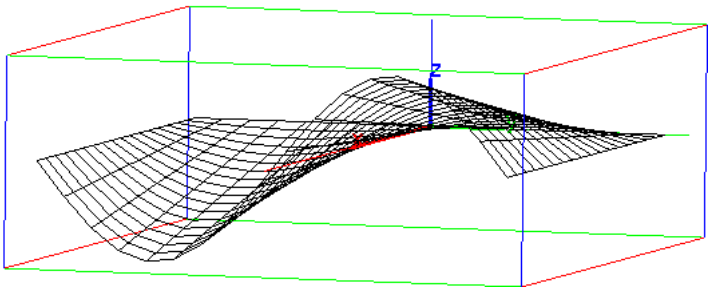
As an example where you specify the x and y discretization step with `xstep` and `ystep`:

```
> plotfunc(x*sin(y),[x=0..2,y=-pi..pi],xstep=1,ystep=0.5)
```



Alternatively you can specify the number of points used for the representation of the function with `nstep` instead of `xstep` and `ystep`.

```
> plotfunc(x*sin(y),[x=0..2,y=-pi..pi],nstep=300)
```



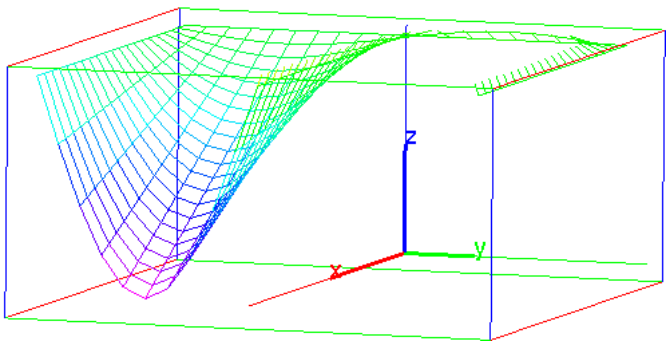
Remarks.

- Like any 3D scene, the viewpoint may be modified by rotation around the `x` axis, the `y` axis or the `z` axis, either by dragging the mouse inside the graphic window (push the mouse outside the parallelepiped used for the representation), or with the shortcuts `x`, `X`, `y`, `Y`, `z` and `Z`.
- If you want to print a graph or get a LaTeX translation, use Menu ▶ print ▶ Print (with Latex).

3D graph with rainbow colors

If the expression with two variables is purely imaginary, i*expr*, then `plotfunc` will still draw the graph, but the color will depend on the height $z = expr$ resulting in a rainbow colored surface. This provides you with an easy way to find points having the same third coordinate. For example:

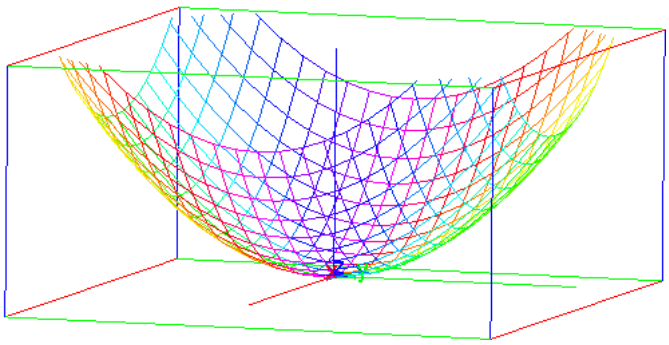
```
> plotfunc(i*x*sin(y),[x=0..2,y=-pi..pi])
```



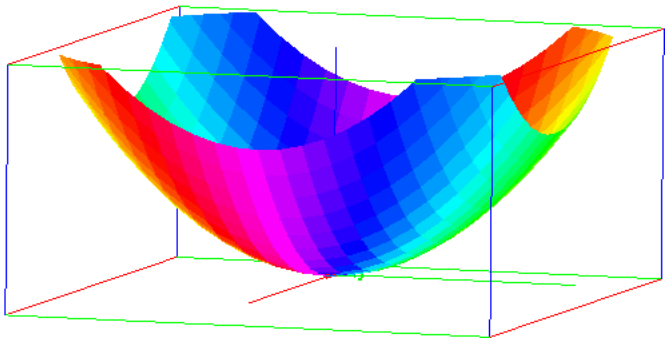
"4D" graph

If *expr* is a complex valued expression whose real part is not identically zero on the discretization mesh, then `plotfunc` will draw the surface $z =$ `abs(`*expr*`)`, where `arg(`*expr*`)` determines the color from the rainbow. This gives you an easy way to see the points having the same argument. Note that if the real part of *expr* is zero on the discretization mesh, then it will look purely imaginary to `plotfunc` and will represented with rainbow colors, as in Section 19.2.2, p. 467. For example:

```
> plotfunc((x+i*y)^2,[x,y])
```

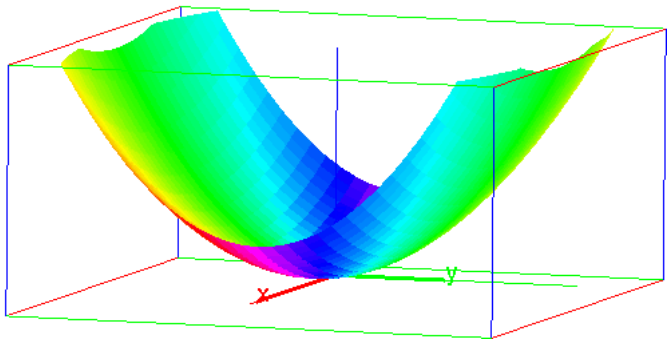


```
> plotfunc((x+i*y)^2,[x,y],display=filled)
```



You can specify the range of variation of x and y and the number of discretization points.

```
> plotfunc((x+i*y)^2,[x=-1..1,y=-2..2],nstep=900,display=filled)
```



19.2.3 2D graph for Maple compatibility

The `plot` command is a MAPLE-compatible way to draw the graph of a univariate function.

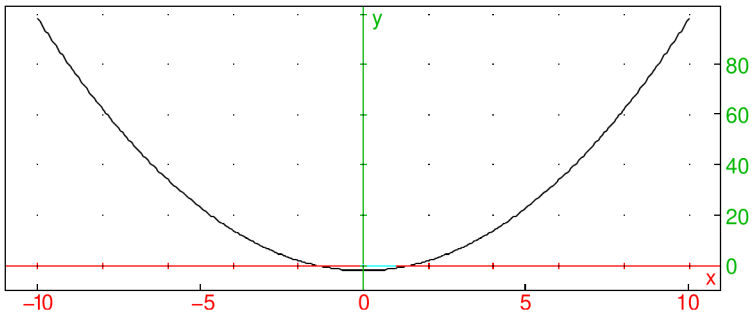
- `plot` takes one mandatory argument and two optional arguments:
  - *func*, a function or an expression involving one variable.

  - Optionally, *var* the name of the variable in the expression (if *func* is an expression), which can also specify a range of values $var = a..b$ (by default it is `x`). If *func* is a function, the optional second argument can simply be a range $a..b$ for the variable.
  - Optionally, *opt*, which can be `xstep=`n to specify the discretization step or `nstep=`n to specify the number of points used to graph.

- `plot(`$expr\langle, var, opt\rangle$`)` draws the graph.

Example

```
> plot(x^2-2,x)
```



19.2.4 3D surfaces for Maple compatibility

The `plot3d` command is a MAPLE-compatible way to draw a surface. It can plot the graph of a function of two variables or a surface given by a parameterization.

- To draw the graph of a function, `plot3d` takes three arguments:
  - *func*, a function or an expression involving two variables.
  - x and y, the names of the variable in the expression (if *func* is an expression) which can also specify a range of values for each variable.

    If *func* is a function, this argument is optional, and are the ranges $a..b$ for the variables.

    If the ranges are not given, the default values are taken from the graph configuration (see Section 2.5.8, p. 17).

- `plot3d(`$func, x, y$`)` draws the graph.

- To draw a parameterized surface, `plot3d` takes one mandatory argument and two optional arguments:
  - *funcs*, a list of three functions or three expressions involving two variables.
  - u and v, the names of the variable in the expression (if *funcs* is a list of expressions), which can also specify a range of values for each variable.

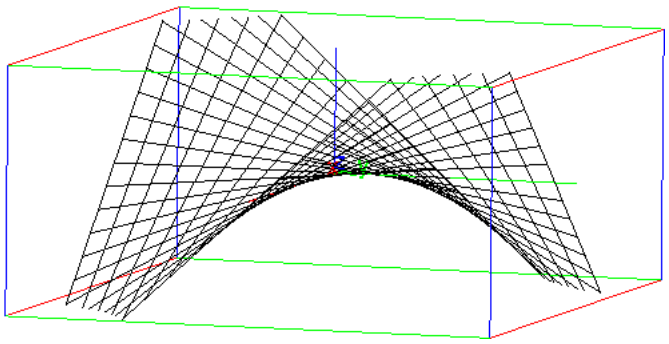
    If *funcs* is a list of functions, this argument is optional, and are the ranges $a..b$ for the variables.

    If the ranges are not given, the default values are taken from the graph configuration (see Section 2.5.8, p. 17).

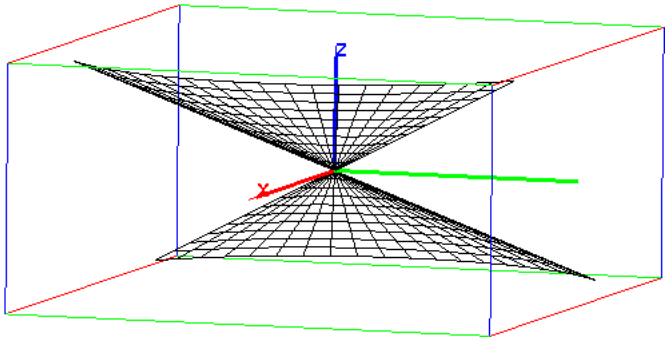
- `plot3d(`$funcs, u, v$`)` draws the surface.

Examples

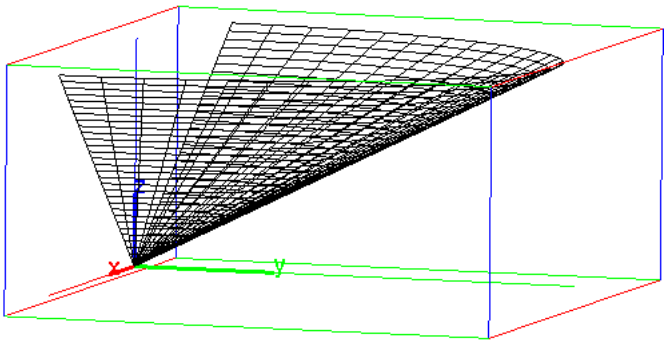
```
> plot3d(x*y,x,y)
```



```
> plot3d([v*cos(u),v*sin(u),v],u,v)
```




```
> plot3d([v*cos(u),v*sin(u),v],u=0..pi,v=0..3)
```



19.2.5 A note on graphing expressions

If a graph depends on a user-defined function, you may want to define the function when the parameter is a formal variable. For this, it can be useful to test the type of the parameter while the function is being defined. (See Chapter Section 25, p. 647 for information about programming in Xcas.)

For example, suppose that `f` and `g` are defined by:

```
f(x):={
  if (type(x)!=DOM_FLOAT) return 'f'(x);
  while (x>0) { x--; }
  return x;
}
```

and

```
g(x):={
  while (x>0) { x--; }
  return x;
}:;
```

Graphing these (see Section 19.2.1, p. 464):

```
> F:=plotfunc(f(x));
  G:=plotfunc(g(x))
```

they will both produce the same graph. However, the graphic `G` won't be reusable. Entering:

```
> F
```

reproduces the graph, but entering:

```
> G
```

produces the error:

Unable to eval test in loop: x>0.0
Error: Bad Argument Value Error:
Bad Argument Value

Internally, `F` and `G` contain the formal expressions `f(x)` and `g(x)`, respectively. When XCAS tries to evaluate `F` and `G`, `x` has no value and so the test `x>0` produces an error in `g(x)`, but the problem is avoided in the second line of `f(x)`.

19.3 Graph of a line and tangent to a graph

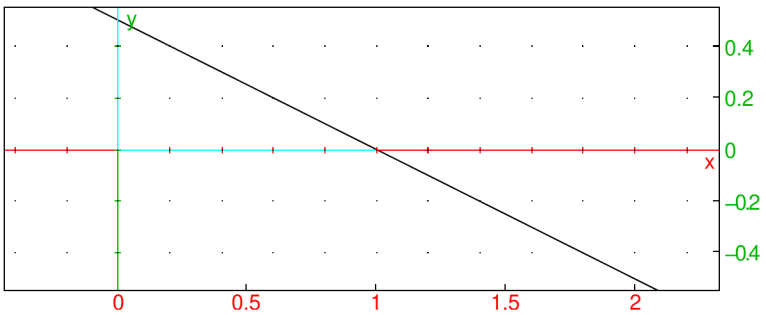
19.3.1 Drawing a line

The `line` command draws and finds lines in $\mathbb{R}^2$ and $\mathbb{R}^3$.

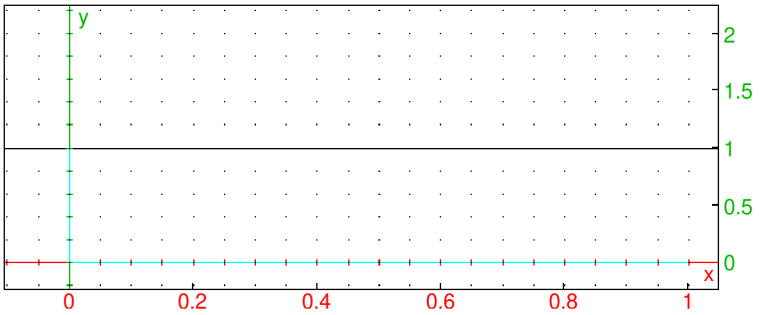
- For a line in $\mathbb{R}^2$, `line` takes *eqn*, a linear equation in the variables `x` and `y`.
- `line(`*eqn*`)` draws and returns the line given by the equation.
- For a line in $\mathbb{R}^3$, `line` takes two arguments: *eqn1* and *eqn2*, two linear equations in the variables `x`, `y` and `z`.
- `line(`*eqn1*, *eqn2*`)` draws and returns the line which is the intersection of the planes given by the equations.

Examples

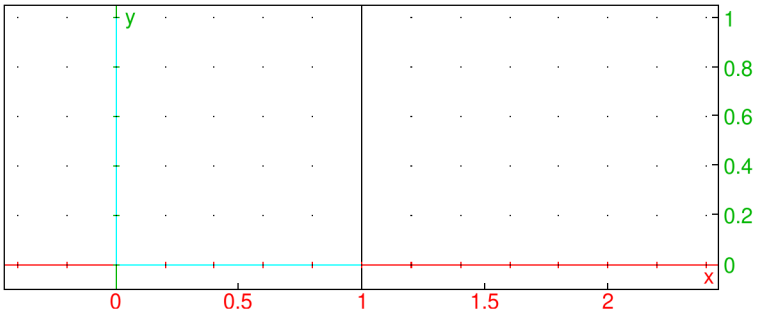
```
> line(2*y+x-1=0)
```



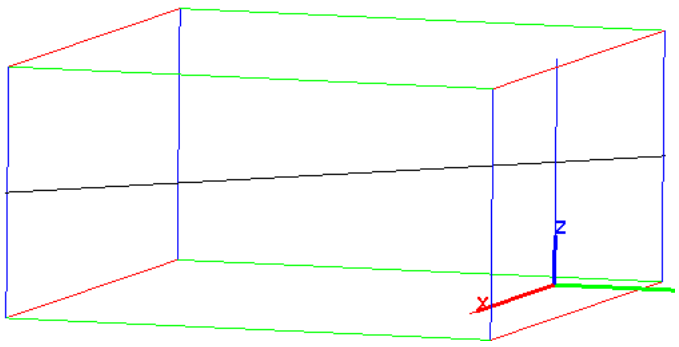
```
> line(y=1)
```



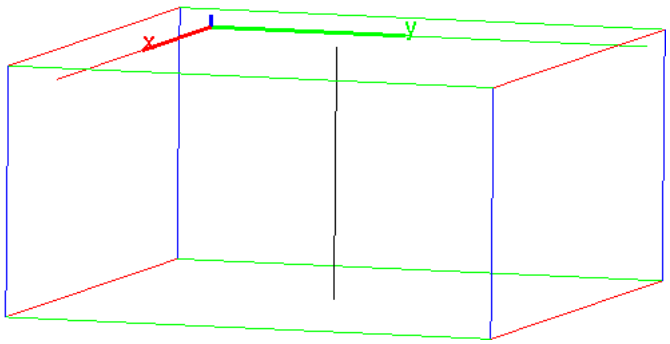

```
> line(x=1)
```




```
> line(x+2*y+z-1=0,z=2)
```




```
> line(y=1,x=1)
```



Remark. `line` defines an oriented line:

- When a 2D line is given by an equation, it is rewritten as $lhs - rhs = ax + by + c = 0$, this determines its normal vector $[a, b]$ and the orientation is given by the vector $[b, -a]$.

- When a 3D line is given by two plane equations, its direction is defined by the cross product of the normals to the planes.

  When the plane equation is rewritten as $lhs - rhs = ax + by + cz + d = 0$, then the normal is $[a, b, c]$. For example, `line(x=y,y=z)` draws the line $x - y = 0, y - z = 0$ and its direction is:

$$[1, -1, 0] \times [0, 1, -1] = [1, 1, 1]$$

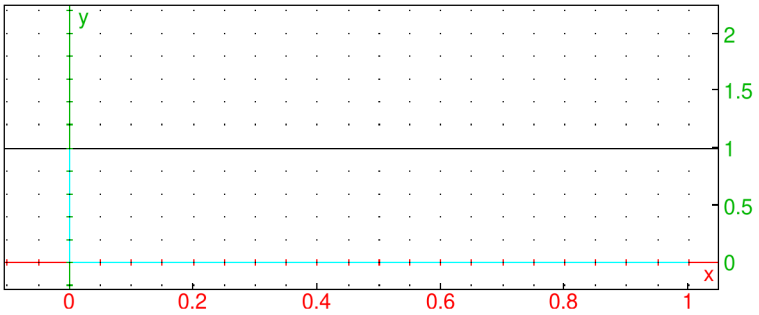
19.3.2 Drawing a 2D horizontal line

The `LineHorz` command draws a horizontal line in $\mathbb{R}^2$.

- `LineHorz` takes a, a number.
- `LineHorz(`a`)` draws the horizontal line $y = a$.

Example

```
> LineHorz(1)
```



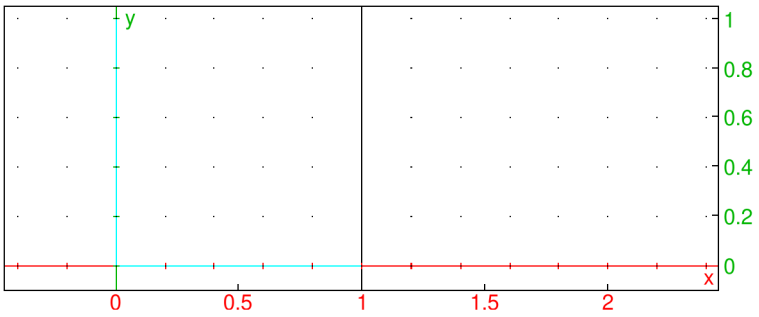
19.3.3 Drawing a 2D vertical line

The `LineVert` command draws a vertical line in $\mathbb{R}^2$.

- `LineVert` takes a, a number.
- `LineVert(`a`)` draws the vertical line $x = a$.

Example

```
> LineVert(1)
```



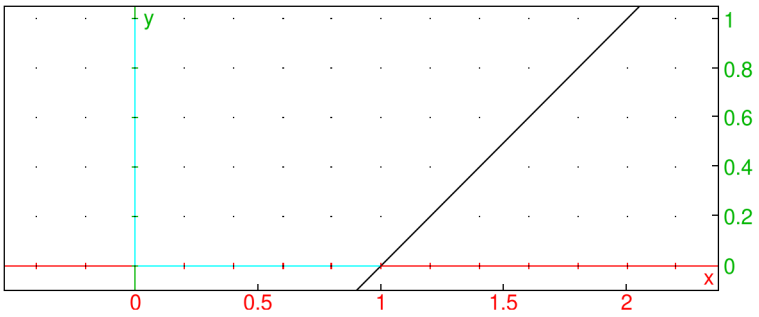
19.3.4 Tangent to a 2D graph

The `LineTan` command draws tangent lines to graphs.

- `LineTan` takes two arguments:
  - *expr*, in the variable `x`.
  - x_0, a value of `x`.
- `LineTan(`*expr*$, x_0$`)` draws the tangent at `x` $= x_0$ to the graph of *expr*.

Example

```
> LineTan(ln(x),1)
```




```
> equation(LineTan(ln(x),1))
```

$$y = (x - 1)$$

19.3.5 Tangent to a 2D graph

The `tangent` command draws tangents to surfaces.

- `tangent` takes two arguments:
  - S, the graph of a two-variable function or a geometric object (see chapter 27).
  - A, a point on S or a number (if S is a graph).
- `tangent(`S, A`)` draws tangent(s) to S passing through A.

Example

Define the function `g`:

```
> g(x):=x^2
```

then the graph `G` of `g` and a point A on the graph:

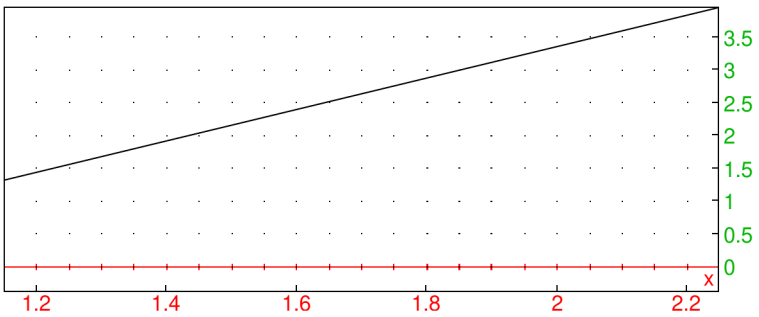
```
> G:=plotfunc(g(x),x):;
  A:=point(1.2,g(1.2)):;
```

If you want to draw the tangent at the point `A` to the graph `G`, enter:

```
> T:=tangent(G, A)
```

or:

```
> T:=tangent(G, 1.2)
```



For the equation of the tangent line, enter:

```
> equation(T)
```

$$y = 2.4x - 1.44$$

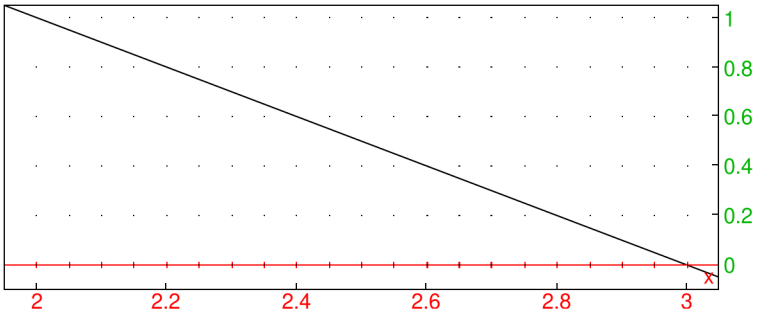
19.3.6 Plotting a line with a point and the slope

The `DrawSlp` command can draw a line given a point and a slope.

- `DrawSlp` takes three arguments: a, b and m, real numbers.
- `DrawSlp`(a, b, m) returns and draws the line through the point (a, b) with slope m.

Example

```
> DrawSlp(2,1,-1)
```



19.3.7 Intersection of a 2D graph with the axis

You can find the intersection of the graph $y = f(x)$ of a function with the axes using the commands covered so far.

- Finding the intersection of the graph with the y-axis is simply evaluating

  $$f(0),$$

  indeed the point with coordinates $(0, f(0))$ is the intersection point of the graph of f with the y-axis.

- Finding the intersection of the graph of f with the x-axis requires solving the equation $f(x) = 0$.
  - If $f(x)$ is polynomial-like, then you can find the the exact values of the abscissa of these points with `solve` (see Section 9.3.6, p. 184).

    ```
    > solve(f(x),x)
    ```

    returns the solution.

– Otherwise, you can find numeric approximations of these abscissa. First, look at the graph for an initial guess x_0 or a range with an intersection and then refine it with `fsolve` (see Section 23.3.2, p. 634).

`fsolve`$(f(x), x, x_0, method)$ returns a numeric approximation of a solution.

19.4 Area graphs

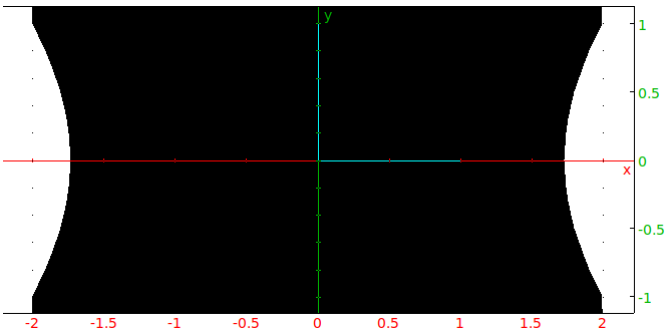
19.4.1 Graphing inequalities with two variables

The `plotinequation` or `inequationplot` command plots the region of the plane where given inequalities hold.

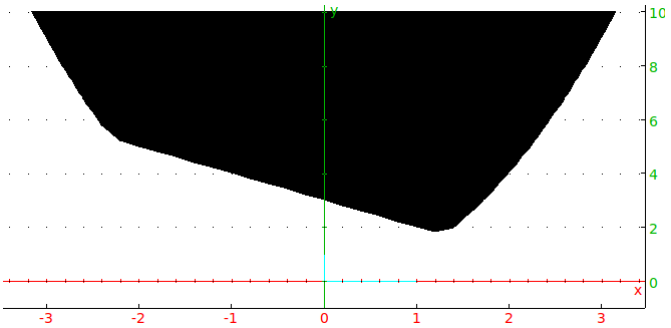
- `plotinequation` takes two arguments:
  – *ineqs*, a list of inequalities in two variables.
  – *vars*, a list of variables *var* or variables $var = a..b$ with their ranges of values. Note that if the ranges are not specified, XCAS takes the default values of `X-,X+,Y-,Y+` defined in the general graphic configuration (Cfg ▶ Graphic configuration, see Section 2.5.8, p. 17).
- `plotinequation`$(ineqs, vars)$ draws the points of the plane whose coordinates satisfy the inequalities *ineqs*.

Examples

```
> plotinequation(x^2-y^2<3,[x=-2..2,y=-2..2],xstep=0.1,ystep=0.1)
```




```
> plotinequation([x+y>3,x^2<y],[x-2..2,y=-1..10],xstep=0.2,ystep=0.2)
```



19.4.2 Computing the area under a curve

The `area` command approximates the area under a graph.

- `area` takes four arguments:

  - *expr*, an expression $f(x)$.
  - $var = a..b$, the variable with a range.
  - n, a positive integer.
  - *method*, the approximation method to use, which can be one of:
    * `trapezoid`
    * `left_rectangle`
    * `right_rectangle`
    * `middle_point`
    * `simpson`
    * `rombergt` (Romberg with the trapezoid method)
    * `rombergm` (Romberg with the midpoint method)
    * `gauss15` (The 15 point Gaussian quadrature)

- `area(`*expr*, $var = a..b$, n, *method*`)` returns an approximation to the area under the graph over the given interval, using the specified method with n subdivisions (or 2^n subdivisions for `rombergt`, `rombergm` and `gauss15`).

Examples

```
> area(x^2,x=0..1,8,trapezoid)
```

$$0.3359375$$

```
> area(x^2,x=0..1,8,rombergm)
```

$$0.333333333333$$

```
> area(x^2,x=0..1,3,gauss15)
```

$$0.333333333333$$

```
> area(x^2,x=0..1)
```

$$\frac{1}{3}$$

19.4.3 Graphing the area under a curve

The `plotarea` or `areaplot` command draws the area below a graph.

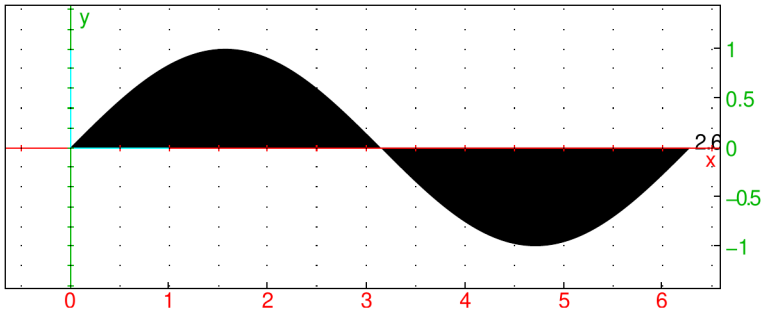
- `plotarea` takes two mandatory arguments and two optional arguments:
  - *expr*, an expression representing the function to graph.
  - $var = a..b$, the variable and the range of values.
  - Optionally, n.
  - Optionally, `method`, a method to approximate the region under the graph, which can be one of: `trapezoid`, `rectangle_left`, `rectangle_right` or `middle_point`.

- `plotarea(`*expr*, $var = a..b$`)` draws and shades the area between the graph of *expr* and the y-axis for $a < var < b$.

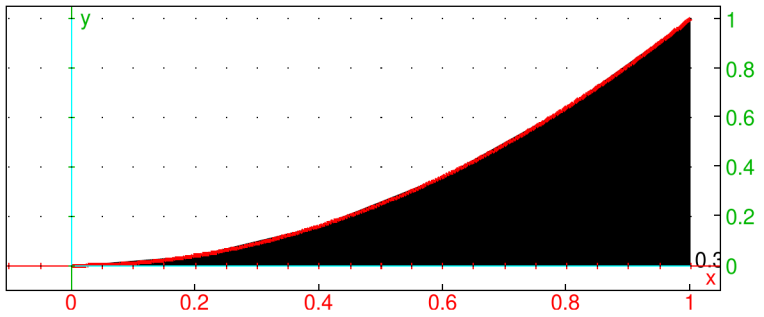
- `plotarea(`*expr*, *var*=*a*`..`*b* ⟨, *n*, *method*⟩`)` draws and shades the region used by the numeric approximation method *method* for area between the graph of *expr* and the y-axis for $a < var < b$, when $[a, b]$ is cut into n equal parts, along with the graph in red.

Examples

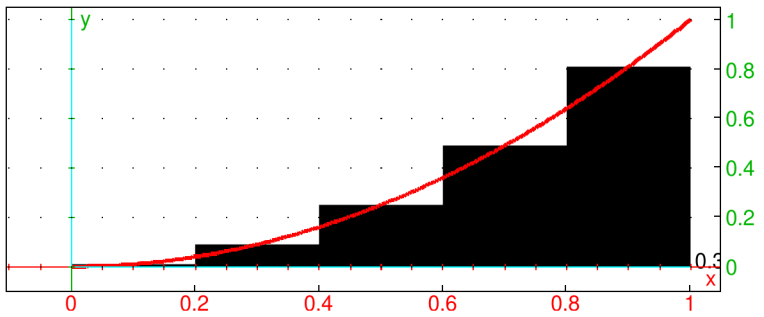
```
> plotarea(sin(x),x=0..2*pi)
```




```
> plotarea(x^2,x=0..1,5,trapezoid)
```




```
> plotarea((x^2,x=0..1,5,middle_point)
```



19.5 Contour and density graphs for surfaces

19.5.1 Contour lines

The `plotcontour` or `DrwCtour` or `contourplot` command draws contour lines for functions of two variables.

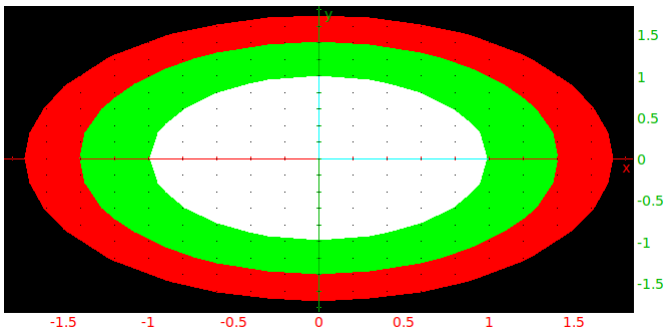
- `plotcontour` takes two mandatory arguments and one optional argument:
  - *expr*, an expression involving two variables.
  - *vars*, a list of the two variables.

– Optionally, *values*, a list of values of the contour lines to draw; *expr=value*, for *value* in *values* (by default, $[-10, -8, \ldots, 8, 10]$).

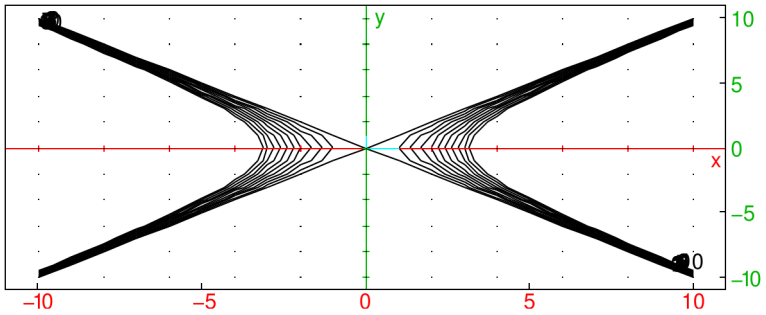
- `plotcontour(`*expr*, *vars* ⟨, *values*⟩`)` draws the contour lines *expr=value* for *value* in *values*.

Examples

```
> plotcontour(x^2+y^2,[x=-3..3,y=-3..3],[1,2,3],display=[green,red,black]+[filled$3])
```

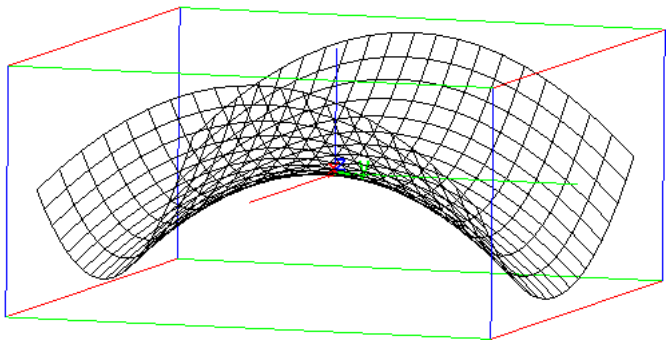



```
> plotcontour(x^2-y^2,[x,y])
```



If you want to draw the surface in 3D representation, use `plotfunc` (see Section 19.2.2, p. 465).

```
> plotfunc(x^2-y^2,[x,y])
```



19.5.2 2D graph of a surface with colors

The `plotdensity` or `densityplot` command draws the graph of a function of two variables in the plane where the values of z are represented by the rainbow colors.

- `plotdensity` takes two mandatory arguments and three optional arguments:
  – *expr*, an expression of two variables.
  – *vars*, a list of the variables and their ranges.

  - Optionally, `z=`*a*`..`*b*, the range of *z* to correspond to the full rainbow (by default, it is deduced from the minimum and maximum value of *expr* on the discretization.
  - Optionally, *xstep*, which can be `xstep=`*n* to specify the discretization step in the *x* direction.
  - Optionally, *ystep*, which can be `ystep=`*m* to specify the discretization step in the *y* direction.

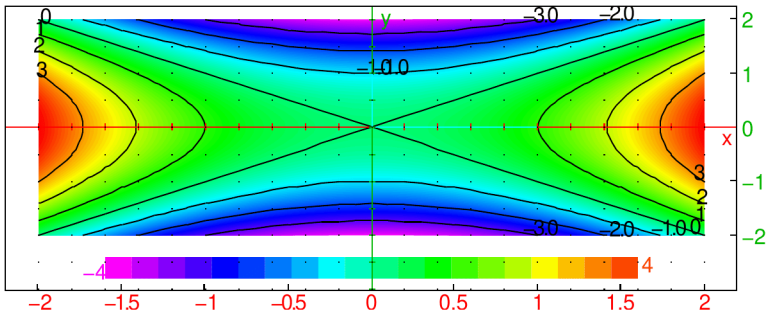
    Instead of *xstep* and `ystep`, you could use the option `nstep=`*n* to specify the number of points used to graph.

- `plotdensity(`*expr*, *vars* ⟨, `z=`*a*`..`*b*, *xstep*, *ystep*⟩`)` draws the graph of *expr* in the plane where the values of *z* are represented by the rainbow colors.

Remark. A rectangle representing the scale of colors will be displayed below the graph.

Example

```
> plotdensity(x^2-y^2,[x=-2..2,y=-2..2],xstep=0.1,ystep=0.1)
```



19.6 Implicit graphs

The `plotimplicit` or `implicitplot` command draws curves or surfaces defined by an implicit expression or equation. If the option `unfactored` is given as the last argument, the original expression is taken unmodified. Otherwise, the expression is normalized, then replaced by the factorization of the numerator of its normalization.

Each factor of the expression corresponds to a component of the implicit curve or surface. For each factor, Xcas tests if it is of total degree less or equal to 2, in which case `conic` or `quadric` is called. Otherwise the numeric implicit solver is called.

Optional step and ranges arguments may be passed to the numeric implicit solver, note that they are dismissed for each component that is a conic or a quadric.

19.6.1 2D implicit curve

- For an implicit plot in $\mathbb{R}^2$, `plotimplicit` takes three mandatory arguments and three optional arguments:
  - *expr*, an expression of two variables implicitly defining a curve by *expr=0*.
  - *vars*, a list of the two variables, optionally with their ranges *var*`=`*a*`..`*b*. If a range is not given, the ranges are determined by `WX-, WX+` and `WY-, WY+` in the graphical configuration (see Section 2.5.8, p. 17).
  - Optionally, *xstep*, which can be `xstep=`*n* to specify the discretization step in the *x* direction.
  - Optionally, *ystep*, which can be `ystep=`*m* to specify the discretization step in the *y* direction.
  - Optionally, `unfactored`.

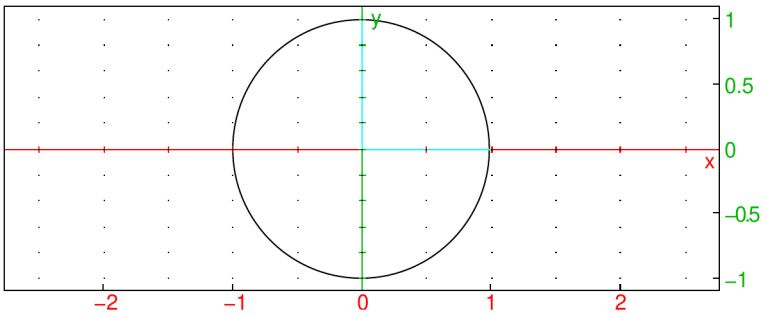
- `plotimplicit(`*expr*, *vars* ⟨, *xstep*, *ystep*, `unfactored` ⟩`)` draws the graphic representation of the curve defined by the implicit equation *expr=0* over the given ranges of the variables.

Example

```
> plotimplicit(x^2+y^2-1,x,y)
```

or:

```
> plotimplicit(x^2+y^2-1,x,y,unfactored)
```



19.6.2 3D implicit surface

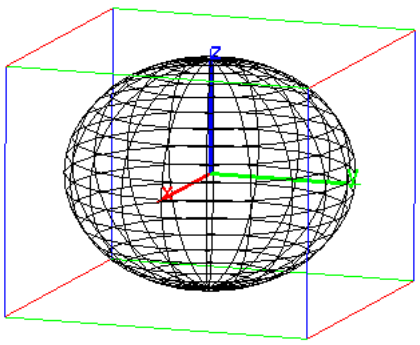
- For an implicit plot in $\mathbb{R}^3$, `plotimplicit` takes four mandatory arguments and three optional arguments:
  - *expr*, an expression of three variables implicitly defining a curve by *expr=0*.
  - *xvar*, *yvar* and *zvar*, the first, second and third variables,optionally with their ranges *var*=*a*`..`*b*.
  - Optionally, *xstep*, which can be `xstep=`*n* to specify the discretization step in the x direction.
  - Optionally, *ystep*, which can be `ystep=`*m* to specify the discretization step in the y direction.
  - Optionally, *zstep*, which can be `zstep=`*p* to specify the discretization step in the y direction.
  - Optionally, `unfactored`.
- `plotimplicit(`*expr*, *xvar*, *yvar*, *zvar* ⟨, *xstep*, *ystep*, *zstep*, `unfactored` ⟩`)` draws the graphic representation of the surface defined by the implicit equation *expr=0* over the given ranges of the variables.

Examples

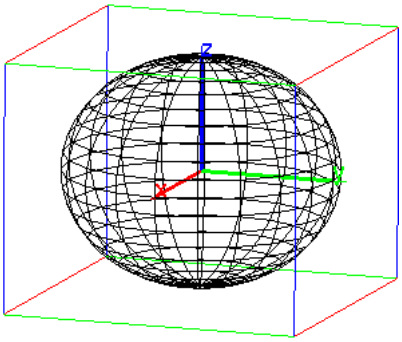
```
> plotimplicit(x^2+y^2+z^2-1,x,y,z,xstep=0.2,ystep=0.1,zstep=0.3)
```

or:

```
> plotimplicit(x^2+y^2+z^2-1,x,y,z,xstep=0.2,ystep=0.1,zstep=0.3,unfactored)
```



```
> plotimplicit(x^2+y^2+z^2-1,x=-1..1,y=-1..1,z=-1..1)
```



19.7 Parametric curves and surfaces

The `plotparam` or `paramplot` or `DrawParm` command draws parametric curves and surfaces.

19.7.1 2D parametric curve

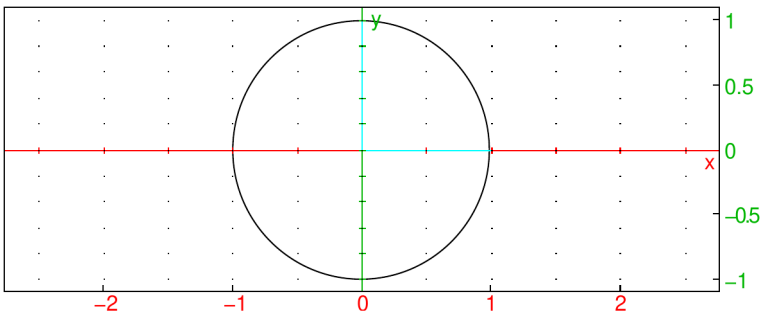
- To draw a parametric curve in $\mathbb{R}^2$, `plotparam` takes two mandatory and one optional argument:
  - *exprs*, a list of two real expressions or one complex expression involving the parameter.
  - *var*, the parameter, optionally with a range $var = a..b$. If no range is given, the values of `t-` and `t+` from the graphics configuration will be used (see Section 2.5.8, p. 17).
  - Optionally, `tstep=`n, to set the discretization step.
- `plotparam(`*exprs*, *var* ⟨, `tstep=`n ⟩`)` draws the parametric representation of the curve.

Examples

```
> plotparam(cos(x)+i*sin(x),x)
```

or:

```
> plotparam([cos(x),sin(x)],x)
```



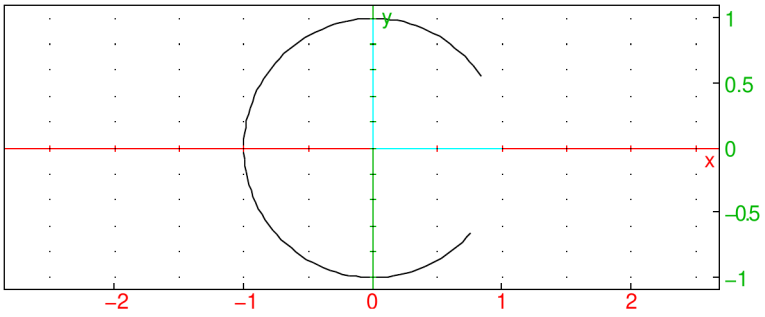

```
> plotparam(sin(t)+i*cos(t),t=-4..1)
```

or:

```
> plotparam(sin(x)+i*cos(x),x=-4..1)
```

or, with `t-=4,t+=1` in the graphic configuration:

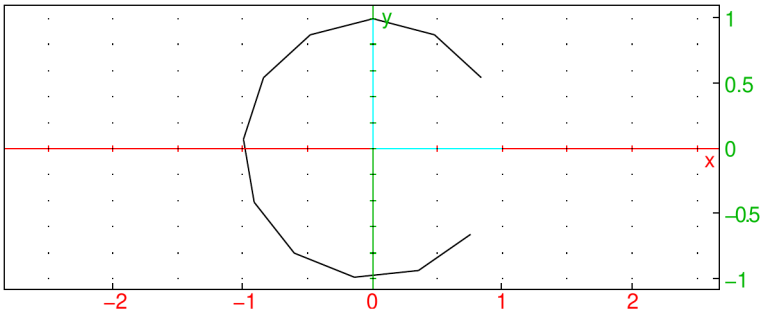
```
> plotparam(sin(t)+i*cos(t))
```




```
> plotparam(sin(t)+i*cos(t),t=-4..1,tstep=0.5)
```

or, with `t-=4,t+=1` in the graphic configuration:

```
> plotparam(sin(t)+i*cos(t),t,tstep=0.5)
```



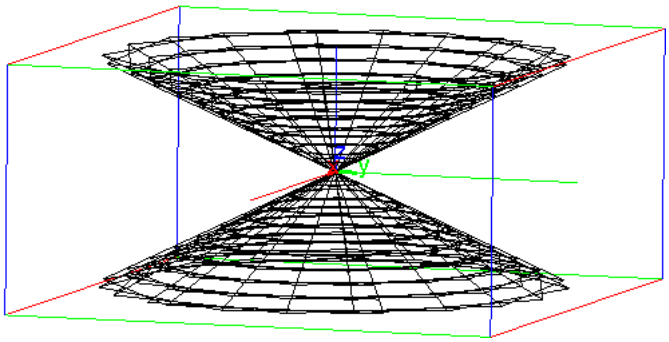
19.7.2 3D parametric surface

To draw a parametric surface in $\mathbb{R}^3$:

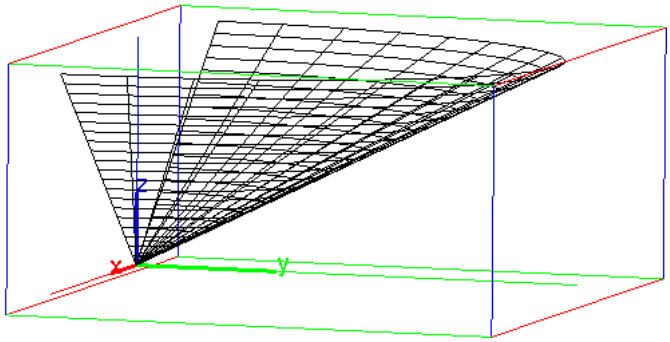
- `plotparam` takes two mandatory arguments and two optional arguments:
  - *exprs*, a list of three expressions involving two parameters.
  - *vars*, a list of the parameters, optionally with a range $var = a..b$.
  - Optionally, `ustep=`n, to set the discretization step of the first parameter.
  - Optionally, `vstep=`m, to set the discretization step of the second parameter.
- `plotparam(`*exprs*, *vars* ⟨, `ustep=`n, `vstep=`m ⟩`)` draws the parametric representation of the surface.

Examples

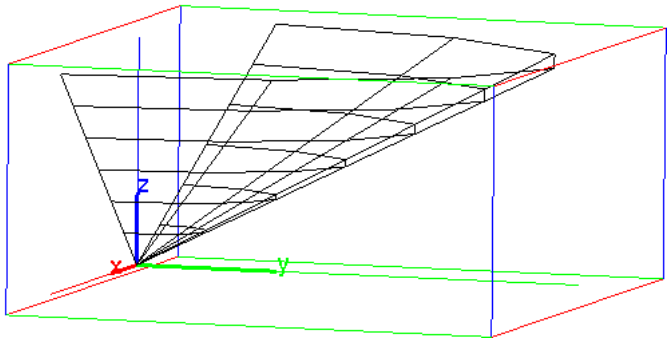
```
> plotparam([v*cos(u),v*sin(u),v],[u,v])
```



```
> plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3])
```




```
> plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3],ustep=0.5,vstep=0.5)
```



19.8 Plotting curves and sequences

19.8.1 Bezier curves

The Bezier curve with the control points $P_0, P_1, \ldots, P_n$ is the curve parameterized by

$$\sum_{j=0}^{n} \binom{n, j}{t}^{j} (1-t)^{n-j} P_j.$$

The `bezier` command plots Bezier curves.

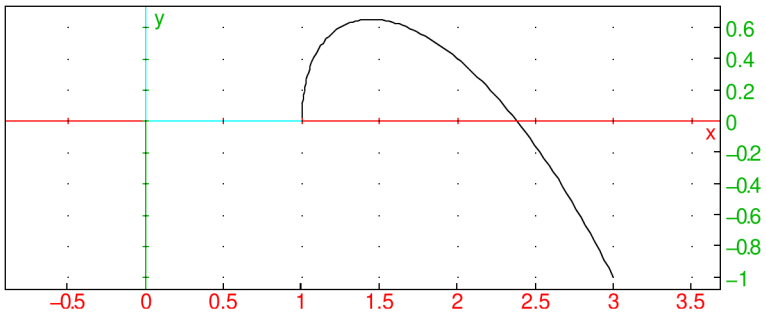
- `bezier` takes a sequence of arguments:

  - *controls*, a sequence of control points.
  - `plot`, the symbol.

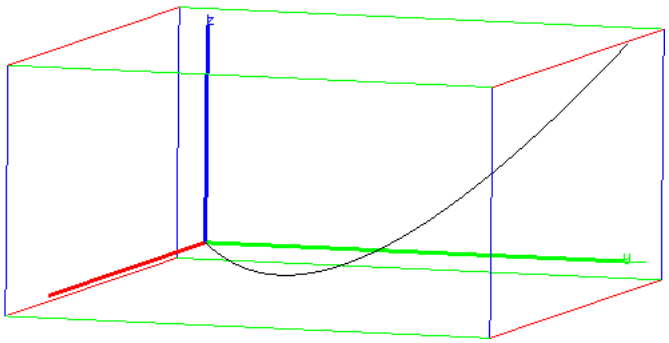
- `bezier(`*controls*`,plot)` plots the Bezier curve with the given control points.

Examples

```
> bezier(1,1+i,2+i,3-i,plot)
```




```
> bezier(point(0,0,0),point(1,1,0),point(0,1,1),plot)
```



To get the parameterization of the curve, use the `parameq` command (see Section 26.12.8, p. 724).

Examples

```
> parameq(bezier(1,1+i,2+i,3-i))
```

$$(1-t)^3+3t\,(1-t)^2\,(1+\mathrm{i})+3t^2\,(1-t)\,(2+\mathrm{i})+t^3\,(3-\mathrm{i})$$

```
> parameq(bezier(point([0,0,0]),point([1,1,0]),point([0,1,1])))
```

$$\left[2t\,(1-t)\,,2t\,(1-t)+t^2,t^2\right]$$

19.8.2 Curves in polar coordinates

The `plotpolar` or `polarplot` or `DrawPol` or `courbe_polaire` command draws a curve given in polar coordinates.

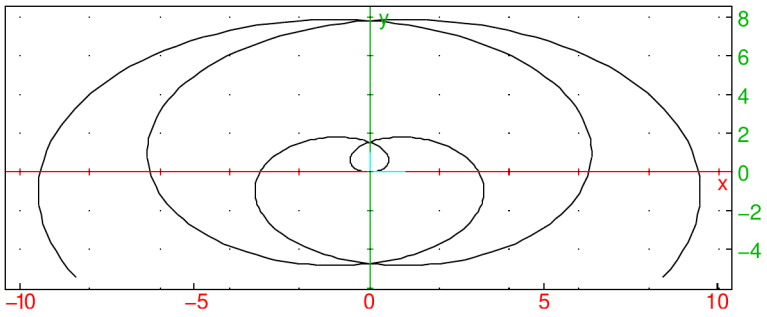
- `polarplot` takes two arguments:
  - *expr*, an expression involving a variable (which will represent the angle).
  - `var`, the variable. This can optionally include the range $var = a\,..\,b$.

– Optionally, `tstep=`n to specify the discretization.

`polarplot(`*expr*, *var* ⟨, `tstep=`n ⟩`)` draws the curve defined by $\rho = expr$ for $\theta = var$; in Cartesian coordinates that is the curve $(expr \cdot \cos(var), expr \cdot \sin(var))$.

Examples

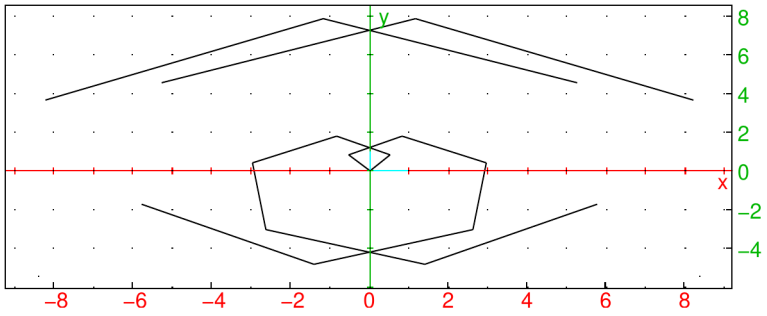
```
> plotpolar(t,t)
```




```
> plotpolar(t,t,tstep=1)
```

or:

```
> plotpolar(t,t=0..10,tstep=1)
```



19.8.3 Plotting solutions of differential equations

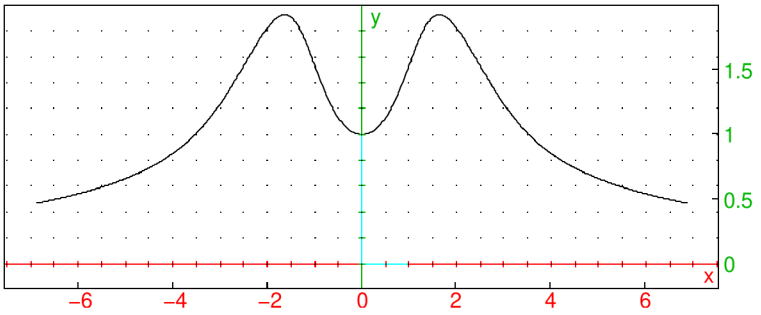
The `plotode` or `odeplot` command draws solutions of differential equations.

- `plotode` takes three mandatory arguments and one optional argument:
  - *expr* an expression depending on two or three variables, a time variable and one or two dependent variables.
  - *vars*, a list of the time variable and the dependent variable. The time variable can optionally have a range of values, such as $t = a..b$. The dependent variable can also be a vector of size two.
  - *init*, the initial values of the variables.
  - Optionally, for when there are two dependent variables, `plane`, the symbol, to draw the solution in the plane.
- `plotode(`*expr*, *vars*, *init*`)` draws the solution of the differential equation $y' = expr$ (where y is the dependent variable) passing through the initial point *init*.

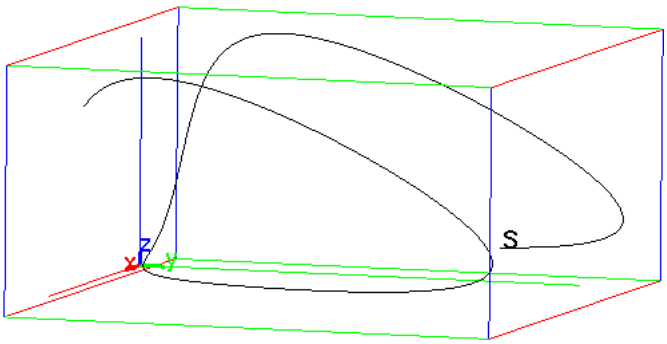
To compute the values of the solutions, see Section 23.4.1, p. 637.

Examples

```
> plotode(sin(t*y),[t,y],[0,1])
```

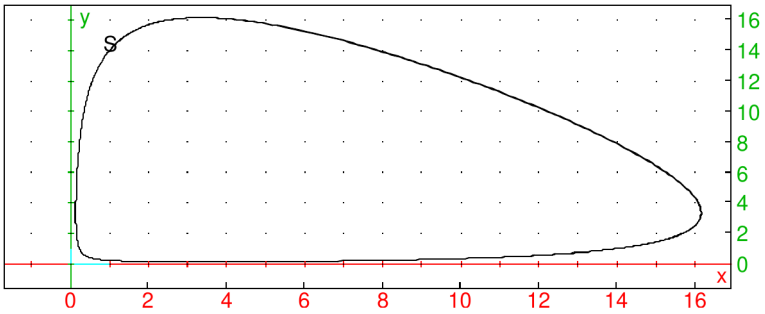



```
> S:=plotode([h-0.3*h*p,0.3*h*p-p],[t,h,p],[0,0.3,0.7])
```



Input for a 2D graph in the plane:

```
> S:=odeplot([h-0.3*h*p,0.3*h*p-p],[t,h,p],[0,0.3,0.7],plane)
```



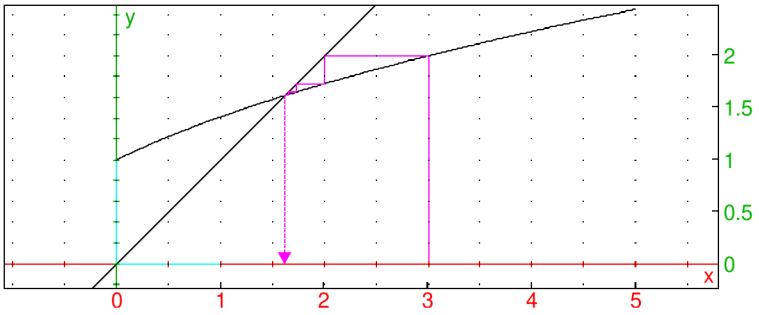
19.8.4 Graphing recurrent sequences

The `plotseq` or `seqplot` or `graphe_suite` command draws the process of finding the terms of a recurrent sequence.

- `plotseq` takes:
  - *expr*, an expression depending on a variable.
  - *var*=*a*, the variable and a beginning value. (If *var*=`x`, then the variable name can be omitted.) The *a* value can be replaced by a list of three elements, $[a, x_-, x_+]$ where $x_- \ldots x_+$ will be passed as the range of the variable for the graph computation.
  - *n*, the ending value of the variable.
- `plotseq(`*expr*, *var*=*a*, *n*`)` draws the line $y = x$, the graph of $y =$ *expr*, and the n first terms of the recurrent sequence defined by: $u_0 = a$, $u_n = f(u_{n-1})$ where f is the function determined by *expr*.

Example

```
> plotseq(sqrt(1+x),x=[3,0,5],5)
```



19.9 Plotting fields

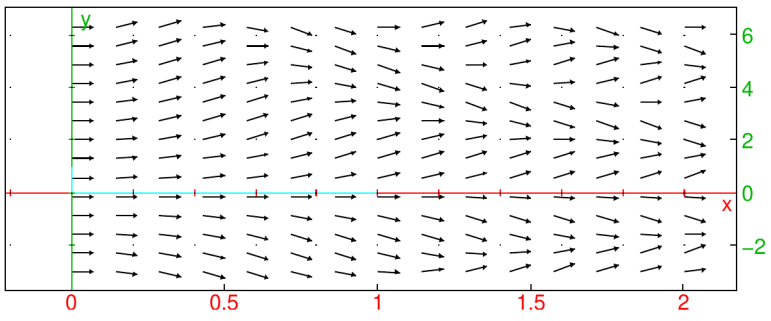
19.9.1 Tangent field

The `plotfield` or `fieldplot` command draws the tangent field of a differential equation or a vector field.

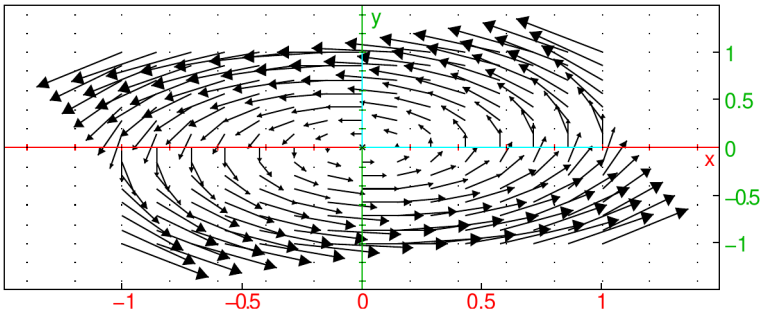
- To draw the tangent field of a differential equation, `plotfield` takes two mandatory arguments and one optional argument:
  - *expr*, an expression depending on two variables, a time variable and a dependent variable.
  - *vars*, a list of the two variables $[t, y]$, where t is the time variable and y is the dependent variable. The variables can optionally include their ranges; $[t = a..b, y = c..d]$.
  - Optionally, `ystep=`n to specify the discretization.
- `plotfield(`*expr*, *vars* ⟨`, ystep=`n⟩`)` draws the tangent field of the differential equation $y' = f(t, y)$.
- To draw a vector field `plotfield` takes two mandatory arguments and two optional arguments:
  - V, a list of two expressions involving two variables.
  - *vars*, a list of the two variables. The variables can optionally include their ranges *var*=$a..b$.
  - Optionally, `xstep=`n to specify the discretization of the first variable.
  - Optionally, `ystep=`m to specify the discretization of the second variable.
- `plotfield(`V, *vars* ⟨`, xstep=`n, `ystep=`m⟩`)` draws the vector field given by V.

Example

```
> plotfield(4*sin(t*y),[t=0..2,y=-3..7])
```



```
> plotfield(5*[-y,x],[x=-1..1,y=-1..1])
```



19.9.2 Interactive plotting of solutions of a differential equation

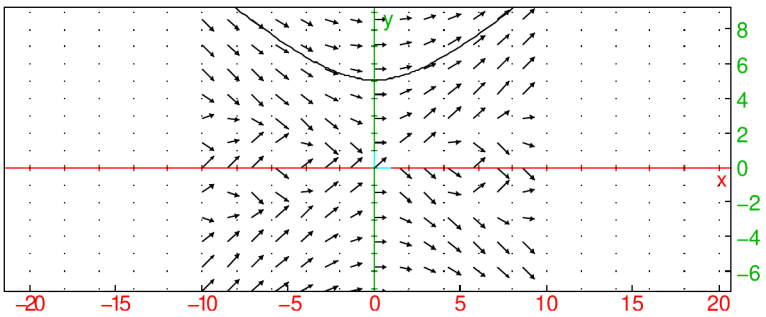
The `interactive_plotode` or `interactive_odeplot` command draws interactive tangent fields of differential equations.

- `interactive_plotode` takes two arguments:
  - *expr* an expression depending on two or three variables, a time variable and one or two dependent variables.
  - *vars*, a list of the time variable and the dependent variable.
- `interactive_plotode`(*expr*, *vars*) draws the tangent field of the differential equation $y' = expr$ (where y is the dependent variable) in a new window. In this window, one can click on a point to get the plot of the solution of $y' = expr$ passing through this point.

  You can further click to display several solutions. To stop, press the Esc key.

Example

```
> interactive_plotode(sin(t/y),[t,y])
```



Solutions of the differential equation can be plotted by clicking on an initial point in the graphic screen (see Section 19.1.1, p. 454).

19.10 Animated graphs

Xcas can display animated 2D, 3D or "4D" graphs. This is done first by computing a sequence of graphic objects, then after completion, by displaying the sequence in a loop. To stop or start the animation, click on the button ▶| (to the left of Menu).

19.10.1 Animation of a 2D graph

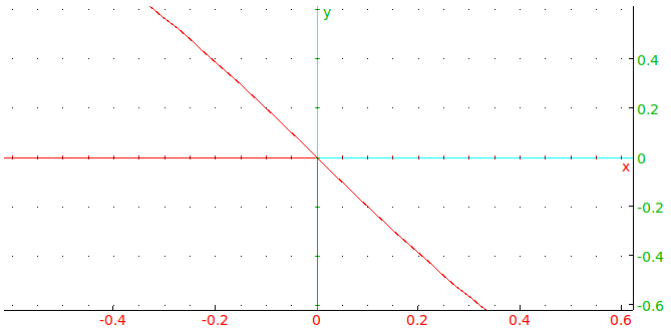
The `animate` command creates 2D animations using graphs of functions depending on a parameter. (See also Section 19.2.3, p. 468.)

- `animate` takes three mandatory arguments and two optional arguments:
  - *expr*, an expression involving two variables, one of which will be regarded as the parameter.
  - *var* the name of the (non-parameter) variable in the expression, which can also specify a range of values $var = a..b$.
  - *param* the name of the parameter, which can also specify a range of values.
  - Optionally, `frames=`n, where n is an integer specifying the number of frames.
  - Optionally, *opt*, which can be `xstep=`n to specify the discretization step or `nstep=`n to specify the number of points used to graph.
- `animate(`*expr*, *var*, *param*, `frames=`n ⟨, *opt*⟩`)` draws an animation consisting of graph of the function as the parameter varies.

Examples

```
> animate(sin(a*x),x=-pi..pi,a=-2..2,frames=10,color=red)
```

The output is an animation beginning with:



19.10.2 Animation of a 3D graph

The `animate3d` command creates 3D animations using graphs of functions depending on a parameter. (See also Section 19.2.2, p. 465.)

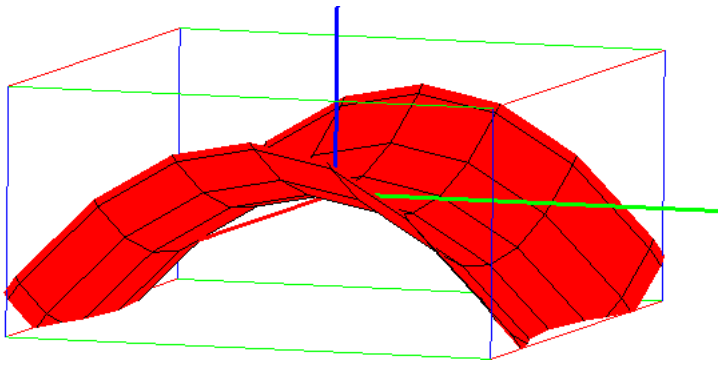
- `animate` takes three mandatory arguments and two optional arguments:
  - *expr*, an expression involving three variables, one of which will be regarded as the parameter.
  - *vars* a list of the the (non-parameter) variables in the expression, which can also specify ranges of values $var = a..b$.
  - *param* the name of the parameter, which can also specify a range of values.
  - Optionally, `frames=`n, where n is an integer specifying the number of frames.
  - Optionally, *xstep*, which can be `xstep=`n to specify the discretization step in the x direction.
  - Optionally, *ystep*, which can be `ystep=`m to specify the discretization step in the y direction.

    Instead of *xstep* and `ystep`, you could use the option `nstep=`n to specify the number of points used to graph.
- `animate3d(`*expr*, *vars*, *param*, `frames=`n ⟨, *opt*, `xstep=`n, `ystep=`m ⟩`)` draws an animation consisting of graph of the function as the parameter varies.

Example

```
> animate3d(x^2+a*y^2,[x=-2..2,y=-2..2],a=-2..2,frames=10,display=red+filled)
```

The output is an animation beginning with:



19.10.3 Animation of a sequence of graphic objects

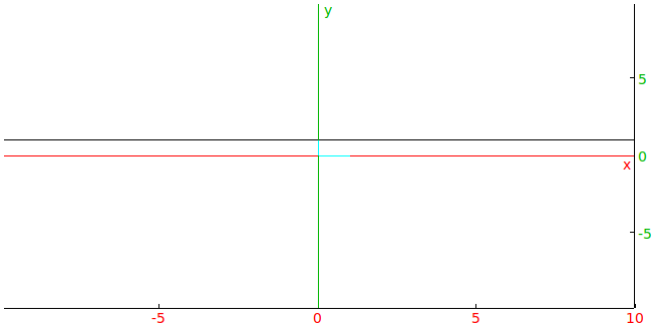
The `animation` command creates animations using sequences of graphic objects, which can be graphs (see Section 19.2, p. 464) or not (see chapters 26 and 27). The sequence of objects depends most of the time on a parameter and is defined using the `seq` command but it is not mandatory. To define a sequence of graphic objects with `seq`, enter the definition of the graphic object (depending on the parameter), the parameter name, its minimum value, its maximum value maximum and optionally a step value.

- `animation` takes:
  - *objs*, a sequence of graphic objects.
- `animation(`*objs*`)` draws an animation consisting of the sequence of objects.

Examples

In each example below, only the first frame of the resulting animation is shown.

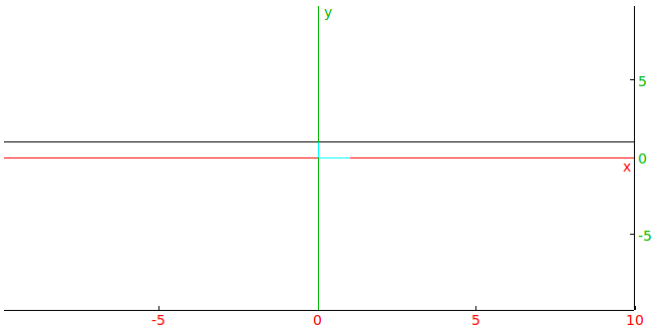
```
> animation(seq(plotfunc(cos(a*x),x),a,0,10))
```



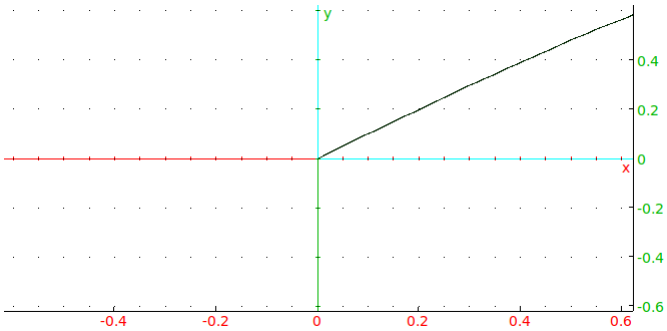

```
> animation(seq(plotfunc(cos(a*x),x),a,0,10,0.5))
```

or:

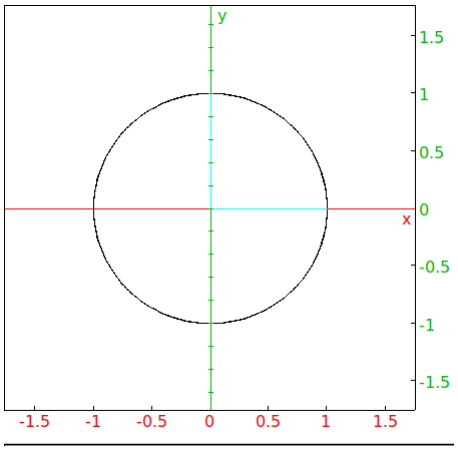
```
> animation(seq(plotfunc(cos(a*x),x),a=0..10,0.5))
```



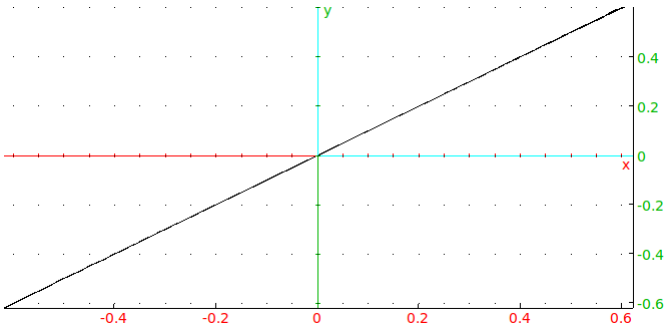

```
> animation(seq(plotfunc([cos(a*x),sin(a*x)],x=0..2*pi/a), a,1,10))
```



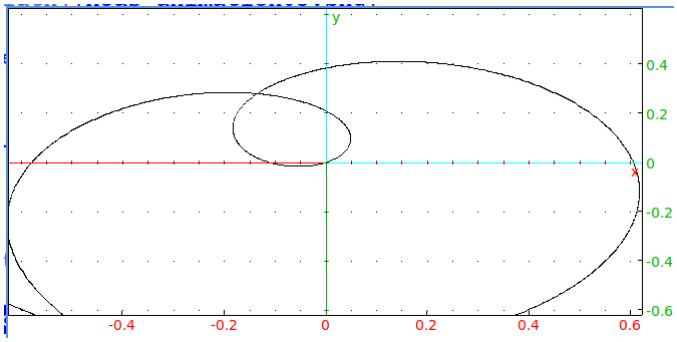

```
> animation(seq(plotparam([cos(a*t),sin(a*t)],t=0..2*pi),a,1,10))
```



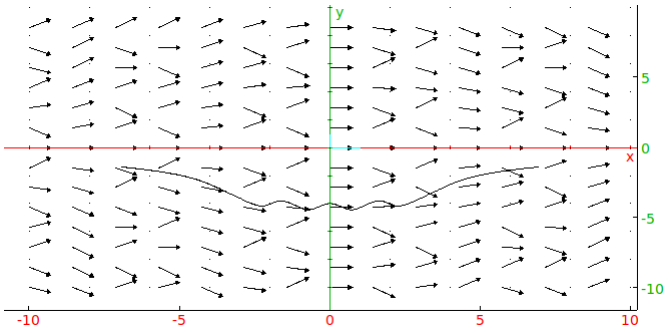

```
> animation(seq(plotparam([sin(t),sin(a*t)],t,0,2*pi,tstep=0.01),a,1,10))
```



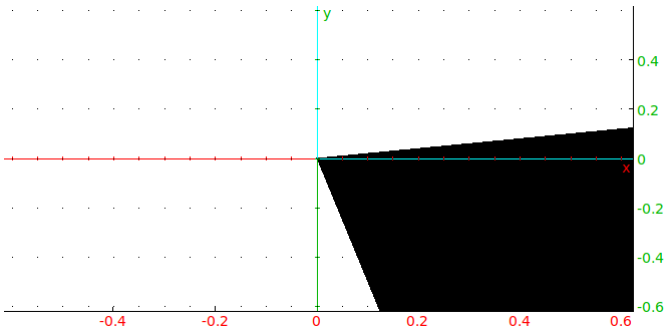

```
> animation(seq(plotpolar(1-a*0.01*t^2, t,0,5*pi,tstep=0.01),a,1,10))
```



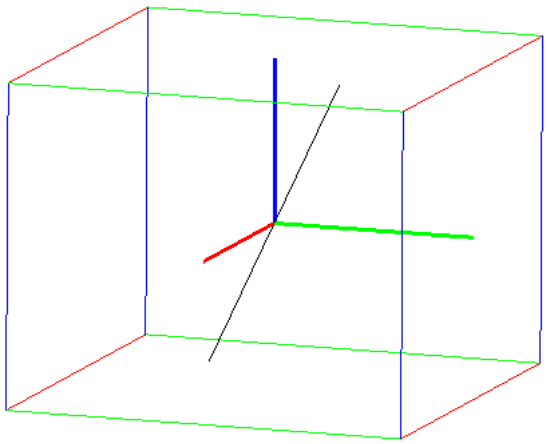

```
> plotfield(sin(x*y),[x,y]);
  animation(seq(plotode(sin(x*y),[x,y],[0,a]),a,-4,4,0.5))
```



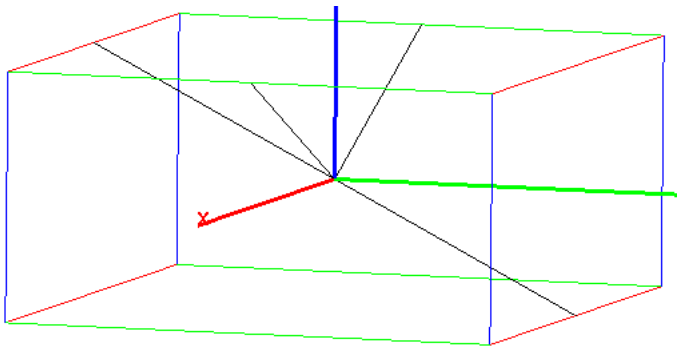

```
> animation(seq(display(square(0,1+i*a),filled),a,-5,5))
```



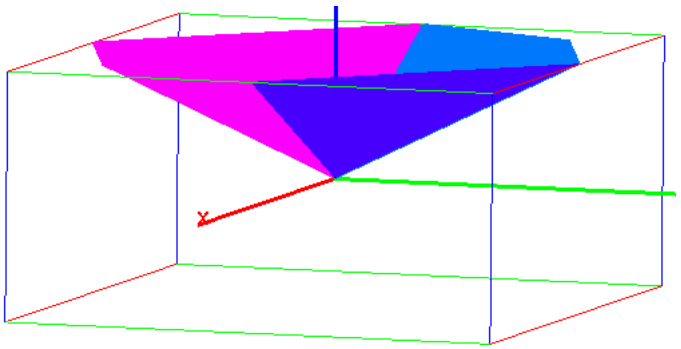

```
> animation(seq(line([0,0,0],[1,1,a]),a,-5,5))
```



```
> animation(seq(plotfunc(x^2-y^a,[x,y]),a=1..3))
```



```
> animation(seq(plotfunc((x+i*y)^a,[x,y],display=filled),a=1..10)
```

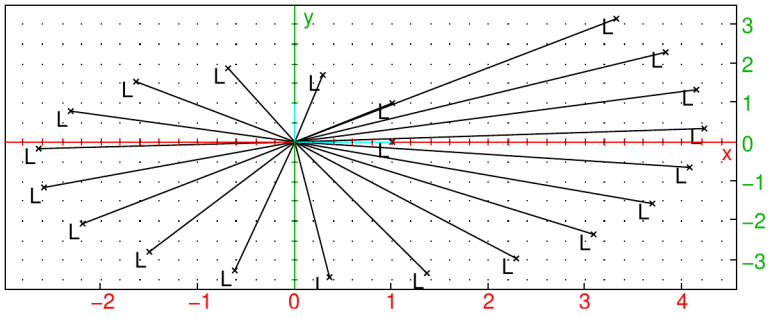


Remark. You can also define the sequence with a program. For example if you want to draw the segments of length $1, \sqrt{2} \ldots \sqrt{20}$ constructed with a right triangle of side 1 and the previous segment (note that there is a `c:=evalf(..)` statement to force approximate evaluation otherwise the computing time would be too long):

```
seg(n):={
  local a,b,c,j,aa,bb,L;
  a:=1;
  b:=1;
  L:=[point(1)];
  for (j:=1;j<=n;j++) {
    L:=append(L,point(a+i*b));
    c:=evalf(sqrt(a^2+b^2));
    aa:=a;
    bb:=b;
    a:=aa-bb/c;
    b:=bb+aa/c;
  }
  return L;
};
```

then:

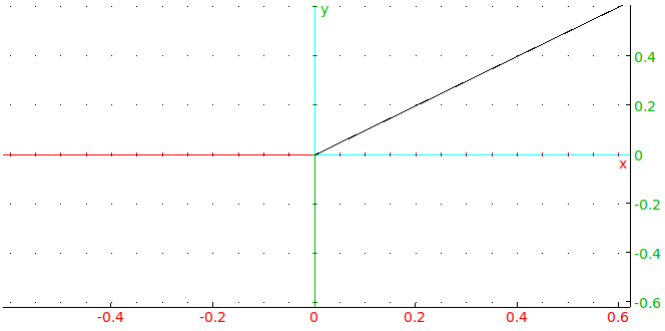
```
> L:=seg(20); s:=segment(0,L[k])$(k=0..20)
```



then:

```
> animation(s)
```

The output is an animation displaying the segments one at a time, beginning with:



20 Statistics

20.1 One variable statistics

XCAS has several functions to perform statistics. Data is typically given as a list of numbers, such as `A:=[0,1,2,3,4,5,6,7,8,9,10,11]`. This particular list will be used in several examples. Statistics on matrices are discussed in Section 14.5.2, p. 353.

20.1.1 Mean

Recall that the mean of a list $x_1, \ldots, x_n$ is simply their numeric average $(x_1 + \cdots + x_n)/n$. The `mean` command finds the mean of a list.

- `mean` takes one mandatory argument and one optional argument:
  - L, a list or matrix of numbers.
  - W, a list or matrix of weights, the same size as L.
- `mean(`$L\,\langle,W\rangle$`)` returns the mean of the list or a list with the means of the columns of the matrix.

Examples

```
> mean([1,2,3,4])
```

$$\frac{5}{2}$$

since $(1+2+3+4)/4 = 5/2$.

```
> mean([[1,2,3],[5,6,7]])
```

$$[3, 4, 5]$$

since $(1+5)/2 = 3$, $(2+6)/2 = 4$ and $(3+7)/2 = 5$.

```
> mean([2,4,6,8],[2,2,3,3])
```

$$\frac{27}{5}$$

since $(2\cdot 2 + 4\cdot 2 + 6\cdot 3 + 8\cdot 3)/(2+2+3+3) = 27/5$.

```
> mean([[1,2],[3,4]],[[1,2],[2,1]])
```

$$\left[\frac{7}{3}, \frac{8}{3}\right]$$

since $(1\cdot 1 + 3\cdot 2)/(1+2) = 7/3$ and $(2\cdot 2 + 4\cdot 1)/(2+1) = 8/3$.

20.1.2 Variance

The variance of a list of numbers measures how close the numbers are to their mean by finding the average of the squares of the differences between the numbers and the mean; specifically, given a list of numbers $[x_1, \ldots, x_n]$ with mean $\mu = (x_1 + \cdots + x_n)/n$, the variance is

$$\frac{(x_1 - \mu)^2 + \cdots + (x_n - \mu)^2}{n}.$$

The squares help ensure that the numbers above the mean and those below the mean do not cancel out. The `variance` command computes the variance.

- `variance` takes one mandatory argument and one optional argument:
  - L, a list or matrix of numbers.
  - W, a list or matrix of weights, the same size as L.
- `variance(`L $\langle, W\rangle$`)` returns the variance of the list or a list with the variances of the columns of the matrix.

Examples

```
> variance([1,2,3,4])
```

$$\frac{5}{4}$$

```
> variance([[1,2,3],[5,6,7]])
```

$$[4, 4, 4]$$

```
> variance([2,4,6,8],[2,2,3,3])
```

$$\frac{121}{25}$$

```
> variance([[1,2],[3,4]],[[1,2],[2,1]])
```

$$\left[\frac{8}{9}, \frac{8}{9}\right]$$

20.1.3 Standard deviation

Standard deviation is potentially better than variance to measure how close numbers are to their mean. The standard deviation is the square root of the variance; for example, the list $[1, 2, 3, 4]$ has mean $5/2$, and so the standard deviation will be $2\sqrt{5}/4$, since

$$\sqrt{\frac{(1 - 5/2)^2 + (2 - 5/2)^2 + (3 - 5/2)^2 + (4 - 5/2)^2}{4}} = \frac{2\sqrt{5}}{4}.$$

Note that if the list of numbers have units, then the standard deviation will have the same unit. The `stddev` command finds the standard deviation.

- `stddev` takes one mandatory argument and one optional argument:
  - L, a list or matrix of numbers.
  - W, a list or matrix of weights, the same size as L.
- `stddev(`L $\langle, W\rangle$`)` returns the standard deviation of the list or a list with the standard deviations of the columns of the matrix.

Examples

```
> stddev([1,2,3,4])
```

$$\frac{\sqrt{5}}{2}$$

```
> stddev([1,2,3],[2,1,1])
```

$$\frac{4}{16}\sqrt{11}$$

```
> stddev([[1,2],[3,6]])
```

$$[1, 2]$$

20.1.4 Population standard deviation

Given a large population, rather than collecting all of the numbers it might be more feasible to get a smaller collection of numbers and try to extrapolate from that. For example, to get information about the ages of a large population, you might get the ages of a sample of 100 of the people and work with that.

If a list of numbers is a sample of data from a larger population, then the mean of the sample can be used to estimate the mean of the population. The standard deviation uses the mean to find the standard deviation of the sample, but since the mean of the sample is only an approximation to the mean of the entire population, the standard deviation of the sample does not provide an optimal estimate of the standard deviation of the population. An unbiased estimate of the standard deviation of the entire population is given by the population standard deviation; given a list $L = [x_1, \ldots, x_n]$ with mean μ, the population standard deviation is

$$s = \sqrt{\frac{(x_1 - \mu)^2 + \cdots + (x_n - \mu)^2}{n - 1}}.$$

Note that

$$s^2 = \frac{n}{n-1}\sigma^2.$$

where σ is the standard deviation of the sample.

The `stddevp` command finds the standard deviation.

`stdDev` is a synonym for `stddevp`, for TI compatibility. There is no population variance function; if needed, it can be computed by squaring the `stddevp` function.

- `stddevp` takes one mandatory argument and one optional argument:
  - L, a list or matrix of numbers.
  - W, a list or matrix of weights, the same size as L.
- `stddevp`$(L\,\langle, W\rangle)$ returns the population standard deviation of the list or a list with the population standard deviations of the columns of the matrix.

Examples

```
> stddev([1,2,3,4])
```

$$\frac{\sqrt{5}}{2}$$

while:

```
> stddevp([1,2,3,4])
```

$$\frac{\sqrt{15}}{3}$$

```
> A:=[0,1,2,3,4,5,6,7,8,9,10,11]:;
  stddevp(A,A)
```

$$\frac{\sqrt{66}}{3}$$

20.1.5 Median

Although the average of a list of numbers typically means the mean, there are other notions of "average". Another such notion is the median; the median of a list of numbers is the middle number when they are listed in numeric order. For example, the median of the list $[1, 2, 5, 7, 20]$ is simply 5. If the length of a list of numbers is even, the median is the mean of the two middle numbers; for example, the median of $[1, 2, 5, 7, 20, 21]$ is $(5 + 7)/2 = 6$.

The `median` function finds the median of a list.

- `median` takes one mandatory argument and one optional argument:
  - L, a list of numbers.
  - Optionally, W, a list of positive integers for weights, where the weight of number represents how many times it is counted in a list.
- `median(`$L\,\langle, W\rangle$`)` returns the median of the list.

Examples

```
> median([1,2,5,7,20])
```

$$5$$

```
> median([1,2,5,7,20],[5,3,2,1,2])
```

$$2$$

since the median of $1, 1, 1, 1, 1, 2, 2, 2, 5, 5, 7, 20, 20$ is 2.

20.1.6 Quartiles

Recall that the quartiles of a list of numbers divide it into four equal parts; the first quartile is the number q_1 such that one-fourth of the list numbers fall below q_1; i.e., the median of that part of the list which fall at or below the list median. The second quartiles is the number q_2 such that half of the list numbers fall at or below q_2; more specifically, the median of the list. And of course the third quartile is the number q_3 such that three-fourths of the list numbers fall at or below q_3.

The function `quartiles` finds the minimum of a list, the first quartile, the second quartile, the third quartile and the maximum of the list.

- `quartiles` takes one mandatory argument and one optional argument:
  - L, a list of numbers.
  - Optionally, W, a list of weights.

- `quartiles(`L $\langle, W\rangle$`)` returns a column vector consisting of the minimum, first second and third quartile, and the maximum of L.

The `min`, `quartile1`, `median`, `quartile3` and `max` commands find the individual entries of this list.

Example

```
> A:=[0,1,2,3,4,5,6,7,8,9,10,11];
  quartiles(A)
```

$$\begin{bmatrix} 0.0 \\ 2.0 \\ 5.0 \\ 8.0 \\ 11.0 \end{bmatrix}$$

```
> min(A),quartile1(A),median(A),quartile3(A),max(A)
```

$$0, 2.0, 5.0, 8.0, 11$$

```
> quartiles(A,A)
```

$$[0, 6, 8, 10, 11]$$

20.1.7 Quantiles

Similar to quartiles, a quantile of a list is the number q such that a given fraction of the list numbers fall at or below q. The first quartile, for example, is the quantile with the fraction 0.25.

The `quantile` command finds quantiles.

- `quantile` takes two mandatory arguments and one optional argument:
  - L, a list of numbers.
  - Optionally, W, a list of weights.
  - p, a number between 0 and 1.

  `quantile(`L, p`)` returns the pth quantile of L.

Examples

```
> A:=[0,1,2,3,4,5,6,7,8,9,10,11]:;
  quantile(A,0.1)
```

$$1.0$$

```
> quantile(A,A,0.25)
```

$$6$$

20.1.8 Box-and-whisker diagrams

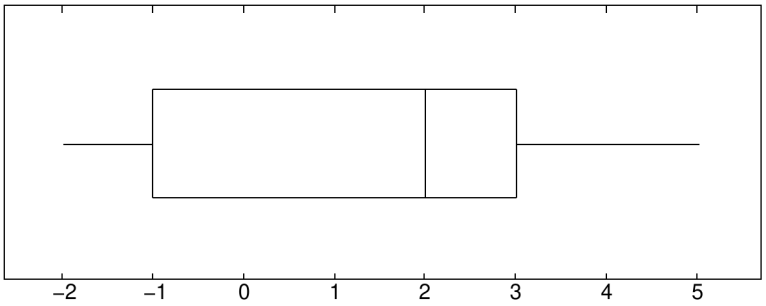
A box-and-whisker diagram is a graphical view of the quartiles of a list of numbers. The diagram consists of a line segment from the the minimum of the list to the first quartile, leading to a rectangle from the first quartile to the third quartile, followed by a line segment from the third quartile to the maximum of the list. The rectangle contains a vertical segment indicating the median.

The `boxwhisker` or `mustache` or `boxplot` command creates boxwhisker(s) for given list(s).

- `boxwhisker` takes one mandatory argument and four optional arguments:
  - L, a list or matrix of real numbers.
  - Optionally, `x=`*xmin*`..`*xmax* or `y=`*ymin*`..`*ymax*, which spreads several boxwhiskers horizontally (with `x`) or vertically (with `y`, which is the default).
  - Optionally, `color=`*c*, where *c* is a color or a list of colors specifying the fill color for each boxwhisker (if *c* is a single color, then all boxwhiskers are colored with that color). The list may be empty, in which case light grey color is used. By default, no fill color is specified and boxwhiskers are drawn only with lines.
  - Optionally, `legend=`*d*, where *d* is a list of strings which are drawn next to boxwhiskers, in the given order.
  - Optionally, `axes`, the symbol specifying that the usual axes should be drawn.
- `boxwhisker(`L ⟨, *options*⟩`)` draws a boxwhisker for the list L or one for each column of the matrix L, aligning them vertically (topmost is the first) or horizontally (leftmost is the first).

Examples

```
> boxwhisker([-1,1,2,2.2,3,4,-2,5])
```

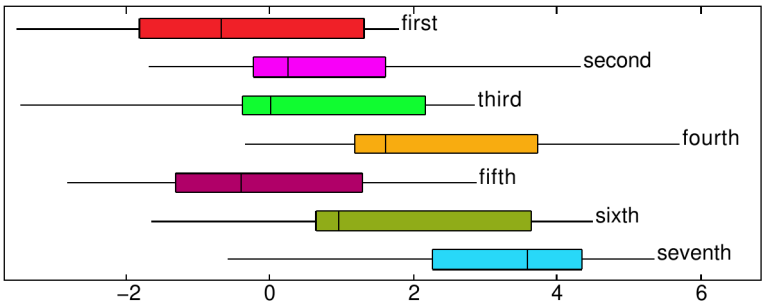


To generate some normally distributed data, enter:

```
> L:=tran([seq(randvector(10,randvar(normal,mean=ln(j),stddev=2)),j=1..10)]):;
```

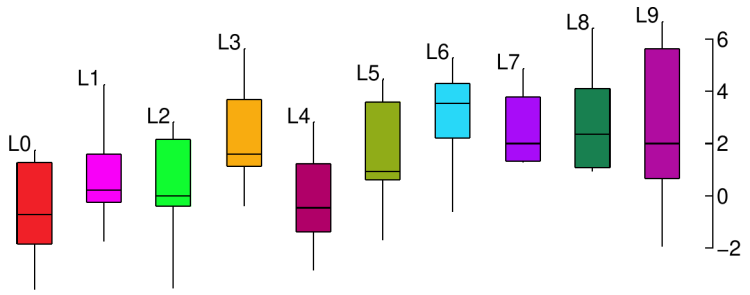
Vertical alignment (the default, suitable for a small number of boxwhiskers):

```
> c:=colormap("discrete","vivid"):;
  names:=["first","second","third","fourth","fifth","sixth","seventh"]:;
  boxplot(tran(col(L,0..6)),color=c,legend=names)
```



Horizontal alignment:

```
> boxplot(L,x=-5..5,color=c,legend=zip(concat,["L"$10],apply(cat,range(0,10))))
```



20.1.9 Classes

The `classes` command groups a collection of numbers into intervals.

- `classes` takes two or three arguments:
  - L, a list or matrix of numbers.
  - Optionally, a and b, numbers. (By default, these will be `class_min` and `class_size` from the graphics configuration, see Section 2.5.8, p. 17, which themselves default to 0 and 1.)

    or:
  - Optionally, a and M, where a is the start of the beginning interval and M consists of the midpoints of the intervals.

    or:
  - Optionally, I, a list of intervals to use.
- `classes(`L, a, b`)` returns the list `[[`$a..a+b, n_1$`],[`$a+b..a+2b, n_2$`],...]` where each number in L is in one of the intervals $[a+kb, a+(k+1)b)$ and n_k is how many numbers from L are in the corresponding interval.
- `classes(`L, a, M`)` returns a similar list, but instead of `[[`$a..a+b, n_1$`],[`$a+b..a+2b, n_2$`],...]`, the intervals are determined by a and the list of midpoints M.
- `classes(`L, I`)` returns a similar list, but instead of `[[`$a..a+b, n_1$`],[`$a+b..a+2b, n_2$`],...]`, the intervals are given by I. In this case, not every element of L is necessarily in an interval.

Examples

```
> classes([0,0.5,1,1.5,2,2.5,3,3.5,4],0,2)
```

$$\begin{bmatrix} 0.0\ldots2.0 & 4 \\ 2.0\ldots4.0 & 4 \\ 4.0\ldots6.0 & 1 \end{bmatrix}$$

```
> classes([0,0.5,1,1.5,2,2.5,3,3.5,4],-1,2)
```

$$\begin{bmatrix} -1.0\ldots1.0 & 2 \\ 1.0\ldots3.0 & 4 \\ 3.0\ldots5.0 & 3 \end{bmatrix}$$

```
> classes([0,0.5,1,1.5,2,2.5,3,3.5,4],1,[1,3,5])
```

$$\begin{bmatrix} 0.0\ldots2.0 & 4 \\ 2.0\ldots4.0 & 4 \\ 4.0\ldots6.0 & 1 \end{bmatrix}$$

```
> classes([0,0.5,1,1.5,2,2.5,3,3.5,4],[1..3,3..6])
```

$$\begin{bmatrix} 1\ldots3 & 4 \\ 3\ldots6 & 3 \end{bmatrix}$$

20.1.10 Histograms

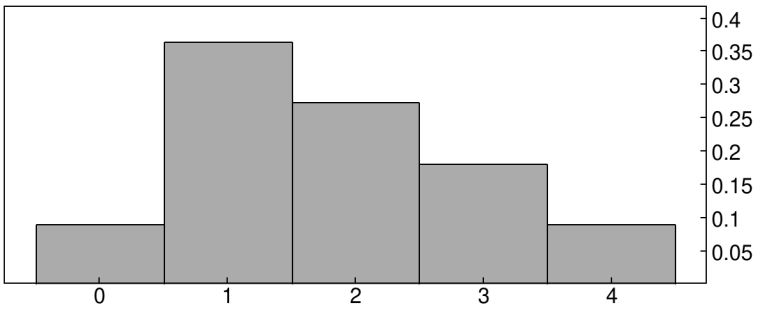
Given a list of intervals and a number of points in each interval, such as is given by the output of the `classes` command (see Section 20.1.9, p. 502), a histogram is a graph consisting of a box over each interval, where the height of each box is proportional to the number of points and the total area of the boxes is 1. The `histogram` or `histogramme` command draws histograms. Data does not have to be sorted.

With sorted data:

- `histogram` takes one mandatory argument and one optional argument:
  - L, a list whose elements are lists of a range $a..b$ and a positive integer.
  - Optionally, `color=`*col*, where *col* is a color object (see Section 19.1.3, p. 458).
- `histogram(`L $\langle$`,color=`*col*$\rangle$`)` draws a histogram for the data. If *col* is provided, then the fill color is changed from grey to *col*.

Example

```
> histogram([[0,1],[1,4],[2,3],[3,2],[4,1]])
```

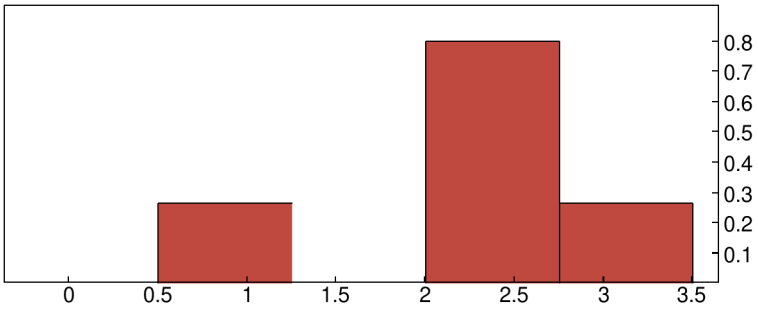


With unsorted data:

- `histogram` takes one mandatory argument and two optional arguments.
  - L, a list of numbers.
  - Optionally, a and b, numbers.
  - Optionally, `color=`*col*, where *col* is a color object (see Section 19.1.3, p. 458).
- `histogram(`L $\langle, a, b\rangle$ $\langle$`,color=`*col*$\rangle$`)` returns the histogram for `classes(`L $\langle, a, b\rangle$`)` (see Section 20.1.9, p. 502). If *col* is given, then the fill color is changed from grey to *col*.

Example

```
> histogram([1,2,2.5,2.5,3],0.5,0.75,color=brown)
```



20.1.11 Accumulating terms

The `accumulate_head_tail` command replaces the first terms of a list by their sum and the last terms of a list by their sum.

- `accumulate_head_tail` takes three arguments:
  - L, a list of numbers.
  - n, the number of initial terms to add.
  - m, the number of end terms to add.
- `accumulate_head_tail(`L, n, m`)` returns the list with the n initial terms and m end terms replaced by their sums.

Example

```
> accumulate_head_tail([1,2,3,4,5,6,7,8,9,10],3,4)
```

$$[6, 4, 5, 6, 34]$$

20.1.12 Frequencies

The frequency of a number in a list is the fraction of the list equal to the number. The `frequencies` or `frequences` command finds the frequencies of the numbers in a list.

- `frequencies` takes L, a list of numbers.
- `frequencies(`L`)` returns a list whose elements are the numbers in the list and their frequencies.

Example

```
> frequencies([1,2,1,1,2,1,2,4,3,3])
```

$$\begin{bmatrix} 1 & 0.4 \\ 2 & 0.3 \\ 3 & 0.2 \\ 4 & 0.1 \end{bmatrix}$$

You can use this, for example, to simulate flipping a fair coin and seeing how many times each side appears; to flip a coin 1000 times, for example:

```
> frequencies([rand(2)$(k=1..1000)])
```

$$\begin{bmatrix} 0 & 0.484 \\ 1 & 0.516 \end{bmatrix}$$

(See Section 20.3.2, p. 520 for information on `rand`.)

20.1.13 Cumulative frequencies

Given a list of numbers L, the cumulated frequency at x is the fraction of numbers in the list less than x. The `cumulated_frequencies` command plots the cumulated frequency of the numbers in a list or given by a matrix.

For numbers in a list:

- `cumulated_frequencies` takes L, a list of numbers.
- `cumulated_frequencies(`L`)` draws the cumulated frequency of the numbers in L, where if L is a matrix, each number in the first column is repeated the number of times given in the second column.

For numbers in a matrix:

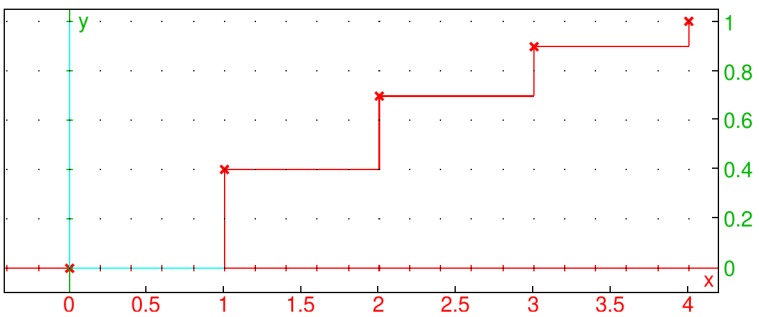
- `cumulated_frequencies` takes M, a matrix.
- `cumulated_frequencies(`M`)` (for a matrix with two columns, whose first column consists of numbers and whose second column consists of positive integers) draws the cumulated frequency of the numbers in the first column, where number in the first column is repeated the number of times given in the second column.
- `cumulated_frequencies(`M`)` (for a matrix with more than two columns, whose first column consists of numbers and whose remaining columns consist of positive integers) draws the cumulated frequencies for the first column paired with each remaining column.
- `cumulated_frequencies(`M`)` (for a matrix with two columns, whose first column consists of ranges $a..b$ and whose second column consists of positive numbers), will normalize the second column so the elements add up to 1 and draw the cumulated frequencies where the second column gives the frequency for the intervals in the first column.

Examples

```
> cumulated_frequencies([1,2,1,1,2,1,2,4,3,3])
```

or:

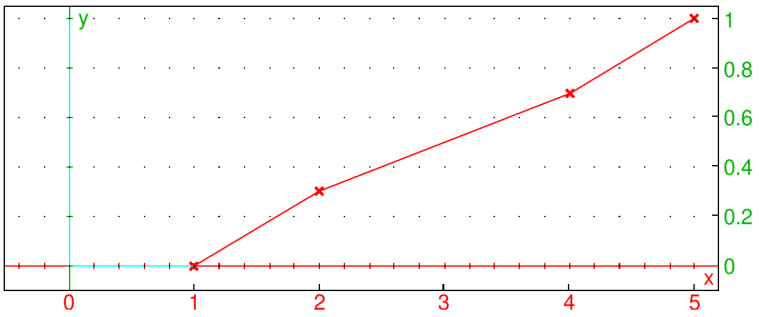
```
> cumulated_frequencies([[1,4],[2,3],[3,2],[4,1]])
```



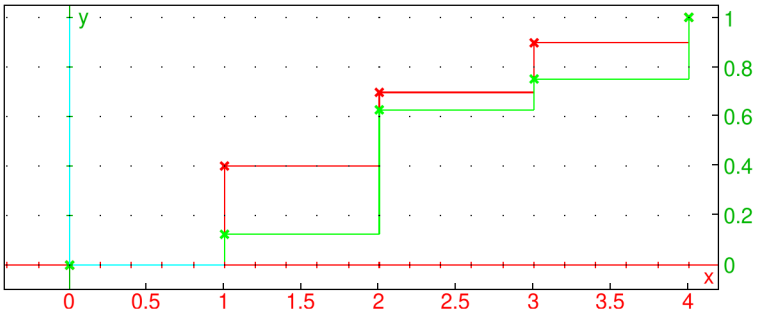
```
> cumulated_frequencies([[1..2,30],[2..4,40],[4..5,30]])
```

or:

```
> cumulated_frequencies([[1..2,0.3],[2..4,0.4],[4..5,0.3]])
```




```
> cumulated_frequencies([[1,4,1],[2,3,4],[3,2,1],[4,1,2]])
```



Here, both the distributions given by `[[1,4],[2,3],[3,2],[4,1]]` and `[[1,1],[2,4],[3,1],[4,2]]` are drawn on the same axes.

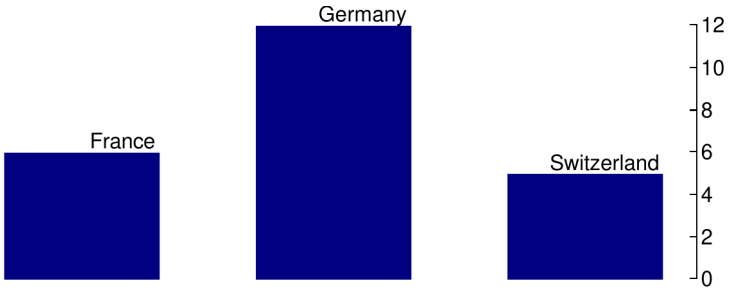
20.1.14 Bar plots

The `bar_plot` or `barplot` command draws bar plots.

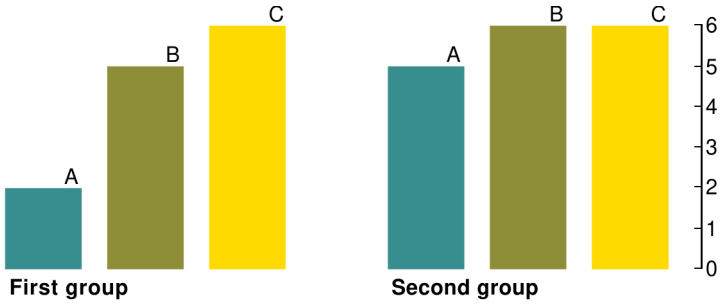
- `bar_plot` takes one mandatory argument and up to three optional arguments:
  - M, a matrix, where each row consists of a label followed by one or more values. If the labels are followed by more than one value, then the corresponding elements in the first row need to be identifiers or strings.
  - Optionally, `color=`*col*, where *col* is a color object or a list of color objects (see Section 19.1.3, p. 458). By default, *col* = `black`.
  - Optionally, *barwidth*, a real number in $\langle 0, 1]$ which specifies the width of bars (by default, *barwidth* = 0.75).
  - Optionally, `axes`, the symbol specifying that usual axes should be drawn, which is useful if you want to combine the output with other graphic objects.
- `bar_plot(`M $\langle$`,color=`*col*, *barwidth*, `axes` $\rangle$`)` draws a bar graph with a bar for each label, whose height is given by the corresponding value. If M is a two-column matrix, then the first column can be numbers specifying bar base midpoints. If the matrix has more than two columns, then there will be a bar plot for each column of values. If *col* is a color object, then the bars are filled with the specified color. If *col* is a list of colors, then the fill color cycles through the list for bars from left to right (each group of bars is colored by using the same colors).

Examples

```
> bar_plot([["France",6],["Germany",12],["Switzerland",5]],color=navy,0.618)
```

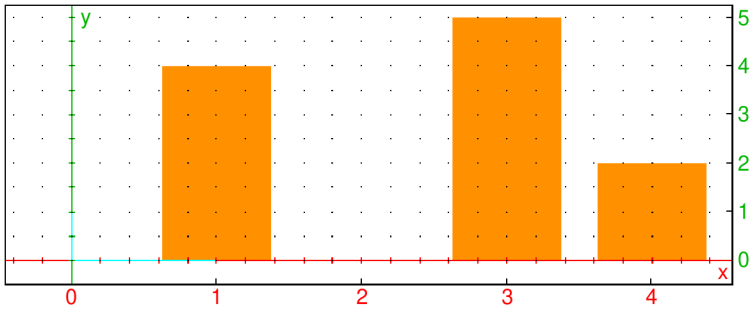



```
> bar_plot([[2,"*First group*","*Second group*"],
    ["A",2,5],["B",5,6],["C",6,6]],color=[teal,olive,gold])
```



To specify the midpoints of bar bases, the first argument must be a list of pairs `[`*midpoint*, *height*`]`. For example:

```
> barplot([[1,4],[3,5],[4,2]],color=orange)
```



Note that usual axes are automatically drawn in this case.

20.1.15 Pie charts

You can draw pie charts using the same structure as bar plots.

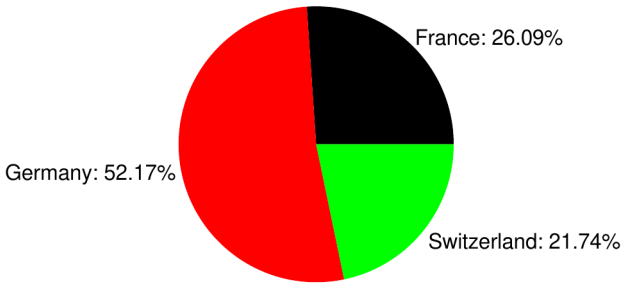
The `camembert` command draws pie charts.

- `camembert` takes one mandatory argument and one optional argument:
  - M, a matrix, where each row consists of a label followed by one or more values. If the labels are followed by more than one value, then the first row needs to be identifiers.
  - Optionally, `color=`*cols*, where *cols* is a list of two or more color objects (see Section 19.1.3, p. 458). By default, *cols* = [`black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`].

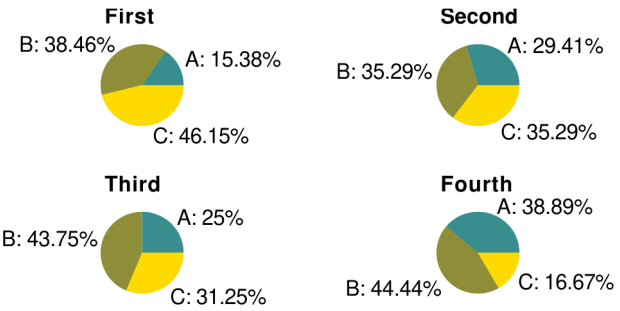
- `camembert(`M`)` draws a pie chart with a sector for each label, whose size is determined by the corresponding value. If the matrix has more than two columns, then there will be a pie chart for each column of values. Pie slices are filled with colors from *cols* such that the color index cycles counter-clockwise through the list (starting from the lowest slice in the first quadrant).

Examples

```
> camembert([["France",6],["Germany",12],["Switzerland",5]])
```




```
> camembert([[4,"*First*","*Second*","*Third*","*Fourth*"],
    ["A",2,5,4,7],["B",5,6,7,8],["C",6,6,5,3]],color=[teal,olive,gold])
```



20.2 Two variable statistics

20.2.1 Covariance and correlation

The covariance of two random variables measures their connectedness; i.e., whether they tend to change with each other. If X and Y are two random variables, then the covariance is the expected value of $(X - \bar{X})(Y - \bar{Y})$, where $\bar{X}$ and $\bar{Y}$ are the means of X and Y, respectively. The `covariance` command calculates covariances.

- `covariance` takes two mandatory and one optional argument:
  - X and Y, two lists.
  - Optionally, W, a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k).

  If the arguments are all lists, then can be entered as the columns of a single matrix.

  If the arguments consist of two lists and a matrix, to make it simpler to enter data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

$$\begin{bmatrix} \text{“XY”} & Y \\ X^T & W \end{bmatrix}$$

  For this, you have to give `covariance` a second argument of `-1`.

- `covariance(`$X, Y \langle, W\rangle$`)` returns the covariance of X and Y.

Examples

```
> covariance([1,2,3,4],[1,4,9,16])
```

or:

```
> covariance([[1,1],[2,4],[3,9],[4,16]])
```

$$\frac{25}{4}$$

```
> covariance([1,2,3,4],[1,4,9,16],[3,1,5,2])
```

or:

```
> covariance([1,2,3,4],[1,4,9,16],[[3,0,0,0],[0,1,0,0],[0,0,5,0],[0,0,0,2]])
```

or:

```
> covariance([["XY",1,4,9,16],[1,3,0,5,0],[2,0,1,0,0],[3,0,0,5,0],[4,0,0,0,2]],-1)
```

$$\frac{662}{121}$$

The linear correlation coefficient of two random variables is another way to measure their connectedness. Given random variables X and Y, their correlation is defined as $\text{cov}(X,Y)/(\sigma(X)\sigma(Y))$, $\text{cov}(X,Y)$ is the covariance of X and Y, and $\sigma(X)$ and $\sigma(Y)$ are the standard deviations of X and Y, respectively.

The `correlation` command finds the correlation of two lists and take the same types of arguments as the `covariance` command.

- `correlation` takes two mandatory and one optional argument:
  - X and Y, two lists.
  - Optionally, W, a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k).

  If the arguments are all lists, then can be entered as the columns of a single matrix.

  If the arguments consist of two lists and a matrix, to make it simpler to enter data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

  $$\begin{bmatrix} \text{"}XY\text{"} & Y \\ X^T & W \end{bmatrix}$$

  For this, you have to give `correlation` a second argument of `-1`.
- `correlation(`$X, Y \langle, W\rangle$`)` returns the correlation of X and Y.

Example

```
> correlation([1,2,3,4],[1,4,9,16])
```

$$\frac{100}{4\sqrt{645}}$$

The `covariance_correlation` command will compute both the covariance and correlation simultaneously, and return a list with both values. This command takes the same type of arguments as the `covariance` and `correlation` commands.

- `covariance_correlation` takes two mandatory and one optional argument:

– X and Y, two lists.

– Optionally, W, a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k).

If the arguments are all lists, then can be entered as the columns of a single matrix.

If the arguments consist of two lists and a matrix, to make it simpler to enter data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

$$\begin{bmatrix} "XY" & Y \\ X^T & W \end{bmatrix}$$

For this, you have to give `covariance_correlation` a second argument of `-1`.

- `covariance_correlation`$(X, Y \langle, W\rangle)$ returns a list consisting of the covariance and the correlation of X and Y.

Example

```
> covariance_correlation([1,2,3,4],[1,4,9,16])
```

$$\left[\frac{25}{4}, \frac{100}{4\sqrt{645}}\right]$$

20.2.2 Scatterplots

A scatter plot is simply a set of points plotted on axes. The `scatterplot` or `nuage_points` command draws scatter plots.

- `scatterplot` takes two arguments: *xcoords* and `ycoords`, a list of x-coordinates and y-coordinates. You can also combine them into a matrix with two columns (each list becomes a column of the matrix).

- `scatterplot`$($*xcoords*, *ycoords*$)$ draws the points with the given coordinates.

The `batons` command will also draw a collection of points, but each point will be connected to the x-axis with a vertical line segment.

- `batons` takes two arguments: *xcoords* and `ycoords`, a list of x-coordinates and y-coordinates. You can also combine them into a matrix with two columns (each list becomes a column of the matrix).

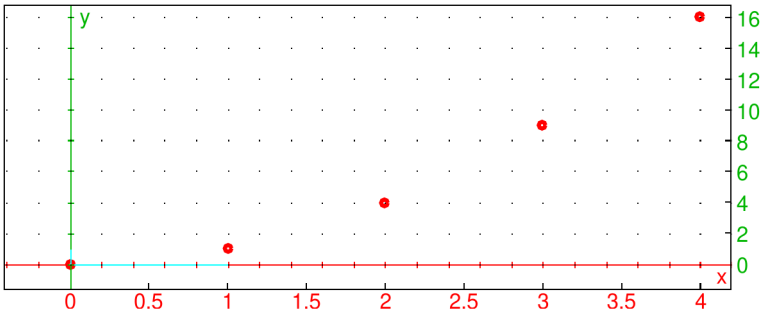
- `batons`$($*xcoords*, *ycoords*$)$ draws the points with the given coordinates and connects them to the x-axis with vertical line segments.

Examples

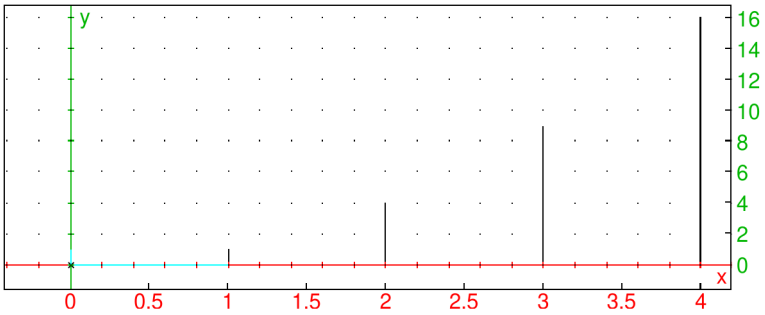
```
> scatterplot([[0,0],[1,1],[2,4],[3,9],[4,16]],display=point_width_3+point_point+red)
```

or:

```
> scatterplot([0,1,2,3,4],[0,1,4,9,16],display=point_width_3+point_point+red)
```




```
> batons([[0,0],[1,1],[2,4],[3,9],[4,16]])
```



As a practical example, assume that a model of the growth of a sunflower is given by the formula:

$$h(t) = \frac{256}{1 + 23\mathrm{e}^{-0.093t}},$$

where t is time in days and h is plant height in centimeters. The measured height is given in the following table:

t (days)	10	20	30	40	50	60	70
h (cm)	23	56	112	160	203	239	246

To define the function h, enter:

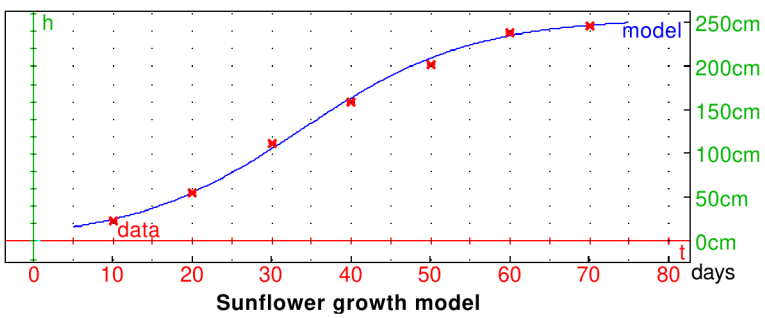
```
> h(t):=256/(1+23*exp(-0.093*t)):;
```

To load the data, enter:

```
> tdata:=[10,20,30,40,50,60,70]:; hdata:=[23,56,112,160,203,239,246]:;
```

To display the model together with data, enter:

```
> title="*Sunflower growth model*";
  labels=["t","h"];
  legend=["days","cm"];
  plotfunc(h(t),t=5..75,color=blue+quadrant4,legend="model");
  scatterplot(tdata,hdata,display=star_point+point_width_2+red+quadrant4,legend="data");
```



20.2.3 Polygonal paths

The `polygonplot` or `ligne_polygonale` command draws a polygonal path through given points.

- `polygonplot` takes one mandatory argument and one optional argument:
  - Optionally, *xcoords*, a list of x-coordinates. By default, the x-coordinates will be a list of integers starting at 0.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `polygonplot(`⟨*xcoords*,⟩*ycoords*`)` draws the polygonal path through the given points, from left to right (so the points are automatically ordered by increasing x-coordinate).

Examples

```
> polygonplot([0,1,4,9,16])
```

or:

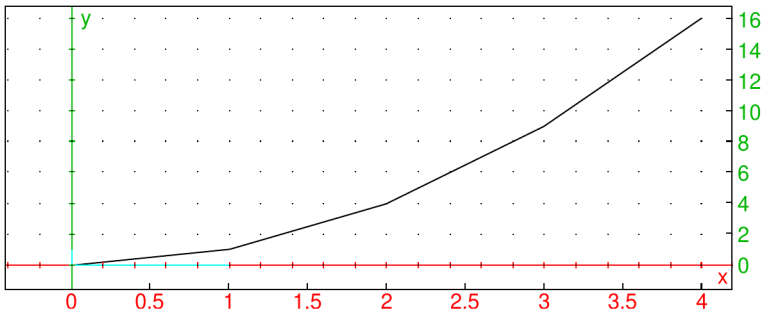
```
> polygonplot([0,1,2,3,4],[0,1,4,9,16])
```

or:

```
> polygonplot([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

or:

```
> polygonplot([[2,4],[0,0],[3,9],[1,1],[4,16]])
```



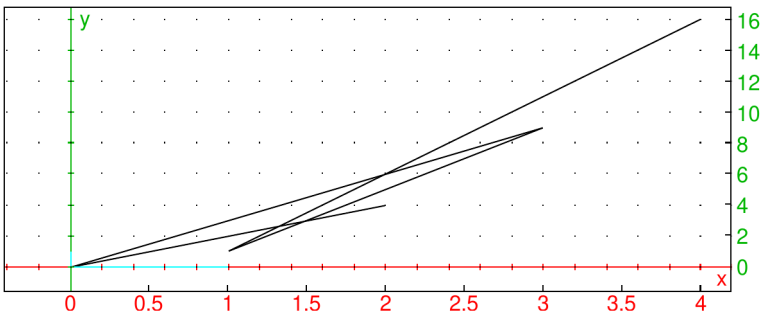
The `listplot` or `plotlist` draws a polygonal path, but in an order determined by you.

- `plotlist` takes L, a list of points or a list of numbers (which will be taken as y-coordinates, with the x-coordinates being the integers starting at 0).
- `plotlist(`L`)` draws a polygonal path through the points in the order given by the list.

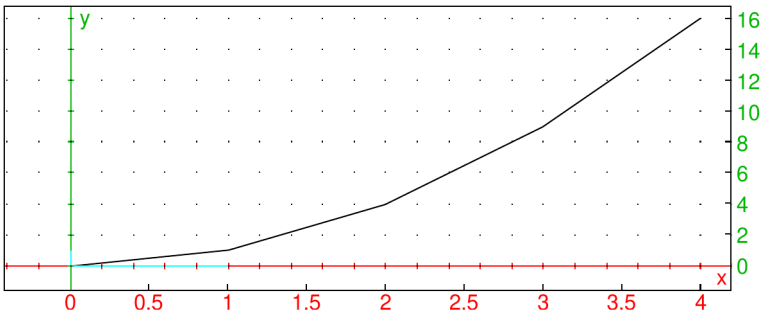
Unlike `polygonplot`, the `listplot` command cannot be given two lists of numbers as arguments.

Examples

```
> listplot([[2,4],[0,0],[3,9],[1,1],[4,16]])
```




```
> listplot([0,1,4,9,16])
```



If you want to get coordinates on the polygonal path, use the The `linear_interpolate` command will find coordinates on the polygonal path.

- `linear_interpolate` takes four arguments:
  - M, a two-row matrix consisting of the x-coordinates and the y-coordinates.
  - x_{min}, the minimum value of x that you are interested in.
  - x_{max}, the maximum value of x.
  - x_{step}, the step size that you want.

  The values of x_{min} and x_{max} must be between the smallest and largest x-coordinates of the points.
- `linear_interpolate`$(M, x_{min}, x_{max}, x_{step})$ returns a matrix with two rows, the first row will be $[x_{min}, x_{min} + x_{step}, x_{min} + 2x_{step}, \ldots, x_{max}]$ and the second row will be the corresponding y-coordinates of the points on the polygonal path.

Example

```
> linear_interpolate([[1,2,6,9],[3,4,6,12]],2,7,1)
```

$$\begin{bmatrix} 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 4.0 & 4.5 & 5.0 & 5.5 & 6.0 & 8.0 \end{bmatrix}$$

20.2.4 Linear regression

Given a set of points $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$, linear regression finds the line $y = mx + b$ that comes closest to passing through all of the points; i.e., that makes

$$\sqrt{(y_0 - (mx_0 + b))^2 + \cdots + (y_{n-1} - (mx_{n-1} + b))^2}$$

as small as possible. The `linear_regression` command finds the linear regression of a set of points.

- `linear_regression` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `linear_regression(`*xcoords*, *ycoords*`)` returns a sequence m, b of the slope and y-intercept of the regression line.

Example

```
> linear_regression([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

or:

```
> linear_regression([0,1,2,3,4],[0,1,4,9,16])
```

$$4, -2$$

which means that the line $y = 4x - 2$ is the best fit line.

The `linear_regression_plot` command draws the best fit line.

- `linear_regression_plot` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

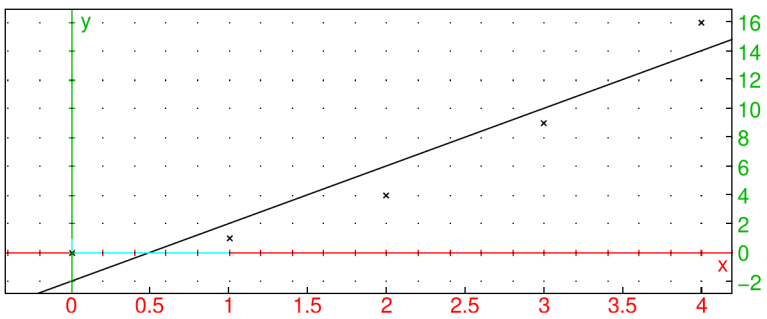
  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `linear_regression_plot(`*xcoords*, *ycoords*`)` draws the line of best fit through the points. It will also give you the equation at the top, as well as the R^2 value, which is

$$R^2 = \frac{\sum_{j=0}^{n-1}(mx_j + b - \bar{y})^2}{\sum_{j=0}^{n-1}(y_j - \bar{y})^2}$$

  (The R^2 value will be between 0 and 1 and is one measure of how good the line fits the data; a value close to 1 indicates a good fit, a value close to 0 indicates a bad fit.)

Example

```
> linear_regression_plot([0,1,2,3,4],[0,1,4,9,16])
```



20.2.5 Exponential regression

You might expect a set of points to lie on an exponential curve $y = ba^x$. The `exponential_regression` command finds the values of a and b which give you the best fit exponential.

- `exponential_regression` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `exponential_regression(`*xcoords*, *ycoords*`)` returns a sequence a, b of the numbers in the best fit exponential $y = ba^x$.

Example

```
> evalf(exponential_regression([[1,1],[2,4],[3,9],[4,16]]))
```

or:

```
> evalf(exponential_regression([1,2,3,4],[1,4,9,16]))
```

(where the `evalf` is used to get a numeric approximation to an exact expression, see Section 7.3.1, p. 128).

$$2.49146187923, 0.5$$

so the best fit exponential curve will be $y = 0.5 \cdot (2.49146187923)^x$.

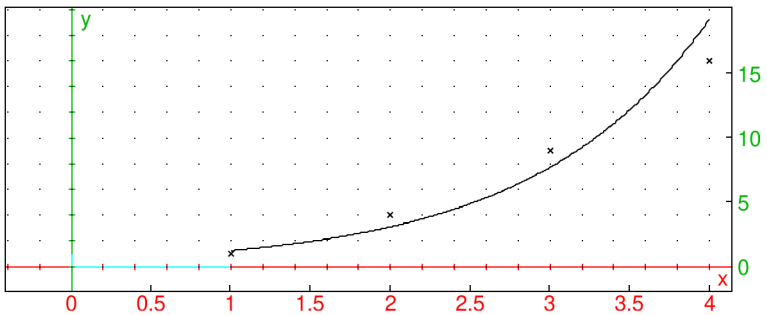
The `exponential_regression_plot` command draws the best fit exponential.

- `exponential_regression_plot` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `exponential_regression_plot(`*xcoords*, *ycoords*`)` draws the best fit exponential, and puts the equation and R^2 value above the graph.

Example

```
> exponential_regression_plot([1,2,3,4],[1,4,9,16])
```



20.2.6 Logarithmic regression

You might expect a set of points to lie on a logarithmic curve $y = m\ln(x)+b$. The `logarithmic_regression` command finds the logarithmic curve of best fit.

- `logarithmic_regression` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `logarithmic_regression(`*xcoords*, *ycoords*`)` returns a sequence m, b of the numbers in the best fit logarithmic curve $y = m\ln(x) + b$.

Example

```
> evalf(logarithmic_regression([[1,1],[2,4],[3,9],[4,16]]))
```

or:

```
> evalf(logarithmic_regression([1,2,3,4],[1,4,9,16]))
```

(where the `evalf` is used to get a numeric approximation to an exact expression):

$$10.1506450002, -0.564824055818$$

so the best fit logarithmic curve will be $y = 10.1506450002\ln(x) - 0.564824055818$.

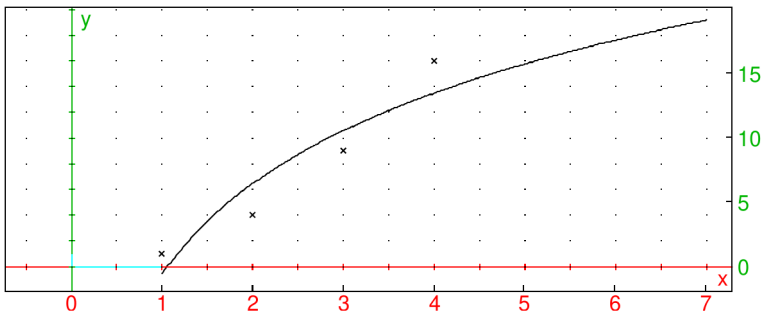
The `logarithmic_regression_plot` command draws the best fit logarithmic curve.

- `logarithmic_regression_plot` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `logarithmic_regression_plot(`*xcoords*, *ycoords*`)` draws the best fit logarithmic curve, and puts the equation and R^2 value above the graph.

Example

```
> logarithmic_regression_plot([1,2,3,4],[1,4,9,16])
```



20.2.7 Power regression

The `power_regression` command finds the graph $y = bx^m$ which best fits a set of data points.

- `power_regression` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `power_regression(`*xcoords*, *ycoords*`)` returns a sequence m, b of the numbers in the best fit power equation $y = bx^m$.

Example

```
> power_regression([[1,1],[2,4],[3,9],[4,16]])
```

or:

```
> power_regression([1,2,3,4],[1,4,9,16])
```

$$2.0, 1.0$$

so the best fit (in this case, exact fit) power curve will be $y = 1.0\,x^2$.

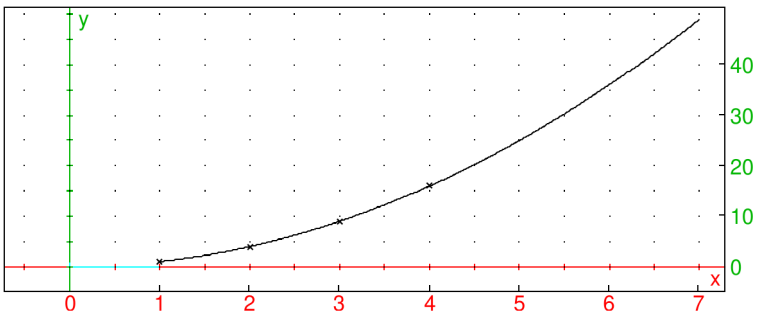
The `power_regression_plot` command draws the best fit power function.

- `power_regression_plot` takes two arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.

  You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `power_regression_plot(`*xcoords*, *ycoords*`)` draws the best fit power function, and puts the equation and R^2 value above the graph.

Example

```
> power_regression_plot([1,2,3,4],[1,4,9,16])
```



Note that in this case the R^2 value is 1, indicating that data points fall directly on the curve.

20.2.8 Polynomial regression

The `polynomial_regression` command finds a more general polynomial $y = a_0x^n + \cdots + a_n$ which best fits a set of data points.

- `polynomial_regression` takes three arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.
  - n, the degree of the polynomial.

  You can combine the first two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `polynomial_regression(`*xcoords*, *ycoords*, n`)` returns the list $[a_n, \ldots, a_0]$ of coefficients of the best fit polynomial.

Example

```
> polynomial_regression([[1,1],[2,2],[3,10],[4,20]],3)
```

or:

```
> polynomial_regression([1,2,3,4],[1,2,10,20],3)
```

$$\left[-\frac{5}{6}, \frac{17}{2}, -\frac{56}{3}, 12\right]$$

so the best fit polynomial will be $y = (-5/6)x^3 + (17/2)x^2 - (56/3)x + 12$.

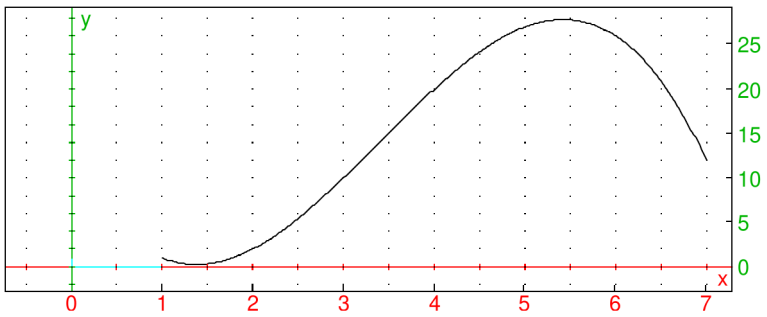
The `polynomial_regression_plot` command draws the best fit polynomial.

- `polynomial_regression_plot` takes three arguments:
  - *xcoords*, a list of x-coordinates.
  - *ycoords*, a list of y-coordinates.
  - n, the degree of the polynomial.

  You can combine the first two arguments into a matrix with two columns (each list becomes a column of the matrix).
- `polynomial_regression_plot(`*xcoords*, *ycoords*, n`)` draws the best fit polynomial of degree n, and puts the equation and R^2 value above the graph.

Example

```
> polynomial_regression_plot([1,2,3,4],[1,2,10,20],3)
```



20.2.9 Logistic regression

Differential equations of the form $y' = y(ay + b)$ come up often, particularly when studying bounded population growth. With the initial condition $y(x0) = y0$, the solution is the logistic equation

$$y = \frac{-by_0}{ay_0 - (ay_0 + b)e^{b(x_0 - x)}}.$$

However, you often do not know the values of a and b. You can approximate these values given (x_0, y_0) and $[y'(x_0), y'(x_0 + 1), \ldots, y'(x_0 + n - 1)]$ by taking the initial value $y(x_0) = y_0$ and the approximation $y(t + 1) \approx y(t) + y'(t)$ to get the approximations

$$\begin{aligned} y(x_0 + 1) &\approx y_0 + y'(x_0) \\ y(x_0 + 2) &\approx y_0 + y'(x_0) + y'(x_0 + 1) \\ &\vdots \\ y(x_0 + n) &\approx y_0 + y'(x_0) + \cdots + y'(x_0 + n - 1) \end{aligned}$$

Since $y'/y = a + by$, you can take the approximate values of $y'(x0 + j)/y(x0 + j)$ and use linear interpolation to get the best fit values of a and b, and then solve the differential equation.

The `logistic_regression` command uses this approach to find the best fit logistic equation for given data.

- `logistic_regression` takes three arguments:
  - L, a list representing $[y_{10}, y_{11}, \ldots, y_{1(n-1)}]$, where y_{1j} represents the value of $y'(x_0 + j)$.
  - x_0, the initial x value.
  - y_0, the initial y value.
- `logistic_regression(`L, x_0, y_0`)` returns a list $[y, y', C, y_{max}, x_{max}, R, Y]$ where y is the logistic function, y' is the derivative, $C = -b/a$, y_{max} is the maximum value of y', x_{max} is where y' has its maximum, R is linear correlation coefficient R of $Y = y'/y$ as a function of y with $Y = ay + b$.

Example

```
> logistic_regression([0.0,1.0,2.5],0,1)
```

```
Pinstant=0.132478632479*Pcumul+0.0206552706553
Correlation 0.780548607383, Estimated total P=-0.155913978495
Returning estimated Pcumul, Pinstant, Ptotal, Pinstantmax, tmax, R
```

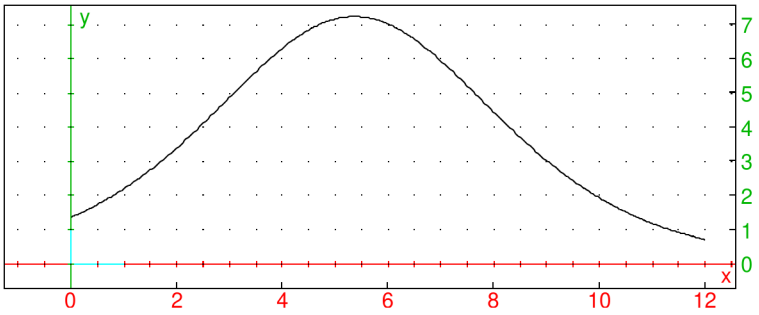
$$\left[-\frac{0.155913978495}{1 + e^{-0.0554152581707x + (0.140088513344 + 3.14159265359i)}},\right.$$
$$-\frac{0.00161022271237}{1 + \cos\left(-i\left(-0.0554152581707x + (0.140088513344 + 3.14159265359i)\right)\right)},$$
$$-0.155913978495, -0.000805111356186, 2.52797727501 + 56.6918346552i,$$
$$\left.0.780548607383\right]$$

The `logistic_regression_plot` command draws the best fit logistic equation.

- `logistic_regression_plot` takes three arguments:
  - L, a list representing $[y_{10}, y_{11}, \ldots, y_{1(n-1)}]$, where y_{1j} represents the value of $y'(x_0 + j)$.
  - x_0, the initial x value.
  - y_0, the initial y value.
- `logistic_regression_plot(`L, x_0, y_0`)` draws the best fit logistic equation.

Example

```
> logistic_regression_plot([1,2,4,6,8,7,5],0,2.0)
```



20.3 Random numbers

20.3.1 Initializing the random number generator

The `srand` and `RandSeed` or `randseed` commands initialize (or re-initialize) the random numbers given by `rand`.

- `srand` takes one optional argument:

  Optionally, n, an integer.

- `srand(`n`)` initializes the random numbers.
- `srand` (no parentheses) initializes the random numbers using the system clock.
- `RandSeed` takes n, an integer.
- `RandSeed(`n`)` initializes the random numbers.

20.3.2 Generating uniformly distributed random numbers

The `rand` or `random` or `hasard` command produces random numbers, chooses random elements from a list, or creates functions that produce random numbers.

To produce random real numbers:

- `rand` takes two optional arguments.
  - Optionally, a and b, two real numbers. By default, $a = 0$ and $b = 1$.
- `rand([a,b])` returns a number in $[a, b)$ randomly and with equal probability.

To produce random integers:

- `rand` takes n, an integer.
- `rand(`n`)` returns a random integer in $[0, n)$ (or $(n, 0]$ if n is negative).

Examples

```
> rand()
```

(to produce a random number in $[0, 1)$).

0.528489416465

```
> rand(1,1.5)
```

(to produce a random number in $[1, 1.5)$).

1.0012010464

```
> rand(5)
```

3

You can then use `rand` to find a random integer in a specified interval; if you want an random integer between 6 and 10, inclusive, for example, enter:

```
> 6+rand(11-6)
```

7

Another way to get a random integer in a specified interval is with the `randint` command.

- `randint` takes two arguments: n_1 and n_2, two integers.
- `randint`(n_1, n_2) returns a random integer between n_1 and n_2, inclusive.

Example

```
> randint(6,10)
```

8

- To make a function which produces random numbers, `rand` takes $a..b$, a range with real numbers a and b.
- `rand`$(a..b)$ returns a function which will generate a random number in the interval from a to b.

Example

```
> r:=rand(1.0..2.5):;
  r()
```

1.68151313369

- To choose elements without replacement, `rand` takes two or three arguments:
  - p, a positive integer.
  - Either: n_1 and n_2, two integers.
  - or: L, a list.
- `rand`(p, n_1, n_2) returns a list of p distinct random integers from n_1 to n_2.
- `rand`(L) returns p elements without replacement from the list L.

Examples

```
> rand(2,1,10)
```

$[2, 9]$

```
> rand(3,["a","b","c","d","e","f","g","h"])
```

[“e”, “g”, “a”]

The list can have repeated elements.

```
> rand(4,["r","r","r","r","v","v","v"])
```

[“r”, “v”, “v”, “r”]

The `sample` command will also randomly select items from a list without replacement.

- `sample` takes two arguments:
  - L, a list.
  - p, an integer.
- `sample(`L, p`)` returns a list of p items chosen randomly from L, without replacement.

Note that with the `sample` command, the list comes first and then the integer.

Example

```
> sample(["r","r","r","r","v","v","v"],4)
```

[“v”, “v”, “v”, “r”]

20.3.3 Sampling from probability distributions

Generating random numbers from binomial distribution. The `randbinomial` command finds random numbers chosen according to the binomial distribution (see Section 20.4.3, p. 535).

- `randbinomial` takes two arguments:
  - n, an integer.
  - p, a probability (a number between 0 and 1).
- `randbinomial(`n, p`)` returns an integer from 0 to n chosen randomly according to the binomial distribution with parameters n and p; i.e., the number of successes you might get if you did an experiment n times, where the probability of success each time is p.

Example

```
> randbinomial(100,0.4)
```

42

Generating random numbers from multinomial distribution. The `randmultinomial` command finds random numbers chosen according to a multinomial distribution (see Section 20.4.5, p. 538).

- `randmultinomial` takes one mandatory and one optional argument:
  - P, a list $P = [p_0, \ldots, p_{n-1}]$ of n probabilities which add to 1 (representing the probability that one of several mutually exclusive events occurs).
  - Optionally, K, a list of length n.
- `randmultinomial(`L`)` returns an index chosen randomly according to the corresponding multinomial distribution.
- `randmultinomial(`L, K`)` returns an element of K whose index is chosen randomly.

Examples

```
> randmultinomial([1/2, 1/3, 1/6])
```

1

```
> randmultinomial([1/2, 1/3, 1/6],["R","V","B"])
```

"R"

Generating random numbers from Poisson distribution. Recall that given a number $\lambda > 0$, the corresponding Poisson distribution $P(\lambda)$ satisfies

$$\text{Prob}(X \leq k) = \mathrm{e}^{-\lambda}\frac{\lambda^k}{k!}.$$

It will have mean λ and standard deviation $\sqrt{\lambda}$. (See also Section 20.4.6, p. 539.)

The `randpoisson` command finds a random integer according to a Poisson distribution.

- `randpoisson` takes λ, a positive number.
- `randpoisson(`λ`)` returns an integer chosen randomly according to the Poisson distribution with paramter λ.

Example

```
> randpoisson(10.6)
```

16

Generating random numbers from normal distribution. The `randnorm` or `randNorm` command chooses a random number according to a normal distribution.

- `randnorm` takes two arguments:
  - μ, a real number (the mean).
  - σ, a positive real number (the standard deviation).
- `randnorm(`μ, σ`)` returns a number chosen randomly according the normal distribution with mean μ and standard deviation σ.

Example

```
> randnorm(2,1)
```

3.39283224858

Generating random numbers from Student's distribution. The `randstudent` command finds random numbers chosen according to Student's distribution (see Section 20.4.8, p. 542).

- `randstudent` takes n, an integer (the degrees of freedom).
- `randstudent(`n`)` returns a number chosen randomly according to Student's distribution with n degrees of freedom.

Example

```
> randstudent(5)
```

0.268225314184

Generating random numbers from χ^2 distribution. The `randchisquare` command finds random numbers chosen according to the χ^2 distribution (see Section 20.4.9, p. 544).

- `randchisquare` takes n, an integer (the degrees of freedom).
- `randchisquare(`n`)` returns a number chosen randomly according to the χ^2 distribution with n degrees of freedom.

Example

```
> randchisquare(5)
```

4.53970828547

Generating random numbers from Fisher-Snédécor distribution. The `randfisher` command finds random numbers chosen according to the Fisher-Snédécor distribution (see Section 20.4.10, p. 545).

- `randfisher` takes two arguments: n_1 and n_2, integers (degrees of freedom).
- `randfisher(`n_1, n_2`)` returns a number chosen randomly according to the Fisher-Snédécor distribution with n_1 and n_2 degrees of freedom.

Example

```
> randfisher(2,3)
```

2.33137725333

Generating random numbers from gamma distribution. The `randgammad` command finds random numbers chosen according to the gamma distribution (see Section 20.4.11, p. 547).

- `randgammad` takes two arguments: a and b, positive real numbers (the parameters).
- `randgammad(`a, b`)` returns a number chosen randomly according to the gamma distribution with parameters a and b.

Example

```
> randgammad(3,1)
```

4.91461463472

Generating random numbers from beta distribution. The `randbetad` command finds random numbers chosen according to the beta distribution (see Section 20.4.12, p. 548).

- `randbetad` takes two arguments: a and b, positive real numbers (the parameters).
- `randbetad(`a, b`)` returns a number chosen randomly according to the beta distribution with parameters a and b.

Example

```
> randbetad(2,3)
```

0.524453873081

Generating random numbers from geometric distribution. The `randgeometric` command finds random numbers chosen according to the geometric distribution (see Section 20.4.13, p. 549).

- `randgeometric` takes p, a probability (a number between 0 and 1).
- `randgeometric(`p`)` returns a number chosen randomly according to the geometric distribution with probability p.

Example

```
> randgeometric(0.2)
```

11

Generating random numbers from exponential distribution. The `randexp` command finds random numbers chosen according to the exponential distribution (see Section 20.4.15, p. 552).

- `randexp` takes λ, a positive real number (the parameter).
- `randexp(`λ`)` returns a number chosen randomly according to the exponential distribution with parameter λ.

Example

```
> randexp(2.1)
```

0.0288626239833

20.3.4 Random variables

The `randvar` or `random_variable` command produces an object representing a random variable. The value(s) can be generated subsequently by calling `sample` (see Section 20.3.2, p. 520), `rand` (see Section 20.3.2, p. 520), `randvector` (see Section 20.3.5, p. 531) or `randmatrix` (see Section 20.3.6, p. 532).

- `randvar` takes a sequence of arguments: *distspec*, which specifies a probability distribution with parameters. The following distributions are supported:
  - **Uniform distribution** (see Section 20.4.2, p. 534). Arguments:
    - `uniform` or `uniformd`.
    - a and b, two numbers specifying the end points of a range.

      The range can also be specified by a`..`b or `range=`a`..`b.
  - **Binomial distribution** (see Section 20.4.3, p. 535). Arguments:
    - `binomial`.
    - n, a positive integer.
    - p, a probability (a number between 0 and 1).
  - **Negative binomial distribution** (see Section 20.4.4, p. 537). Arguments:
    - `negbinomial`.
    - n, a positive integer.
    - p, a probability (a number between 0 and 1).
  - **Multinomial distribution** (see Section 20.3.3, p. 523). Arguments:
    - `multinomial`.
    - $[p_0, p_1, \ldots, p_j]$, a list of probabilities with $p_0 + \cdots + p_j = 1$.
    - Optionally, $[a_0, a_1, \ldots, a_j]$, a list of possible return values.
  - **Normal distribution** (see Section 20.4.7, p. 540). Arguments:
    - `normal` or `normald`.
    - no arguments (for the standard normal distribution) or two numbers μ and σ specifying the mean and the standard deviation.
  - **Poisson distribution** (see Section 20.4.6, p. 539). Arguments:
    - `poisson`.
    - λ, a positive real number.
  - **Student's distribution** (see Section 20.4.8, p. 542). Arguments:
    - `student`.
    - n, an integer (the degrees of freedom).
  - χ^2 **distribution** (see Section 20.4.9, p. 544). Arguments:
    - `chisquare`.
    - n, an integer (the degrees of freedom).
  - **Fisher-Snédécor distribution** (see Section 20.4.10, p. 545). Arguments:
    - `fisher`, `fisherd`, or `snedecor`.
    - n_1 and n_2, integers (the degrees of freedom).

  - **Gamma distribution** (see Section 20.4.11, p. 547). Arguments:
    * `gammad`.
    * a and b, real numbers.
  - **Beta distribution** (see Section 20.4.12, p. 548). Arguments:
    * `betad`.
    * a and b, real numbers.
  - **Geometric distribution** (see Section 20.4.13, p. 549). Arguments:
    * `geometric`.
    * p, a number between 0 and 1.
  - **Cauchy distribution** (see Section 20.4.14, p. 551). Arguments:
    * `cauchy` or `cauchyd`.
    * a and b, real numbers.
  - **Exponential distribution** (see Section 20.4.15, p. 552). Arguments:
    * `exp` or `exponential` or `exponentiald`.
    * λ, a positive real number.
  - **Weibull distribution** (see Section 20.4.16, p. 553). Arguments:
    * `weibull` or `weibulld`.
    * k, an integer.
    * λ, a real number.
  - **Discrete (categorical) distributions**. Arguments:
    * $W = [w_1, w_2, \ldots, w_n]$, a list of nonnegative weights.
    * Optionally, $V = [v_1, v_2, \ldots, v_n]$, a list of values, or $[[v_1, w_1], [v_2, w_2], \ldots, [v_n, w_n]]$, a list of object-weight pairs, or f, a nonnegative function.
    * $a..b$ or `range=`$a..b$ with real numbers a and b, a range specification.
    * Optionally, N, a positive integer or $V = [v_0, v_1, v_2, \ldots, v_n]$, a list of values with $n = b - a$ (here a and b have to be integers).

    The weights are automatically scaled by the inverse of their sum to obtain the values of the probability mass function. If a function f is given instead of a list of weights, then $w_k = f(a + k)$ for $k = 0, 1, \ldots, b - a$ unless N is given, in which case $w_k = f(x_k)$ where $x_k = a + (k - 1)\frac{b-a}{N}$ and $k = 1, 2, \ldots, N$. The resulting random variable X has values in $\{0, 1, \ldots, n - 1\}$ for 0-based modes (e.g. XCAS) resp. in $\{1, 2 \ldots, n\}$ for 1-based modes (e.g. MAPLE). If the list V of custom objects is given, then $V[X]$ is returned instead of X. If N is given, then $v_k = x_k$ for $k = 1, 2, \ldots, N$.

- The parameters of uniform, normal, Poisson, geometric, exponential, binomial, negative binomial, beta, gamma, and Weibull distribution can be computed from the first and/or second moment which can be specified by the following arguments:
  - `mean=`μ, to specify a mean of μ.
  - `stddev=`σ, to specify a standard deviation.
  - `variance=`σ^2, to specify a variance.

If there is no distribution of the given type that fits the given moments, an error is returned. Note that binomial and negative binomial distributions, which depend on an integral parameter, may not fit the moments exactly.

- `randvar(`*distspec*`)` returns an object representing a random variable.

Examples

Define a random variable with a Fisher-Snedecor distribution (two degrees of freedom).

```
> X:=random_variable(fisher,2,3)
```

$$\text{fisherd}\,(2,3)$$

To generate one or more values of X, use the following commands.

```
> rand(X)
```

or:

```
> sample(X)
```

$$0.30584514472$$

```
> randvector(5,X)
```

or:

```
> sample(X,5)
```

$$[2.2652, 0.1397, 6.3320, 1.0556, 0.2995]$$

Define a random variable with multinomial distribution.

```
> M:=randvar(multinomial,[1/2,1/3,1/6],[a,b,c])
```

$$\text{multinomial}, \left[\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right], [a, b, c]$$

```
> randvector(10,M)
```

$$[b, b, b, b, b, b, a, a, b, b]$$

Some continuous distributions can be defined by specifying their first and/or second moment.

```
> randvector(10,randvar(poisson,mean=5))
```

$$[7, 2, 5, 6, 7, 9, 8, 4, 3, 4]$$

```
> randvector(5,randvar(weibull,mean=5.0,stddev=1.5))
```

$$[1.6124, 3.2720, 7.02627, 5.5360, 3.1929]$$

```
> X:=randvar(binomial,mean=18,stddev=4)
```

$$\binom{162}{\frac{1}{9}}$$

```
> X:=randvar(weibull,mean=12.5,variance=1)
```

$$\text{weibulld}\,(3.08574940721, 13.9803128143)$$

```
> mean(randvector(1000,X))
```

12.5728578447

```
> G:=randvar(geometric,stddev=2.5)
```

geometric (0.327921561087)

```
> evalf(stddev(randvector(1000,G)))
```

2.57913473863

```
> randvar(gammad,mean=12,variance=4)
```

gammad (36, 3)

Uniformly distributed random variables can be defined by specifying the support as an interval.

```
> randvector(5,randvar(uniform,range=15..81))
```

[77.0025, 77.7644, 63.2414, 52.0707, 66.3837]

```
> rand(randvar(uniform,e..pi))
```

3.1010453504

The following examples demonstrate various ways to define a discrete random variable.

```
> X:=randvar([["apple",1/3],["orange",1/4],["pear",1/5],["plum",13/60]]):;
  randvector(5,X)
```

[“orange”, “apple”, “apple”, “plum”, “apple”]

```
> W:=[1,4,5,3,1,1,1,2]:; X:=randvar(W):;
  approx(W/sum(W))
```

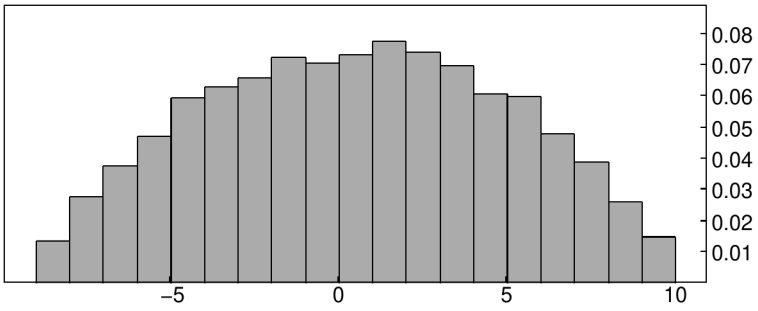
[0.0556, 0.2222, 0.2778, 0.1667, 0.0556, 0.0556, 0.0556, 0.1111]

```
> frequencies(randvector(10000,X))
```

(See Section 20.1.12, p. 504.)

$$\begin{bmatrix} 0 & 0.0527 \\ 1 & 0.2189 \\ 2 & 0.2791 \\ 3 & 0.1698 \\ 4 & 0.0546 \\ 5 & 0.0557 \\ 6 & 0.059 \\ 7 & 0.1102 \end{bmatrix}$$

```
> X:=randvar(k->1-(k/10)^2,range=-10..10):;
  histogram(randvector(10000,X))
```




```
> X:=randvar([3,1,2,5],[alpha,beta,gamma,delta]):;
  randmatrix(5,4,X)
```

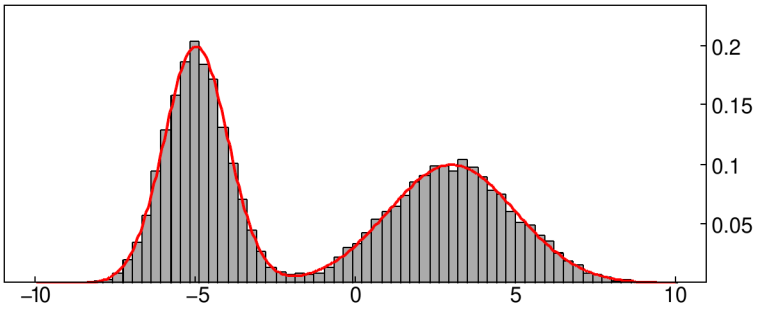
$$\begin{bmatrix} \delta & \delta & \beta & \delta \\ \delta & \gamma & \gamma & \beta \\ \alpha & \delta & \alpha & \delta \\ \alpha & \alpha & \gamma & \alpha \\ \delta & \delta & \beta & \delta \end{bmatrix}$$

Discrete random variables can be used to approximate custom continuous random variables. For example, consider a probability density function f as a mixture of two normal distributions on the support $S = [-10, 10]$. You can sample f in $N = 10000$ points in S.

```
> F:=normald(3,2,x)+normald(-5,1,x):;
  c:=integrate(F,x=-10..10):;
  f:=unapply(1/c*F,x):;
  X:=randvar(f,range=-10..10,10000):;
```

Now generate 25000 values of X and plot a histogram:

```
> R:=sample(X,25000):;
  hist:=histogram(R,-10,0.3):;
  PDF:=plot(f(x),display=red+line_width_2):;
  hist,PDF
```



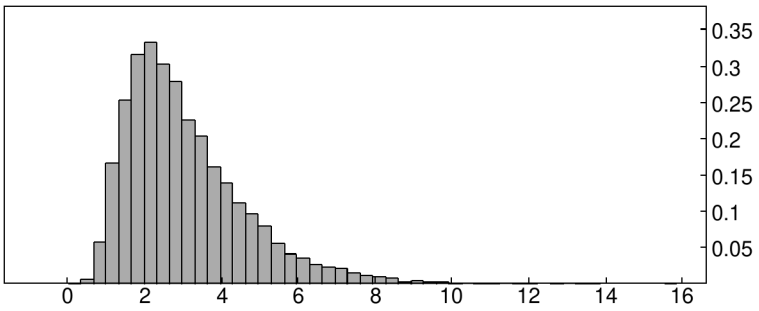
Sampling from discrete distributions is fast: for instance, generating 25 million samples from the distribution of X which has about 10000 outcomes takes only couple of seconds. In fact, the sampling complexity is constant. Also, observe that the process does not slow down by spreading it across multiple calls of `randvector`.

```
> for k from 1 to 1000 do randvector(25000,X); od:;
```

Evaluation time: 2.12

Independent random variables can be combined in an expression, yielding a new random variable. In the example below, you define a log-normally distributed variable Y from a variable X with the standard normal distribution.

```
> X:=randvar(normal):; mu,sigma:=1.0,0.5:;
  Y:=exp(mu+sigma*X):;
  L:=randvector(10000,Y):;
  histogram(L,0,0.33)
```



It is known that $E[Y] = e^{\mu+\sigma^2/2}$. The mean of L should be close to that number.

```
> mean(L); exp(mu+sigma^2/2)
```

$$3.0789, 3.0802$$

If a compound random variable is defined as an expression containing several independent random variables $X, Y, \ldots$ of the same type, you sometimes need to prevent its evaluation when passing it to `randvector` and similar functions. Let e.g.

```
> X:=randvar(normal):; Y:=randvar(normal):;
```

If you want to generate, for example, the random variable X/Y, you would have to forbid automatic evaluation of the latter expression; otherwise it would reduce to 1 since X and Y are both `normald`$(0, 1)$.

```
> randvector(5,eval(X/Y,0))
```

$$[-0.358479277895, 5.03004946974, -5.5414073892, -0.885656967277, -2.63689662108]$$

To save typing, you can define Z with `eval`$(*, 0)$ and pass `eval`$(Z, 1)$ to `randvector` or `randmatrix`.

```
> Z:=eval(X/Y,0):; randvector(5,eval(Z,1))
```

$$[0.404123429613, -4.06194898981, 0.00356038536404, 1.61619003525, -2.85682173195]$$

Parameters of a distribution can be entered as symbols to allow (re)assigning them at any time. For example, input:

```
> purge(lambda):; X:=randvar(exp,lambda):; lambda:=1:;
```

Now execute the following command line several times in a row. The parameter λ is updated in each iteration.

```
> r:=rand(X); lambda:=sqrt(r)
```

Output obtained by executing the above command line three times:

$$8.5682, 2.9272$$
$$1.5702, 1.2531$$
$$0.53244, 0.72968$$

20.3.5 Generating random vectors and lists

The `randvector` command creates random vectors (see also Section 11.1.28, p. 224).

- `randvector` takes one mandatory argument and one optional argument:

  - n, an integer.
  - Optionally, X, which can be an integer or a random variable. In place of a random variable, the specifications for a distribution can be used. (See Section 20.3.4, p. 526 for random variables and their specifications.)

- `randvector`(n, X) returns a vector of size n containing random integers:
  - with no second argument, distributed uniformly between -99 and $+99$.
  - with a second argument of X, distributed uniformly between 0 and $k-1$.
  - with a second argument the specification of a distribution, distributed according to this distribution.

Examples

```
> randvector(3)
```

$$[-64, -30, 70]$$

```
> randvector(3,5)
```

or:

```
> randvector(3,'rand(5)')
```

$$[2, 4, 2]$$

```
> randvector(3,'randnorm(0,1)')
```

$$[-0.361127118455, -0.018325111754, 1.11875485898]$$

```
> randvector(3,2..4)
```

$$[3.18034843914, 2.48592940345, 2.57507958449]$$

20.3.6 Generating random matrices

The `randmatrix` or `ranm` or `randMat` command produces random vectors and matrices. (See also Section 11.1.28, p. 224 and Section 20.3.5, p. 531.)

- `randmatrix` takes one mandatory argument and three optional arguments:
  - n, an integer.
  - Optionally, p, an integer.
  - Optionally, a, an integer.
  - Optionally, $a..b$, a range.
  - Optionally, *distr*, a distribution, which can be one of:
    - `'rand(`n`)'` (see Section 20.3.2, p. 520).
    - `'binomial(`n, p`)'`, `'binomial,`n, p`'` or `'randbinomial(`n, p`)'`, for a binomial distribution (see Section 20.4.3, p. 535 and Section 20.3.3, p. 522).
    - `'multinomial(`P, K`)'`, `'multinomial,`P, K`'` or `'randmultinomial(`P, K`)'` for a multinomial distribution (see Section 20.4.5, p. 538 and Section 20.3.3, p. 523).
    - `'poisson(`λ`)'`, `'poisson, `λ`'` or `'randpoisson(`λ`)'` for a Poisson distribution (see Section 20.4.6, p. 539 and Section 20.3.3, p. 523).

  - `'normald(`μ,σ`)'`, `'normald,`μ,σ`'` or `'randnorm(`μ,σ`)'` for a normal distribution (see Section 20.4.7, p. 540 and Section 20.3.3, p. 523).
  - `'exp(`a`)'`, `'exp,`a`'` or `'randexp(`a`)'` for an exponential distribution (see Section 20.4.15, p. 552 and Section 20.3.3, p. 525).
  - `'fisher(`n,m`)'`, `'fisher,`n,m`'` or `'randfisher(`n,m`)'` for a Fisher-Snédécor distribution (see Section 20.4.10, p. 545 and Section 20.3.3, p. 524).

  Note that *distr* is in quotes.

- `randmatrix(`n`)` returns a vector of length n whose elements are integers chosen randomly from $\{-99,-98,\ldots,98,99\}$ with equal probability.
- `randmatrix(`n,p`)` returns an $n\times p$ matrix whose elements are integers chosen randomly from $\{-99,\ldots,99\}$ with equal probability.
- `randmatrix(`n,p,a`)` returns an $n\times p$ matrix whose elements are integers chosen randomly from $[0,a)$ (or $(a,0]$ is a is negative) with equal probability.
- `randmatrix(`n,p,a`..`b`)` returns an $n\times p$ matrix whose elements are real numbers chosen randomly from $[a,b)$ with equal probability.
- `randmatrix(`n,p`,`*distr*`)` returns an $n\times p$ matrix whose elements are numbers chosen randomly according to distribution *distr*.

Examples

```
> randmatrix(5)
```

$$[-48,54,28,-51,63]$$

```
> randmatrix(2,3)
```

$$\begin{bmatrix}40 & -74 & -87\\ 40 & -19 & 20\end{bmatrix}$$

```
> randmatrix(2,3,10)
```

$$\begin{bmatrix}4 & 2 & 1\\ 4 & 4 & 0\end{bmatrix}$$

```
> randmatrix(2,3,0..1)
```

$$\begin{bmatrix}0.384355471935 & 0.655490326229 & 0.924850208685\\ 0.159429819323 & 0.952957109548 & 0.220945354551\end{bmatrix}$$

```
> randmatrix(2,3,'randnorm(2,1)')
```

$$\begin{bmatrix}2.17670501195 & 0.653882567048 & 2.94543112983\\ 2.46150672679 & 2.19251320854 & 2.44211638655\end{bmatrix}$$

20.4 Density and distribution functions

20.4.1 Distributions and inverse distributions

Let $p(x)$ be a probability density function, so $p(x) \geq 0$ for all x, and for a discrete density function,

$$\sum_{x \in \mathbb{Z}} p(x) = 1,$$

while for a continuous density function,

$$\int_{-\infty}^{+\infty} p(x)\,\mathrm{d}x = 1.$$

The corresponding cumulative distribution function

$$P(x) = \mathrm{Prob}(X \leq x)$$

is the probability that a randomly (according to the probability being considered) chosen value is less than or equal to x. This can be used to find the probability that a randomly chosen value is between two numbers:

$$\mathrm{Prob}(x < X \leq y) = P(y) - P(x).$$

Given a value h between 0 and 1, the inverse distribution function for a distribution takes h to the value of x for which $\texttt{Prob}(X \leq x) = h$.

20.4.2 Uniform distribution

The probability density function for the uniform distribution. Given two values a and b with $a < b$, the uniform distribution on $[a, b]$ has density function $1/(b-a)$ for x in $[a, b]$. The `uniform` (or `uniformd`) command computes this density function.

- `uniform` (or `uniformd`) takes three arguments:
  - a and b, real numbers with $a < b$.
  - x, a real number.
- `uniform(`a, b, x`)` (or `uniformd(`a, b, x`)`) returns the value of the probability density function for the uniform distribution from a to b, namely $1/(b-a)$.

Example

```
> uniform(2.2,3.5,2.8)
```

0.769230769231

The cumulative distribution function for the uniform distribution. The `uniform_cdf` or `uniformd_cdf` command finds the cumulative distribution function for the uniform distribution.

- `uniform_cdf` takes three mandatory arguments and one optional argument:
  - a and b, real numbers with $a < b$.
  - x, a real number.
  - Optionally y, a real number.
- `uniform_cdf(`a, b, x`)` returns the value of the cumulative distribution function for the uniform distribution from a to b, which in this case will be $(x-a)/(b-a)$.
- `uniform_cdf(`a, b, x, y`)` returns $\texttt{Prob}(x \leq X \leq y)$, which in this case will be $(y-x)/(b-a)$.

Examples

```
> uniform_cdf(2,4,3.2)
```

0.6

```
> uniform_cdf(2,4,3,3.2)
```

0.1

The inverse distribution function for the uniform distribution. The `uniform_icdf` or `uniformd_icdf` command computes the inverse distribution for the uniform distribution.

- `uniform_icdf` takes three arguments:
  - a and b, real numbers with $a < b$.
  - h, a real number between 0 and 1.
- `uniform_icdf(`a, b, h`)` returns the value of the inverse distribution function to the uniform distribution from a to b; namely the value of x for which $h = \text{Prob}(X \leq x_)$.

Example

```
> uniform_icdf(2,3,.6)
```

2.6

20.4.3 Binomial distribution

The probability density function for the binomial distribution. If you perform an experiment n times where the probability of success each time is p, then the probability of exactly k successes is:

$$\texttt{binomial}(n, k, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{20.1}$$

This determines the binomial distribution.

The `binomial` command computes the density function for the binomial distribution.

- `binomial` takes two mandatory arguments and one optional argument.
  - n, a positive integer.
  - k, a nonnegative integer less than or equal to n.
  - Optionally, p, a probability (a real number between 0 and 1).
- `binomial(`n, k`)` returns the binomial coefficient $\binom{n}{k}$ (see Section 12.1.2, p. 268), same as `comb(`n, k`)`.
- `binomial(`n, k, p`)` returns the probability given by (20.1).

Examples

```
> binomial(10,2)
```

or:

```
> comb(10,2)
```

45

```
> binomial(10,2,0.4)
```

0.120932352

The cumulative distribution function for the binomial distribution. The `binomial_cdf` command computes the cumulative distribution function for the binomial distribution.

- `binomial_cdf` takes three mandatory arguments and one optional argument:
  - n, a positive integer.
  - p, a probability (a real number between 0 and 1).
  - x, a real number.
  - Optionally, y, a real number.
- `binomial_cdf`(n, p, x) returns

$$\text{Prob}(X \leq x) = \text{binomial}(n, 0, p) + \cdots + \text{binomial}(n, \lfloor x \rfloor), p)$$

- `binomial_cdf`(n, p, x, y) returns

$$\text{Prob}(x \leq X \leq y) = \text{binomial}(n, \lceil x \rceil, p) + \cdots + \text{binomial}(n, \lfloor y \rfloor, p)$$

Examples

```
> binomial_cdf(4,0.5,2)
```

0.6875

```
> binomial_cdf(2,0.3,1,2)
```

0.51

The inverse distribution function for the binomial distribution. The `binomial_icdf` command computes the inverse distribution function for the binomial distribution.

- `binomial_icdf` takes three mandatory arguments and one optional argument:
  - n, a positive integer.
  - p, a probability (a real number between 0 and 1).
  - h, a real number between 0 and 1.
- `binomial_icdf`(n, p, h) returns the value of the inverse distribution for the binomial distribution with n trials and probability p; namely, the smallest value of x for which $\text{Prob}(X \leq x) \geq h$.

Example

```
> binomial_icdf(4,0.5,0.9)
```

3

Note that `binomial_cdf`$(4, 0.5, 3) = 0.9375$, which is bigger than 0.9, while `binomial_cdf`$(4, 0.5, 2) = 0.6875$, which is smaller than 0.9.

20.4.4 Negative binomial distribution

The probability density function for the negative binomial distribution. If you repeatedly perform an experiment with probability of success p, then, given an integer n, the probability of k failures that occur before you have n successes is given by the negative binomial distribution, which can be computed by

$$\binom{n+k-1}{k} p^n (1-p)^k. \tag{20.2}$$

The `negbinomial` command finds the density function for the negative binomial distribution.

- `negbinomial` takes three arguments:
  - n and k, integers.
  - p, a probability (a real number between 0 and 1).
- `negbinomial`(n, k, p) returns the value of the negative binomial distribution, given in (20.2).

Example

```
> negbinomial(4,2,0.5)
```

0.15625

Note that

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\dots(n-k+1)}{k!}.$$

The second formula makes sense even if n is negative, and you can write

$$\texttt{negbinomial}(n, k, p) = \binom{-n}{k} p^n (p-1)^k,$$

from which the name negative binomial distribution comes from. This also makes it simple to determine the mean $(n(1-p)/p)$ and variance $(n(1-p)/p^2)$. The negative binomial is also called the Pascal distribution (after Blaise Pascal) or the Pólya distribution (after George Pólya).

The cumulative distribution function for the negative binomial distribution. The `negbinomial_cdf` command finds the cumulative distribution function for the negative binomial distribution.

- `negbinomial_cdf` takes three mandatory arguments and two optional arguments:
  - n, an integer.
  - p, a probability (between 0 and 1).
  - x, a number.
  - Optionally, y, a number.

- `negbinomial_cdf(`n, p, x`)` returns

$$\text{Prob}(X \leq x) = \texttt{negbinomial}(n, 0, p) + \cdots + \texttt{negbinomial}(n, \lfloor x \rfloor, p).$$

- `negbinomial_cdf(`n, p, x, y`)` returns

$$\text{Prob}(x \leq X \leq y) = \texttt{negbinomial}(n, \lceil x \rceil, p) + \cdots + \texttt{negbinomial}(n, \lfloor y \rfloor, p)$$

Examples

```
> negbinomial_cdf(4,0.5,2)
```

0.34375

```
> negbinomial_cdf(4,0.5,2,5)
```

0.40234375

The inverse distribution function for the negative binomial distribution. The `negbinomial_icdf` command gives the inverse distribution function for the negative binomial distribution.

- `negbinomial_icdf` takes three arguments:
  - n, a positive integer.
  - p, a probability (a real number between 0 and 1).
  - h, a real number between 0 and 1.
- `negbinomial_icdf(`n, p, h`)` returns the value of the inverse distribution for the negative binomial distribution with n and probability p; namely, the smallest value of x for which $\text{Prob}(X \leq x) \geq h$.

Example

```
> negbinomial_icdf(4,0.5,0.9)
```

8

20.4.5 Multinomial distribution

If X follows a multinomial probability distribution with $P = [p_0, p_1, \ldots, p_j]$ (where $p_0 + \cdots + p_j = 1$), then for $K = [k_0, \ldots, k_j]$ with $k_0 + \cdots + k_j = n$, the probability that $X = K$ is given by

$$\frac{n!}{k_0! k_1! \cdots k_j!} p_0^{k_0} p_1^{k_1} \cdots p_j^{k_j}. \tag{20.3}$$

The `multinomial` command computes the density function for the multinomial distribution.

- `multinomial` takes three arguments:
  - n, an integer.
  - $P = [p_0, p_1, \ldots, p_j]$, a probability vector (i.e., $p_k \geq 0$ for all k and $p_0 + \cdots + p_j = 1$).
  - $K = [k_0, \ldots, k_j]$, a list of integers with $k_0 + \cdots + k_j = n$.
- `multinomial(`n, P, K`)` returns the probability that $X = K$, given in (20.3).

You will get an error if $k_0 + \cdots + k_j$ is not equal to n, although you won't get one if $p_0 + \cdots + p_j$ is not equal to 1.

Example

Suppose you make 10 choices, where each choice is one of three items; the first has a 0.2 probability of being chosen, the second a 0.3 probability and the third a 0.5 probability. The probability that you end up with 3 of the first item, 2 of the second and 5 of the third will be:

```
> multinomial(10,[0.2,0.3,0.5],[3,2,5])
```

0.0567

20.4.6 Poisson distribution

The probability density function for the Poisson distribution. Recall that for the Poisson distribution with parameter λ, the probability of a non-negative integer k is $e^{-\lambda}\frac{\lambda^k}{k!}$. This distribution has mean λ and variance λ.

The `poisson` command gives the density function for the Poisson distribution.

- `poisson` takes two arguments:
  - λ, a real number.
  - k, a non-negative integer.
- `poisson(`λ, k`)` returns the value of the Poisson probability density function with parameter λ at x, namely $e^{-\lambda}\frac{\lambda^k}{k!}$.

Example

```
> poisson(10.0,9)
```

0.125110035721

The cumulative distribution function for the Poisson distribution. The `poisson_cdf` command computes the cumulative distribution function for the Poisson distribution.

- `poisson_cdf` takes two arguments:
  - μ, a real number.
  - x, a real number.
  - Optionally, y, a real number.
- `poisson_cdf(`μ, x`)` returns

  $$\text{Prob}(X \leq x) = \texttt{poisson}(\mu, 0) + \cdots + \texttt{poisson}(\mu, \lfloor x \rfloor))$$

  for the Poisson distribution with parameter μ.
- `poisson_cdf(`μ, x, y`)` returns

  $$\text{Prob}(x \leq X \leq y) = \texttt{poisson}(\mu, \lceil x \rceil) + \cdots + \texttt{poisson}(\mu, \lfloor y \rfloor)).$$

Examples

```
> poisson_cdf(10.0,3)
```

0.0103360506759

```
> poisson_cdf(10.0,3,10)
```

0.580270354477

The inverse distribution function for the Poisson distribution. The `poisson_icdf` command finds the inverse distribution function for the Poisson distribution.

- `poisson_icdf` takes three arguments:
  - μ, a real number.
  - h, a real number between 0 and 1.
- `poisson_icdf(`μ, h`)` returns the value of the inverse distribution for the Poisson distribution with parameter μ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> poisson_icdf(10.0,0.975)
```

17

20.4.7 Normal distributions

The probability density function for a normal distribution. The density function of the normal distribution with mean μ and standard deviation σ at the point x is

$$\texttt{normald}(\mu, \sigma, x) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{(x-\mu)^2/2}. \tag{20.4}$$

The `normald` (or `loi_normal`) command finds the value of this density function.

- `normald` (or `loi_normal`) command takes two optional arguments and one mandatory argument:
  - Optionally, μ and σ, the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
  - x, a real number.
- `normald(`$\langle \mu, \sigma, \rangle x$`)` returns the value of the normal density function with parameter μ and standard deviation σ at the value x, given in (20.4).

Examples

```
> normald(2,1,3)
```

$$\frac{\mathrm{e}^{-\frac{1}{2}}}{\sqrt{2\pi}}$$

```
> normald(2)
```

$$\frac{1}{\sqrt{2\pi}\mathrm{e}^2}$$

The cumulative distribution function for normal distribution. The `normal_cdf` (or `normald_cdf`) command computes the cumulative distribution function for the normal distribution.

- `normal_cdf` (or `normald_cdf`) takes three optional arguments and one mandatory argument:
  - Optionally, μ and σ, the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
  - x, a real number.
  - Optionally, y, a real number.
- `normal_cdf(`$\langle\mu, \sigma, \rangle x$`)` returns $\text{Prob}(X \leq x)$ for the normal distribution with mean μ and standard deviation σ.
- `normal_cdf(`$\langle\mu, \sigma, \rangle x, y$`)` returns $\text{Prob}(x \leq X \leq y)$.

Examples

```
> normal_cdf(1,2,1.96)
```

0.684386303484

```
> normal_cdf(1,2.1,1.2)
```

0.537937144066

```
> normal_cdf(1,2.1,1.2,9)
```

0.461993238584

The inverse distribution function for normal distribution. The `normal_icdf` (or `normald_icdf`) command computes the inverse distribution for the normal distribution.

- `normal_icdf` (or `normald_icdf`) takes two optional arguments and one mandatory argument:
  - Optionally, μ and σ, the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
  - h, a real number.
- `normal_icdf(`$\langle\mu, \sigma, \rangle h$`)` returns the inverse distribution for the normal distribution with mean μ and standard deviation σ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Examples

```
> normal_icdf(0.975)
```

1.95996398454

```
> normal_icdf(1,2,0.495)
```

0.974933060984

The upper tail cumulative function for normal distributions. The `UTPN` (the Upper Tail Probability-Normal distribution) computes Prob$(X > x)$ for a normal distribution.

- `UTPN` takes two optional arguments and one mandatory argument:
  - Optionally, μ and σ^2, the mean variance deviation. (By default, $\mu = 0$ and $\sigma^2 = 1$, giving the standard normal distribution.)

    Note that, unlike `normald` and `normal_cdf`, the `UTPN` takes the variance and not the standard deviation.
  - x, a real number.
- `UTPN(`$\langle \mu, \sigma^2, \rangle x$`)` returns Prob$(X > x)$, for the normal distribution with mean μ and variance σ^2.

Examples

```
> UTPN(1.96)
```

0.0249978951482

```
> UTPN(1,4,1.96)
```

0.315613696516

20.4.8 Student's distribution

The probability density function for Student's distribution. Student's distribution (also called Student's t-distribution or the t-distribution) with n degrees of freedom has density function given by

$$\texttt{student}(n, x) = \frac{\Gamma((n+1)/2)}{\Gamma(n/2)\sqrt{n\pi}} \left(1 + \frac{x^2}{n}\right)^{-n-1/2} \tag{20.5}$$

where recall the Gamma function (see Section 7.3.13, p. 136) is defined for $x > 0$ by

$$\Gamma(x) = \int_0^{+\infty} \mathrm{e}^{-t} t^{x-1} \, \mathrm{d}x.$$

The `student` or `studentd` command finds the density function for Student's distribution.

- `student` takes two arguments:
  - n, an integer (the degrees of freedom).
  - x, a real number.

  `student(`n, x`)` returns the value of the density function for Student's distribution with n degrees of freedom at x, given in (20.5).

Example

```
> student(2,3)
```

$$\frac{2\sqrt{\pi}\sqrt{\frac{11}{2}}^{-1}}{2\sqrt{2\pi} \cdot 11}$$

which can be numerically approximated by:

```
> evalf(student(2,3))
```

0.0274101222343

The cumulative distribution function for Student's distribution. The `student_cdf` command computes the cumulative distribution function for Student's distribution.

- `student_cdf` takes two mandatory arguments and one optional argument.
  - n, an integer (the degrees of freedom).
  - x, a real number.
  - Optionally, y, a real number.
- `student_cdf`(n, x) returns Prob$(X \leq x)$ for Student's distribution with n degrees of freedom.
- `student_cdf`(n, x, y) returns Prob$(x \leq X \leq y)$.

Examples

```
> student_cdf(5,2)
```

0.949030260585

```
> student_cdf(5,-2,2)
```

0.89806052117

The inverse distribution function for Student's distribution. The `student_icdf` command computes the inverse distribution for Student's distribution.

- `student_icdf` takes two arguments:
  - n, an integer (the degrees of freedom).
  - h, a real number between 0 and 1.
- `student_icdf`(n, h) returns the inverse distribution for Student's distribution with n degrees of freedom; namely, the value of x for which Prob$(X \leq x) = h$.

Example

```
> student_icdf(5,0.95)
```

2.01504837333

The upper tail cumulative function for Student's distribution. The `UTPT` (the Upper Tail Probability-T distribution) computes Prob$(X > x)$ for Student's distribution.

- `UTPT` takes two arguments:
  - n, an integer (the degrees of freedom).
  - x, a real number.
- `UTPT`(n, x) returns Prob$(X > x)$ for Student's distribution with n degrees of freedom.

Example

```
> UTPT(5,2)
```

0.0509697394149

20.4.9 χ^2 distribution

The probability density function for the χ^2 distribution. The χ^2 distribution with n degrees of freedom has density function given by

$$\chi^2(n,x) = \frac{x^{n/2-1}\mathrm{e}^{-x/2}}{2^{n/2}\Gamma(n/2)}. \tag{20.6}$$

The `chisquare` command computes this density function.

- `chisquare` takes two arguments:
  - n, an integer (the degrees of freedom).
  - x, a real number.
- `chisquare`(n,x) returns the value of the χ^2 density function with n degrees of freedom, given in (20.6).

Example

```
> chisquare(5,2)
```

$$\frac{2\sqrt{2}}{\mathrm{e}\left(\frac{3}{4}\sqrt{\pi}\sqrt{2}\cdot 2^2\right)}$$

which can be numerically approximated by:

```
> evalf(chisquare(5,2))
```

$$0.138369165807$$

The cumulative distribution function for the χ^2 distribution. The `chisquare_cdf` command computes the cumulative distribution function for the χ^2 distribution.

- `chisquare_cdf` takes two mandatory arguments and one optional argument:
  - n, an integer (the degrees of freedom).
  - x, a real number.
  - Optionally, y, a real number.
- `chisquare_cdf`(n,x) returns $\mathrm{Prob}(X \le x)$ for the χ^2 distribution with n degrees of freedom.
- `chisquare_cdf`(n,x,y) returns $\mathrm{Prob}(x \le X < y)$.

Examples

```
> chisquare_cdf(5,11)
```

$$0.948620016517$$

```
> chisquare_cdf(3,1,2)
```

$$0.22884525243$$

The inverse distribution function for the χ^2 distribution. The `chisquare_icdf` command computes the inverse distribution for the χ^2 distribution.

- `chisquare_icdf` takes two arguments:
  - n, an integer (the degrees of freedom).
  - h, a real number between 0 and 1.
- `chisquare_icdf`(n, h) returns the inverse distribution for the χ^2 distribution with n degrees of freedom; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> chisquare_icdf(5,0.95)
```

11.0704976935

The upper tail cumulative function for the χ^2 distribution. The `UTPC` (the Upper Tail Probability-Chi-square distribution) computes $\text{Prob}(X > x)$ for the χ^2 distribution.

- `UTPC` takes two arguments:
  - n, an integer (the degrees of freedom).
  - x, a real number.
- `UTPC`(n, x) returns $\text{Prob}(X > x)$ for the χ^2 distribution with n degrees of freedom.

Example

```
> UTPC(5,11)
```

0.0513799834831

20.4.10 Fisher-Snédécor distribution

The probability density function for the Fisher-Snédécor distribution. The Fisher-Snédécor distribution (also called the F-distribution) with n_1 and n_2 degrees of freedom has density function given by, for $x \geq 0$,

$$\texttt{fisher}(n_1, n_2, x) = \frac{(n_1/n_2)^{n_1/2}\Gamma((n_1+n_2)/2)}{\Gamma(n_1/2)\Gamma(n_2/2)} \cdot \frac{x^{(n_1-2)/2}}{(1+(n_1/n_2)x)^{(n_1+n_2)/2}}. \tag{20.7}$$

The `fisher` or `fisherd` or `snedecor` command computes this density function.

- `fisher` takes three arguments:
  - n_1 and n_2, integers (the degrees of freedom).
  - x, a non-negative real number.
- `fisher`(n_1, n_2, x) returns the value of the Fisher-Snédécor density function with n_1 and n_2 degrees of freedom, given in (20.7).

Example

```
> fisher(5,3,2.5)
```

0.10131184472

The cumulative distribution function for the Fisher-Snédécor distribution. The `fisher_cdf` (or `snedecor_cdf`) command computes the cumulative distribution function for the Fisher-Snédécor distribution.

- `fisher_cdf` takes three mandatory arguments and one optional argument:
  - n_1 and n_2, integers (the degrees of freedom).
  - x, a real number.
  - Optionally, y, a real number.
- `fisher_cdf`(n_1, n_2, x) returns $\text{Prob}(X \leq x)$ for the Fisher-Snédécor distribution with n_1 and n_2 degrees of freedom
- `fisher_cdf`(n_1, n_2, x, y) returns $\text{Prob}(x \leq X < y)$.

Examples

```
> fisher_cdf(5,3,9)
```

$$\beta\left(\frac{5}{2}, \frac{3}{2}, \frac{15}{16}, 1\right)$$

(See Section 7.3.16, p. 138.) This can be numerically approximated with:

```
> evalf(fisher_cdf(5,3,9))
```

$$0.949898927032$$

```
> evalf(fisher_cdf(5,3,9,10))
```

$$0.0066824173023$$

The inverse distribution function for the Fisher-Snédécor distribution. The `fisher_icdf` (or `snedecor_icdf`) command computes the inverse distribution for the Fisher-Snédécor distribution.

- `fisher_icdf` takes three arguments:
  - n_1 and n_2, integers (the degrees of freedom).
  - h, a real number between 0 and 1.
- `fisher_icdf`(n_1, n_2, h) returns the inverse distribution for the Fisher-Snédécor distribution with n_1 and n_2 degrees of freedom; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> fisher_icdf(5,3,0.95)
```

$$9.01345516752$$

The upper tail cumulative function for the Fisher-Snédécor distribution. The `UTPF` (the Upper Tail Probability-Fisher-Snédécor distribution) computes $\text{Prob}(X > x)$ for the Fisher-Snédécor distribution.

- `UTPF` takes three arguments:
  - n_1 and n_2, integers (the degrees of freedom).
  - x, a real number.
- `UTPF`(n_1, n_2, x) returns $\text{Prob}(X > x)$ for the Fisher-Snédécor distribution with n_1 and n_2 degrees of freedom.

Example

```
> UTPF(5,3,9)
```

$$0.050101072968$$

20.4.11 Gamma distribution

The probability density function for the gamma distribution. The gamma distribution depends on two parameters, $a > 0$ and $b > 0$; the value of the density function at $x \geq 0$ is

$$\texttt{gammad}(a, b, x) = x^{a-1}\mathrm{e}^{-bx}b^a/\Gamma(a) \tag{20.8}$$

The `gammad` command computes this density function.

- `gammad` takes three arguments:
  - a and b, positive real numbers (the parameters).
  - x, a real number.
- `gammad(`a, b, x`)` returns the value of the gamma density function with parameters a and b, given in (20.8).

Example

```
> gammad(2,1,3)
```

$$\frac{3}{\mathrm{e}^3}$$

The cumulative distribution function for the gamma distribution. The `gamma_cdf` command computes the cumulative distribution function for the gamma distribution.

- `gamma_cdf` takes three mandatory arguments and one optional argument:
  - a and b, real numbers (the parameters).
  - x, a real number.
  - Optionally, y, a real number.
- `gamma_cdf(`a, b, x`)` returns $\mathrm{Prob}(X \leq x)$ for the gamma distribution with parameters a and b.
- `gamma_cdf(`n, x, y`)` returns $\mathrm{Prob}(x \leq X \leq y)$.

It turns out that

$$\texttt{gammad\_cdf}(n, x) = \texttt{igamma}(a, bx, 1),$$

where `igamma` is the incomplete gamma function (see Section 7.3.15, p. 138),

$$\texttt{igamma}(a, x, 1) = \frac{1}{\Gamma(a)}\int_0^x \mathrm{e}^{-t}t^{a-1}\mathrm{d}t.$$

Examples

```
> gammad_cdf(2,1,0.5)
```

0.090204010431

```
> gammad_cdf(2,1,0.5,1.5)
```

0.351970589198

The inverse distribution function for the gamma distribution. The `gammad_icdf` command computes the inverse distribution for the gamma distribution.

- `gamma_icdf` takes three arguments:
  - a and b, numbers (the parameters).
  - h, a real number between 0 and 1.
- `gamma_icdf`(a, b, h) returns the inverse distribution for the gamma distribution with parameters a and b; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> gammad_icdf(2,1,0.5)
```

1.67834699002

20.4.12 Beta distribution

The probability density function for the beta distribution. The beta distribution depends on two parameters, $a > 0$ and $b > 0$; the value of the density function at x in $[0, 1]$ is (see Section 7.3.13, p. 136):

$$\texttt{betad}(a, b, x) = \frac{\Gamma(a+b)x^{a-1}(1-x)^{b-1}}{\Gamma(a)\Gamma(b)}. \tag{20.9}$$

The `betad` command computes the density function for the beta distribution.

- `betad` takes three arguments:
  - a and b, positive numbers, the parameters.
  - x, a real number.
- `betad`(a, b, x) returns the value of the density function for the beta distribution with parameters a and b, given in (20.9).

Example

```
> betad(2,1,0.3)
```

0.6

The cumulative distribution function for the beta distribution. The `betad_cdf` command computes the cumulative distribution function for the beta distribution.

- `beta_cdf` takes three mandatory arguments and one optional argument:
  - a and b, real numbers (the parameters).
  - x, a real number.
  - Optionally, y, a real number.
- `betad_cdf`(a, b, x) returns $\text{Prob}(X \leq x)$ for the beta distribution with parameters a and b.
- `beta_cdf`(n, x, y) returns $\text{Prob}(x \leq X \leq y)$.

It turns out that `betad_cdf`$(a, b, x) = \frac{\beta(a,b,x)\Gamma(a+b)}{\Gamma(a)\Gamma(b)}$, where $\beta(a, b, x) = \int_0^x t^{a-1}(1-t)^{b-1}\,\mathrm{d}t$ (see Section 7.3.16, p. 138).

Examples

```
> betad_cdf(2,3,0.2)
```

0.1808

```
> betad_cdf(2,3,0.25,0.5)
```

0.42578125

The inverse distribution function for the beta distribution. The `betad_icdf` command computes the inverse distribution for the beta distribution.

- `beta_icdf` takes three arguments:
  - a and b, real numbers (the parameters).
  - h, a real number between 0 and 1.
- `beta_icdf`(a, b, h) returns the inverse distribution for the beta distribution with parameters a and b; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> betad_icdf(2,3,0.2)
```

0.212317128278

20.4.13 Geometric distribution

The probability density function for the geometric distribution. If an experiment with probability of success p is iterated, the probability that the first success occurs on the kth trial is $(1-p)^{k-1}p$. This gives the geometric distribution (with parameter p) on the natural numbers. Given such a p, the geometric density function at n is given by

$$\texttt{geometric}(p, n) = (1-p)^{n-1}p \tag{20.10}$$

The `geometric` command computes this density function.

- `geometric` takes two arguments:
  - p, a probability (a number between 0 and 1).
  - x, a real number.

  `geometric(`p, x`)` returns the value of the geometric density function with probability p, given in (20.10).

Example

```
> geometric(0.2,3)
```

0.128

The cumulative distribution function of the geometric distribution. The `geometric_cdf` command computes the cumulative distribution function for the geometric distribution.

- `geometric_cdf` takes three mandatory arguments and one optional argument:
  - p, a probability (a number between 0 and 1).
  - n, a natural number.
  - Optionally, k, a natural number.
- `betad_cdf(`p, n`)` returns $\text{Prob}(X \leq n)$ for the geometric distribution with probability p.
- `beta_cdf(`p, n, k`)` returns $\text{Prob}(n \leq X \leq k)$.

It turns out that `geometric_cdf`$(p, n) = 1 - (1 - p)^n$.

Examples

```
> geometric_cdf(0.2,3)
```

0.488

```
> geometric_cdf(0.2,3,5)
```

0.31232

The inverse distribution function for the geometric distribution. The `geometric_icdf` command computes the inverse distribution for the geometric distribution.

- `geometric_icdf` takes two arguments:
  - p, a probability (a number between 0 and 1).
  - h, a real number between 0 and 1.
- `geometric_icdf(`a, b, h`)` returns the inverse distribution for the geometric distribution with probability p; namely, the smallest natural number n for which $\text{Prob}(X \leq n) \geq h$.

Example

```
> geometric_icdf(0.2,0.5)
```

4

20.4.14 Cauchy distribution

The probability density function for the Cauchy distribution. The `cauchy` (or `cauchyd`) command computes the probability density function for the Cauchy distribution (sometimes called the Lorentz distribution).

- `cauchy` takes two optional arguments and one mandatory argument:
  - Optionally, a and b, real numbers (the parameters; by default $a = 0$ and $b = 1$).
  - x, a real number.
- `cauchy(`$\langle a, b, \rangle x$`)` returns the value of the density function at x, namely $\frac{b/\pi}{(x-a)^2+b^2}$.

Examples

```
> cauchy(2.2,1.5,0.8)
```

0.113412073462

```
> cauchy(0.3)
```

0.292027418517

The cumulative distribution function for the Cauchy distribution. The `cauchy_cdf` (or `cauchyd_cdf`) command computes the cumulative distribution function for the Cauchy distribution.

- `cauchy_cdf` (or `cauchyd_cdf`) takes three optional arguments and one mandatory argument:
  - Optionally, a and b, the parameters (by default, $a = 0$ and $b = 1$).
  - x, a real number.
  - Optionally, y, a real number.
- `cauchy_cdf(`$\langle a, b, \rangle x$`)` returns $\text{Prob}(X \leq x)$ for the Cauchy distribution with parameters a and b.
- `cauchy_cdf(`$\langle a, b, \rangle x, y$`)` returns $\text{Prob}(x \leq X \leq y)$.

It turns out that `cauchy_cdf`$(a, b, x) = \frac{1}{2} + \frac{1}{\pi} \arctan \frac{x-a}{b}$.

Examples

```
> cauchy_cdf(2,3,1.4)
```

0.437167041811

```
> cauchy_cdf(1.4)
```

0.802568456711

```
> cauchy_cdf(2,3,-1.9,1.4)
```

0.228452641651

The inverse distribution function for the Cauchy distribution. The `cauchy_icdf` (or `cauchyd_icdf`) command computes the inverse distribution for the Cauchy distribution.

- `cauchy_icdf` (or `cauchyd_icdf`) takes two optional arguments and one mandatory argument:
  - Optionally, a and b, parameters (by default, $a = 0$ and $b = 1$).
  - h, a real number between 0 and 1.
- `cauchy_icdf([`a, b`,] `h`)` returns the inverse distribution for the Cauchy distribution with parameters a and b; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> cauchy_icdf(2,3,0.23)
```

$$-1.40283204777$$

20.4.15 Exponential distribution

The probability density function for the exponential distribution. The exponential distribution depends on one parameters, $\lambda > 0$; the value of the density function at $x \geq 0$ is `exponential`$(\lambda, x) = \lambda e^{-\lambda x}$. The `exponential` or `exponentiald` command computes the exponential distribution.

- `exponential` takes two arguments:
  - λ, a positive number (the parameter).
  - x, a positive number.
- `exponential(`λ, x`)` returns the value of the exponential density function with parameter λ at x; namely, `exponential`$(\lambda, x) = \lambda e^{-\lambda x}$.

Example

```
> exponential(2.1,3.5)
```

$$0.00134944395675$$

The cumulative distribution function for the exponential distribution. The `exponential_cdf` (or `exponentiald_cdf`) command computes the cumulative distribution function for the exponential distribution.

- `exponential_cdf` (or `exponentiald_cdf`) takes two arguments:
  - λ, a positive number (the parameter).
  - x, a positive number.
- `exponential_cdf(`λ, x`)` returns $\text{Prob}(X \leq x)$ for the exponential distribution with parameter λ.
- `exponential_cdf(`λ, x, y`)` returns $\text{Prob}(x \leq X \leq y)$.

Examples

```
> exponential_cdf(2.3,3.2)
```

0.99936380154

```
> exponential_cdf(2.3,0.9,3.2)
```

0.125549583246

The inverse distribution function for the exponential distribution. The `exponential_icdf` (or `exponentiald_icdf`) command computes the inverse distribution for the exponential distribution.

- `exponential_icdf` (or `exponentiald_icdf`) takes two arguments:
  - λ, a positive number (the parameter).
  - h, a positive real number.
- `exponential_icdf(`λ, h`)` returns the inverse distribution for the exponential distribution with parameter λ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> exponential_icdf(2.3,0.87)
```

0.887052534142

20.4.16 Weibull distribution

The probability density function for the Weibull distribution. The Weibull distribution depends on three parameters; $k > 0$, $\lambda > 0$ and a real number θ. The probability density at x is given by

$$\frac{k}{\lambda}\left(\frac{x-\theta}{\lambda}\right)^2 \mathrm{e}^{-(x-\theta)^2\lambda^2}. \tag{20.11}$$

The `weibull` or `weibulld` command compute this density function.

- `weibull` takes three mandatory and one optional argument:
  - k, a positive integer.
  - λ, a positive real number.
  - Optionally θ, a real number (by default 0).
  - x, a real number.
- `weibull(`$k, \lambda\,\langle, \theta\,\rangle, x$`)` returns the value of the Weibull density function, given in (20.11).

Example

```
> weibull(2,1,3)
```

or:

```
> weibull(2,1,0,3)
```

$$\frac{6}{\mathrm{e}^9}$$

The cumulative distribution function for the Weibull distribution. The `weibull_cdf` (or `weibulld_cdf`) command computes the cumulative distribution function for the Weibull distribution.

- `weibull_cdf` (or `weibulld_cdf`) takes three mandatory arguments and two optional arguments:
  - k, a positive integer.
  - λ, a positive real number.
  - Optionally θ, a real number (by default 0).
  - x, a real number.
  - Optionally, y, a real number. If this optional argument is included, then θ must also be included.
- `weibull_cdf`$(k, \lambda\,\langle, \theta\,\rangle, x)$ returns $\text{Prob}(X \leq x)$ for the Weibull distribution with parameters k, λ and θ.
- `weibull_cdf`$(k, \lambda\,\langle, \theta\,\rangle, x, y)$ returns $\text{Prob}(x \leq X \leq y)$.

In this case, the Weibull cumulative distribution function is given by the formula `weibull_cdf`$(k, \lambda, \theta, x) = 1 - \mathrm{e}^{-(x-\theta)^2\lambda^2}$.

Examples

```
> weibull_cdf(2,3,5)
```

or:

```
> weibull_cdf(2,3,0,5)
```

$$1 - \mathrm{e}^{-\frac{25}{9}}$$

```
> weibull_cdf(2.2,1.5,0.4,1.9)
```

$$0.632120558829$$

```
> weibull_cdf(2.2,1.5,0.4,1.2,1.9)
```

$$0.410267239944$$

The inverse distribution function for the Weibull distribution. The `weibull_icdf` (or `weibulld_icdf`) command computes the inverse distribution for the Weibull distribution.

- `weibull_icdf` (or `weibulld_icdf`) takes three mandatory arguments and one optional argument:
  - k, a positive integer.
  - λ, a positive real number.
  - Optionally θ, a real number (by default 0).
  - h, a real number between 0 and 1.
- `weibull_icdf`$(k, \lambda\,\langle, \theta\,\rangle, h)$ returns the inverse distribution for the Weibull distribution with parameters k, λ and θ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example

```
> weibull_icdf(2.2,1.5,0.4,0.632)
```

$$1.89977657604$$

20.4.17 Kolmogorov-Smirnov distribution

The density function for the Kolmogorov-Smirnov distribution is given by

$$\texttt{kolmogorovd}(x) = 1 - 2\sum_{k=1}^{\infty}(-1)^{k-1}\mathrm{e}^{-k^2x^2} \tag{20.12}$$

The `kolmogorovd` command computes this density function.

- `kolmogorovd` takes x, a real number.
- `kolmogorovd(`x`)` returns the density function of the Kolmogorov-Smirnov distribution at x, given by (20.12).

Example

```
> kolmogorovd(1.36)
```

$$0.950514123245$$

20.4.18 Wilcoxon or Mann-Whitney distribution

The Wilcoxon test polynomial. The `wilcoxonp` command computes the polynomial for the Wilcoxon or Mann-Whitney test.

- `wilcoxonp` takes one mandatory argument and one optional argument:
  - n, an integer.
  - Optionally, k an integer.
- `wilcoxonp(`$n\,\langle,k\rangle$`)` returns the polynomial for the Wilcoxon test.

Examples

```
> wilcoxonp(4)
```

$$[\frac{1}{16}, \frac{1}{16}, \frac{1}{16}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{16}, \frac{1}{16}, \frac{1}{16}]$$

```
> wilcoxonp(4,3)
```

$$[\frac{1}{35}, \frac{1}{35}, \frac{2}{35}, \frac{3}{35}, \frac{4}{35}, \frac{4}{35}, \frac{1}{7}, \frac{4}{35}, \frac{4}{35}, \frac{3}{35}, \frac{2}{35}, \frac{1}{35}, \frac{1}{35}]$$

The Wilcoxon/Mann-Whitney statistic. The `wilcoxons` command computes the Wilcoxon or Mann-Whitney statistic.

- `wilcoxons` takes two arguments:
  - L, a list.
  - M, either a list or a real number (a median).
- `wilcoxons(`L, M`)` returns the Wilcoxon statistic.

Examples

```
> wilcoxons([1,3,4,5,7,8,8,12,15,17],10)
```

18

```
> wilcoxons([1,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20])
```

128.5

The Wilcoxon or Mann-Whitney test. The `wilcoxont` command will perform the Wilcoxon or Mann-Whitney test.

- `wilcoxont` takes two mandatory arguments and two optional arguments:
  - L, a sample (list).
  - M, either another sample or a number (a median).
  - Optionally, f, a function.
  - Optionally, x, a real number.
- `wilcoxont(`$L, M\ \langle, f, x\rangle$`)` returns the results of the Wilcoxon test.

Examples

```
> wilcoxont([1,2,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20])
```

```
Mann-Whitney 2-sample test, H0 same Median, H1 <>
ranksum 93.0, shifted ranksum 27.0
u1=83, u2=27, u=min(u1,u2)=27
Limit value to reject H0 26
P-value 9055/176358 (0.0513444244094), alpha=0.05 H0 not rejected
```

1

```
> wilcoxont([1,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20],0.3)
```

```
Mann-Whitney 2-sample test, H0 same Median, H1 <>
ranksum 81.5, shifted ranksum 26.5
u1=73.5, u2=26.5, u=min(u1,u2)=26.5
Limit value to reject H0 35
P-value 316/4199 (0.0752560133365), alpha=0.3 H0 rejected
```

0

```
> wilcoxont([1,3,4,5,7,8,8,12,15,17],10,`>`,0.05)
```

```
Wilcoxon 1-sample test, H0 Median=10, H1 M<>10
Wilcoxon statistic: 18, p-value: 0.375, confidence level: 0.05
```

1

20.4.19 Moment generating functions for probability distributions

The `mgf` command finds the moment generating function for a probability distribution (such as normal, binomial, poisson, beta, gamma).

- `mgf` takes one or more mandatory arguments:
  - *distd*, the name of the function that finds the distribution's density function.
  - *params*, any parameters that would normally be passed to *distd*.
- `mgf(`*distd*, *params*`)` returns an expression for the moment generating function for *distd* with parameters *params*.

Examples

Find the moment generating function for the standard normal distribution:

```
> mgf(normald,1,0)
```

$$e^t$$

Find the moment generating function for the binomial distribution:

```
> mgf(binomial,n,p)
```

$$\left(1-p+pe^t\right)^n$$

20.4.20 Cumulative distribution functions

The `cdf` command finds the cumulative distribution function for a probability distribution.

- `cdf` takes one or more mandatory arguments and one optional argument.
  - *distd*, the name of the function that finds the distribution's density function.
  - *params*, any parameters that would normally be passed to *distd*.
  - x, a number.
- `cdf(`*distd*, *params*`)` returns an expression for the cumulative distribution function for *distd* with parameters *params*.
- `cdf(`*distd*, *params*, x`)` returns the value of the cumulative distribution function at x.

Examples

```
> cdf(normald,0,1)
```

$$\frac{\operatorname{erf}\left(\frac{1}{2}x\sqrt{2}\right)+1}{2}$$

```
> cdf(binomial,10,0.5,4)
```

$$0.376953125$$

20.4.21 Inverse distribution functions

The `icdf` command finds the inverse cumulative distribution function for a probability distribution.

- `cdf` takes one or more mandatory arguments and one optional argument.
  - *distd*, the name of the function that finds the distribution's density function.
  - *params*, any parameters that would normally be passed to *distd*.
  - x, a number.
- `icdf(`*distd*, *params*`)` returns an expression for the inverse cumulative distribution function for *distd* with parameters *params*.
- `icdf(`*distd*, *params*, x`)` returns the value of the inverse cumulative distribution function at x.

Example

```
> icdf(normald,0,0.5,0.975)
```

0.97998199227

20.4.22 Kernel density estimation

The `kernel_density` or `kde` command performs kernel density estimation (KDE)[1]. `kernel_density` takes a sample, optionally restricted to an interval $[a, b]$, and obtains an estimate $\hat{f}$ of the (unknown) probability density function f from which the samples are drawn. The function $\hat{f}$ is defined by:

$$\hat{f}(x) = \frac{1}{n\,h} \sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right), \tag{20.13}$$

where K is the Gaussian kernel

$$K(u) = \frac{1}{\sqrt{2\,\pi}} \exp\left(-\frac{1}{2}\,u^2\right)$$

and h is the positive real parameter called the *bandwidth*.

- `kernel_density` takes one mandatory argument and a sequence of optional arguments:
  - L, a list of samples $L = [X_1, X_2, \ldots, X_n]$.
  - Optionally, a sequence of options from:
    - `output=`*type* (or `Output=`*type* to specify the form of the return value $\hat{f}$, where *type* can be one of:
      - `exact`, to return $\hat{f}$ as the sum of Gaussian kernels, i.e. as the right side of (20.13), which is usable only when the number of samples is relatively small (up to few hundreds).
      - `piecewise`, to return $\hat{f}$ as a piecewise expression obtained by the spline interpolation of the specified degree (by default, the interpolation is linear) on the interval $[a, b]$ segmented to the specified number of bins.
      - `list` (the default), to return $\hat{f}$ in discrete form, as a list of values $\hat{f}\left(a + k\,\frac{b-a}{M-1}\right)$ for $k = 0, 1, \ldots, M$, where M is the number of bins.

[1] For the details on kernel density estimation and its implementation see: Artur Gramacki, *Nonparametric Kernel Density Estimation and Its Computational Aspects*, Springer, 2018.

  - `bandwidth=`*value*, to specify the bandwidth. *value* can be one of:
    - h, a positive real number.
    - `select` (the default), to have the bandwidth selected using a direct plug-in method,
    - `gauss` (or `normal` or `normald`) to use Silverman's rule of thumb for selecting bandwidth (this method is fast but the results are close to optimal ones only when f is approximately normal).
  - `bins=`n for a positive integer n (by default 100), the number of bins for simplifying the input data. Only the number if samples in each bin is stored. Bins represent the elements of an equidistant segmentation of the interval S on which KDE is performed. This allows evaluating kernel summations using convolution when `output` is set to `piecewise` or `list`, which significantly lowers the computational burden for large values of n (say, few hundreds or more). If `output` is set to `exact`, this option is ignored.
  - $a..b$ or `range=`$[a,b]$ or `x=`$a..b$ for real numbers a and b, to specify the interval $[a,b]$ on which KDE is performed. If an identifier x is specified, it is used as the variable of the output. If the range endpoints are not specified, they are set to $a = \min_{1\le i\le n} X_i - 3\,h$ and $b = \max_{1\le i\le n} X_i + 3\,h$ (unless `output` is set to `exact`, in which case this option is ignored).
  - `interp=`n for an integer n (by default 1), which specifies the degree of the spline interpolation, ignored unless `output` is set to `piecewise`.
  - `spline=`n for an integer n, which sets `option` to `piecewise` and `interp` to n.
  - `eval=`x_0 to only return the value $\hat{f}(x_0)$ (this cannot be used with `output` set to `list`).
  - x, an unassigned identifier (by default `x`) to use as the variable of the output.
  - `exact`, the same as `output=exact`.
  - `piecewise`, the same as `output=piecewise`.

- `kernel_density(`L $\langle$, *options*$\rangle$`)` returns the function $\hat{f}$ given in (20.13).

Examples

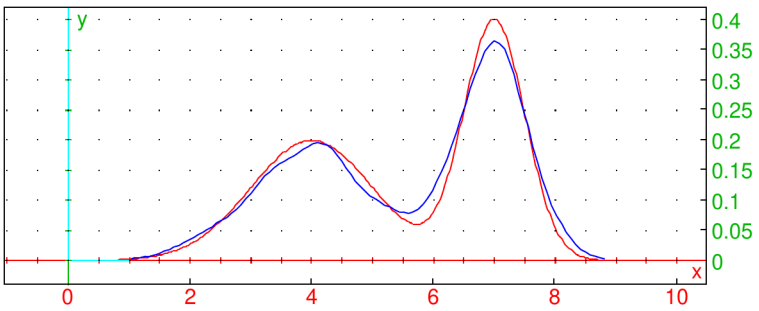
```
> kernel_density([1,2,3,2],bandwidth=1/4,exact)
```

$$\frac{e^{-\frac{(x-1.0)^2}{0.125}} + e^{-\frac{(x-2.0)^2}{0.125}} + e^{-\frac{(x-3.0)^2}{0.125}} + e^{-\frac{(x-2.0)^2}{0.125}}}{2.50662827463}$$

```
> f:=unapply(normald(4,1,x)/2+normald(7,1/2,x)/2,x)
```

$$x \mapsto \frac{\frac{1}{\sqrt{2\pi}} e^{-\frac{(x-4)^2}{2}}}{2} + \frac{\frac{1}{\sqrt{\frac{2\pi}{4}}} e^{-2(x-7)^2}}{2}$$

```
> X:=randvar(f,range=0..10,1000):;
  S:=sample(X,1000):;
  F:=kernel_density(S,piecewise):;
  plotfunc([f(x),F],x=0..10,color=[red,blue])
```



The exact density is drawn in red.

```
> kernel_density(S,bins=50,spline=3,eval=4.75)
```

0.14655478136

```
> time(kernel_density(sample(X,1e5),piecewise))
```

[0.17, 0.1653323]

```
> S:=sample(X,5000):;
  sqrt(int((f(x)-kde(S,piecewise))^2,x=0..10))
```

0.0269841239243

```
> S:=sample(X,25000):;
  sqrt(int((f(x)-kde(S,bins=150,piecewise))^2,x=0..10))
```

0.0144212781377

20.4.23 Distribution fitting by maximum likelihood

The `fitdistr` command finds the parameters for a distribution of a specified type that best fits a set of samples.

- `fitdistr` takes two arguments:
  - L, a list of presumably independent and identically distributed samples.
  - *distr*, a distribution type, which can be one of:
    - `normal` or `normald`, for a normal distribution.
    - `exp`, `exponential` of `exponentiald`, for an exponential distribution.
    - `poisson`, for a Poisson distribution.
    - `geometric`, for a geometric distribution.
    - `gammad`, for a gamma distribution.
    - `betad`, for a beta distribution.
    - `cauchy` or `cauchyd`, for a Cauchy distribution.
    - `weibull` or `weibulld` for a Weibull distribution.
- `fitdistr`(L, *distr*) returns the name of the specified type of distribution with parameters that fit L most closely according to the method of maximum likelihood.

Examples

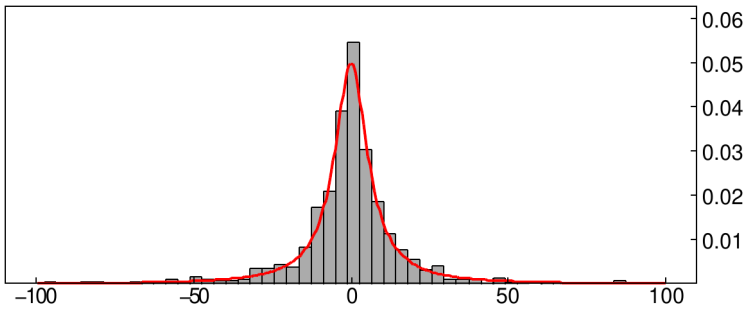
```
> fitdistr(randvector(1000,weibulld,1/2,1),weibull)
```

weibulld (0.517079036032, 1.05683817484)

```
> X:=randvar(normal,stddev=9.5):;
  Y:=randvar(normal,stddev=1.5):;
  S:=sample(eval(X/Y,0),1000):;
  Z:=fitdistr(S,cauchy)
```

cauchyd (0.347058460176, 6.55905486387)

```
> histogram(select(x->(x>-100 and x<100),S));
  plot(Z(x),x=-100..100,display=red+line_width_2)
```




```
> kolmogorovt(S,Z)
```

["D=", 0.0161467485236, "K=", 0.510605021406, "1-kolmogorovd(K)=", 0.956753826255]

The Kolmogorov-Smirnov test indicates that the samples from S are drawn from Z with high probability.

You can fit a lognormal distribution to samples $x_1, x_2, \ldots, x_n$ by fitting a normal distribution to the sample logarithms $\log x_1, \log x_2, \ldots, \log x_n$ because log-likelihood functions are the same. For example, generate some samples according to the lognormal rule with parameters $\mu = 5$ and $\sigma^2 = 2$:

```
> X:=randvar(normal,mean=5,variance=2):;
  S:=sample(eval(exp(X),0),1000):;
```

Then fit the normal distribution to $\log S$:

```
> Y:=fitdistr(log(S),normal)
```

normald(5.04754808715, 1.42751619912)

The mean of Y is about 5.05 and the variance is about 2.04. Now the variable $Z = \mathrm{e}^Y$ has the sought lognormal distribution.

20.4.24 Markov chains

The `markov` command finds characteristic features of a Markov chain.

- `markov` takes M, a transition matrix for a Markov process.
- `markov(`M`)` returns a sequence consisting of
  - the list of the positive recurrent states.
  - the list of corresponding invariant probabilities.
  - the list of other strong connected components.
  - the list of probabilities of ending up in the sequence of recurrent states.

Example

```
> markov([[0,0,1/2,0,1/2],[0,0,1,0,0],[1/4,1/4,0,1/4,1/4],[0,0,1/2,0,1/2],[0,0,0,0,1]])
```

$$\left[4\right], \begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix}, \begin{bmatrix}3 & 1 & 2 & 0\end{bmatrix}, \begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}$$

20.4.25 Generating a random walks

The `randmarkov` command generates random walks or creates stochastic matrices.

- To generate a random walk, `randmarkov` takes two arguments:
  - M, a transition matrix for a Markov chain.
  - i_0, an initial state.
  - n, a positive integer.
- `randmarkov`(M, i_0, n) returns a a random walk (given as a vector) starting at i_0 and taking n random steps, where each step is a transition with probabilities given by M.
- To create a stochastic matrix, `randmarkov` takes two arguments:
  - v, a vector of length p.
  - i_0, the number of transient states.
- `randmatrix`(v, i_0) returns a stochastic matrix with p recurrent loops (given by v) and i_0 transient states.

Examples

```
> randmarkov([[0,1/2,0,1/2],[0,1,0,0],[1/4,1/4,1/4,1/4],[0,0,1/2,1/2]],2,10)
```

$$[2, 3, 2, 0, 3, 2, 2, 0, 3, 2, 0]$$

```
> randmarkov([1,2],2)
```

$$\begin{bmatrix}
1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.289031975209 & 0.710968024791 & 0.0 & 0.0 \\
0.0 & 0.46230383289 & 0.53769616711 & 0.0 & 0.0 \\
0.259262238137 & 0.149948861946 & 0.143448150524 & 0.242132758802 & 0.205207990592 \\
0.231568633749 & 0.145429586345 & 0.155664673778 & 0.282556511895 & 0.184780594232
\end{bmatrix}$$

20.5 Hypothesis testing

20.5.1 General

Given a random variable X, you may want to know whether some effective parameter p is the same as some expected value p_0. You will then want to test the hypothesis $p = p_0$, which will be the null hypothesis H_0. The alternative hypothesis will be H_1. The tests are:

Two-tailed test — This test will reject the hypothesis H_0 if the relevant statistic is outside of a determined interval. This is denoted by `'!='`.

Left-tailed test — This test will reject the hypothesis H_0 if the relevant statistic is less than a specific value. This is denoted by `'<'`.

Right-tailed test — This test will reject the hypothesis H_0 if the relevant statistic is greater than a specific value. This is denoted by `'>'`.

20.5.2 Testing the mean with the Z test

The `normalt` command uses the Z test to test the mean of data.

- `normalt` takes three mandatory arguments and one optional argument:
  - L, a list, which can be one of:
    - $L = [n_s, n_e]$ for the sample data information, where n_s is the the number of successes and n_e is the number of trials n_e.
    - $L = [m, t]$, where m is the mean and t is the sample size.
    - L, a data list from a control sample.
  - σ, the standard deviation of the population. If the data list from a control sample is provided, then this argument is unnecessary.
  - *test*, the type of test, one of `!=`, `<` or `>`.
  - Optionally, c, the confidence level (by default 0.05).
- `normalt(`$L, \sigma,$ *test* $\langle, c \rangle$`)` returns the result of a Z test. It will return 0 if the test fails, 1 if the test succeeds, and it will display a summary of the test.

Examples

```
> normalt([10,30],0.5,0.02,'!=',0.1)
```

```
*** TEST RESULT 0 ***
Summary Z-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1!=mu2.
   Test returns 0 if probability to observe data is less than 0.1
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (cannot reject null hypothesis)
Data mean mu1=10, population mean mu2=0.5
alpha level 0.1, multiplier*stddev/sqrt(sample size)=1.64485*0.02/5.47723
```

0

```
> normalt([0.48,50],0.5,0.1,'<')
```

```
*** TEST RESULT 1 ***
Summary Z-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1<mu2.
   Test returns 0 if probability to observe data is less than 0.05
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (cannot reject null hypothesis)
Data mean mu1=0.48, population mean mu2=0.5
alpha level 0.05, multiplier*stddev/sqrt(sample size)=1.64485*0.1/7.07107
```

1

20.5.3 Testing the mean with the T test

The `studentt` command examines whether data conforms to Student's distribution. For small sample sizes, the `studentt` test is preferable to `normalt` (see Section 20.5.2, p. 563).

- `studentt` command takes four mandatory arguments and one optional argument:
  - L, a list, which can be one of:
    - $L = [n_s, n_e]$ for the sample data information, where n_s is the the number of successes and n_e is the number of trials n_e.
    - $L = [m, t]$, where m is the mean and t is the sample size.
    - L, a data list from a control sample.
  - μ, the mean of the population to or a data list from a control sample.
  - σ, the standard deviation of the population. If the data list from a control sample is provided, then this argument is unnecessary.
  - *test*, the type of test, one of `!=`, `<` or `>`.
  - Optionally, c, the confidence level (by default 0.05).
- `studentt(`$L, \sigma,$ *test* $\langle, c \rangle$`)` returns the result of a T test. It will return 0 if the test fails, 1 if the test succeeds, and it will display a summary of the test.

Examples

```
> studentt([10,20],0.5,0.02,'!=',0.1)
```

```
*** TEST RESULT 0 ***
Summary T-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1!=mu2.
   Test returns 0 if probability to observe data is less than 0.1
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (cannot reject null hypothesis)
Data mean mu1=10, population mean mu2=0.5, degrees of freedom 20
alpha level 0.1, multiplier*stddev/sqrt(sample size)=1.32534*0.02/4.47214
```

```
0
```

```
> studentt([0.48,20],0.5,0.1,'<')
```

```
*** TEST RESULT 1 ***
Summary T-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1<mu2.
   Test returns 0 if probability to observe data is less than 0.05
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (cannot reject null hypothesis)
Data mean mu1=0.48, population mean mu2=0.5, degrees of freedom 20
alpha level 0.05, multiplier*stddev/sqrt(sample size)=1.72472*0.1/4.47214
```

```
1
```

20.5.4 Testing a distribution with the χ^2 distribution

The `chisquaret` command will use the χ^2 test to compare sample data to a specified distribution.

- `chisquaret` takes one mandatory argument and a sequence of optional arguments:
  - L, a list of sample data.
  - Optionally, *distr*. This can be one of
    - The name of a distribution (see Section 20.3.4, p. 526 for a list of distributions and their parameters).
    - Another list of sample data.

    By default this will be the `uniform` distribution.
  - *params*, the parameters of the distribution *distr* or the symbol `classes` and optionally c_{min} and c_{dim}, the minimum size and default size of a statistics class (by default, `class_min` and `class_size`, which themselves default to 0 and 1; see Section 2.5.8, p. 17).
- `chisquaret(`L ⟨,*distr* ⟩`)` returns the result of the χ^2 test between sample data and the named distribution or two sample data lists.

Examples

```
> chisquaret([57,54])
```

Guessing data is the list of number of elements in each class, adequation to uniform distribution
Sample adequation to a finite discrete probability distribution
Chi2 test result 0.0810810810811,
reject adequation if superior to chisquare_icdf(1,0.95)=3.84145882069 or chisquare_icdf(1,1-alpha) if alpha!=5%

0.0810810810811

```
> chisquaret([1,1,1,1,1,0,0,1,0,1,1],[.4,.6])
```

Sample adequation to a finite discrete probability distribution
Chi2 test result 0.742424242424,
reject adequation if superior to chisquare_icdf(1,0.95)=3.84145882069 or chisquare_icdf(1,1-alpha) if alpha!=5%

0.742424242424

```
> chisquaret(ranv(1000,binomial,10,.5),binomial)
```

Binomial: estimating n and p from data 10 0.5055
Sample adequation to binomial(10,0.5055,.), Chi2 test result 7.77825189838,
reject adequation if superior to chisquare_icdf(7,0.95)=14.0671404493 or chisquare_icdf(7,1-alpha) if alpha!=5%

7.77825189838

```
> chisquaret(ranv(1000,binomial,10,.5),binomial,11,.5)
```

Sample adequation to binomial(11,0.5,.), Chi2 test result 125.617374161,
reject adequation if superior to chisquare_icdf(10,0.95)=18.3070380533 or chisquare_icdf(10,1-alpha) if alpha!=5%

125.617374161

As an example using `class_min` and `class_size`:

```
> L:=ranv(1000,normald,0,.2):; chisquaret(L,normald,classes,-2,.25)
```

or (setting `class_min` to -2 and `class_size` to -0.25 in the graphical configuration):

```
> chisquaret(L,normald,classes)
```

```
Normal density, estimating mean and stddev from data -0.00345919752912 0.201708100832
Sample adequation to normald_cdf(-0.00345919752912,0.201708100832,.), Chi2 test result 2.11405080381,
reject adequation if superior to chisquare_icdf(4,0.95)=9.48772903678 or chisquare_icdf(4,1-alpha) if alpha!=5%
```

$$2.11405080381$$

In this last case, you are given the value of d^2 of the statistic $D^2 = \sum_{j=1}^{k}(n_j - e_j)/e_j$, where k is the number of sample classes for `classes(L,-2,0.25)` (or `classes(L)`), n_j is the size of the jth class, and $e_j = np_j$ where n is the size of `L` and p_j is the probability of the jth class interval assuming a normal distribution with the mean and population standard deviation of `L`.

20.5.5 Testing a distribution with the Kolmogorov-Smirnov distribution

The `kolmogorovt` command uses the Kolmogorov test to compare sample data to a specified continuous distribution.

- `kolmogorovt` takes two arguments and possibly additional parameters.
  - L, a list of sample data.
  - Optionally, *distr*. This can be one of:
    - The name of a distribution and the necessary parameters (see Section 20.3.4, p. 526 for a list of distributions and their parameters).
    - Another list of sample data.
- `kolmogorovt(`L`,`*distr*`)` returns a list of three values:
  - The D statistic, which is the maximum distance between the cumulative distribution functions of the samples or the sample and the given distribution.
  - The K value, where $K = D\sqrt{n}$ (for a single data set, where n is the size of the data set) or $K = D\sqrt{n_1 n_2/(n_1+n_2)}$ (when there are two data sets, with sizes n_1 and n_2). The K value will tend towards the Kolmogorov-Smirnov distribution as the data size approaches infinity.
  - `1-kolmogorovd(K)`, which will be close to 1 when the distributions look like they match.

Examples

```
> kolmogorovt(randvector(100,normald,0,1),normald(0,1))
```

$$[D = 0.112141597243, K = 1.12141597243, 1 - \text{kolmogorovd}(K) = 0.161616499536]$$

```
> kolmogorovt(randvector(100,normald,0,1),student(2))
```

$$[D = 0.112592987625, K = 1.12592987625, 1 - \text{kolmogorovd}(K) = 0.158375510292]$$

21 Signal Processing

21.1 Basic functions

21.1.1 Boxcar function

The `boxcar` command creates a function which has the value 1 in a given interval, and otherwise 0.

- `boxcar` takes three arguments:
  - a, b, two real numbers.
  - x, an identifier or expression.
- `boxcar(`a, b, x`)` returns $\theta(x-a)-\theta(x-b)$, where θ is the Heaviside function (see Section 7.3.9, p. 134). The resulting expression defines a function which is zero everywhere except within the segment $[a, b]$, where its value is equal to 1.

Examples

```
> boxcar(1,2,x)
```

$$\theta\left(x-1\right)-\theta\left(x-2\right)$$

```
> boxcar(1,2,3/2)
```

$$1$$

```
> boxcar(1,2,0)
```

$$0$$

21.1.2 Rectangle function

The rectangle function Π is 0 everywhere except on $[-1/2, 1/2]$, where it is 1; namely, $\Pi(x) = \theta(x+1/2)-\theta(x-1/2)$ where θ is the Heaviside function. The rectangle function is a special case of boxcar function (see Section 21.1.1, p. 567) for $a=-\frac{1}{2}$ and $b=\frac{1}{2}$.

The `rect` command computes the rectangle function.

- `rect` takes x, an identifier or an expression.
- `rect(`x`)` returns the value of the rectangle function at x.

Example

```
> rect(x/2)
```

$$\theta\left(\frac{x}{2}+\frac{1}{2}\right)-\theta\left(\frac{x}{2}-\frac{1}{2}\right)$$

To compute the convolution of the rectangle function with itself, you can use the Convolution Theorem (see Section 21.4.2, p. 579).

```
> R:=fourier(rect(x),x,s):; ifourier(R^2,s,x)
```

$$-2x\,\theta\,(x)+x\,\theta\,(x+1)+x\,\theta\,(x-1)+\theta\,(x+1)-\theta\,(x-1)$$

This result is the triangle function $\mathrm{tri}(x)$ (see section 21.1.3).

21.1.3 Triangle function

The triangle function is defined by

$$\Lambda(x)=\begin{cases}1-|x|, & |x|<1,\\ 0, & \text{otherwise.}\end{cases}$$

This is equal to the convolution of rectangle function with itself (see Section 21.1.2, p. 567).

The `tri` command computes the triangle function.

- `tri` takes x, an expression.
- `tri(`x`)` returns the value of triangle function at x.

Example

```
> tri(x-1)
```

$$(-x+1+1)\,(\theta\,(x-1)-\theta\,(x-1-1))+(1+x-1)\,(\theta\,(-x+1)-\theta\,(-x+1-1))$$

21.1.4 Cardinal sine function

The sinc function is defined by

$$\mathrm{sinc}(x)=\begin{cases}\frac{\sin(x)}{x}, & x\neq 0,\\ 1, & x=0.\end{cases}$$

The `sinc` command computes the sinc function.

- `sinc` takes x, an expression.
- `sinc(`x`)` returns the value of the sinc function at x.

Examples

```
> sinc(pi*x)
```

$$\frac{\sin\,(\pi x)}{\pi x}$$

```
> sinc(0)
```

$$1$$

21.2 Common operations on signals

21.2.1 Root mean square

The `rms` command finds the room mean square of a list of numbers $X = [x_1, x_2, \ldots, x_n]$, which is defined by

$$\mathrm{RMS}(X) = \sqrt{\frac{\sum_{k=1}^{n} |x_k|^2}{n}}.$$

(See Section 28.2.16, p. 833 for other uses of `rms`.)

- `rms` takes X, a list of real or complex numbers $[x_1, x_2, \ldots, x_n]$.
- `rms(`X`)` returns $\mathrm{RMS}(X)$ as defined above.

Examples

```
> rms([1,2,5,8,3,6,7,9,-1])
```

$$\sqrt{30}$$

```
> rms([1,1-i,2+3i,5-2i])
```

$$\frac{3}{2}\sqrt{5}$$

21.2.2 Finding and removing leading/trailing zeros

You can find leading and/or trailing zeros in a vector and remove them by using the `trim` command (see Section 5.2.5, p. 57, Section 28.1.9, p. 820 and Section 28.2.2, p. 823 for other usages of `trim`).

- `trim` takes one mandatory argument and up to three optional arguments:
  - v, a vector of real or complex numbers.
  - Optionally, *threshold*, a positive real number specifying the threshold for zeros (by default, *threshold* = `epsilon()`, the working precision).
  - Optionally, either `left` or `right`, the symbol specifiying one-sided trimming.
  - Optionally, `index`, the symbol.
- `trim(`v ⟨, *threshold*⟩ ⟨, `left`|`right` ⟩ ⟨, `index` ⟩`)` finds the index l of the first nonzero element in v and the index u of the first trailing zero in v. A number z is considered to be zero if $|z| \leq$ *threshold*. If `index` is given, then the return value is either l if `left` is given, u if `right` is given, or $(l, u-l)$ otherwise (the last sequence contains the start and length of the truncated vector). If `index` is omitted, then the return value is the portion of v between $v[l]$ (inclusive) and $v[u]$ (exclusive); if `left` is given, then $u =$ `length(`v`)`, and if `right` is given, then $l = 0$.

Example

```
> v:=[0,0,1e-16,0,-1e-13,1,2,3,0,0,0]:;
```

To remove leading zeros from v:

```
> trim(v,left)
```

$$[1, 2, 3, 0, 0, 0]$$

To remove leading and trailing zeros with threshold set to 10^{-14} (the default is the value of `epsilon()`, which is 10^{-12} in this case):

```
> trim(v,1e-14)
```

$$\left[-1.0\times 10^{-13}, 1, 2, 3\right]$$

To return the start and length of the nonzero part of v:

```
> d,n:=trim(v,index)
```

$$5, 3$$

Now, enter

```
> mid(v,d,n)
```

to obtain the trimmed variant of v.

21.2.3 Cross-correlation of two signals

The cross correlation of two complex vectors $v = [v_1, \ldots, v_n]$ and $w = [w_1, \ldots, w_m]$ is the complex vector $z = v \star w$ (note the difference between $\star$ and the convolution operation $*$, see Section 21.2.5, p. 571) of length $N = n + m - 1$ given by

$$z_k = \sum_{i=k}^{N-1} V_{i-k}^* W_i, \quad k = 0, 1, \ldots, N-1,$$

where

$$V = [v_0, v_1, \ldots, v_{n-1}, \underbrace{0, 0, \ldots, 0}_{m-1}] \quad \text{and} \quad W = [\underbrace{0, 0, \ldots, 0}_{n-1}, w_0, w_1, \ldots, w_{m-1}]$$

and V^* denotes the complex conjugate of V. Cross-correlation is typically used for measuring similarity between signals.

The `cross_correlation` command computes the cross correlation of two vectors.

- `cross_correlation` takes two arguments: v, w, two vectors (not necessarily the same length).
- `cross_correlation`(v, w) returns the cross correlation $v \star w$.

Examples

```
> cross_correlation([1,2],[3,4,5])
```

$$[6.0, 11.0, 14.0, 5.0]$$

```
> v:=[2,1,3,2]:; w:=[1,-1,1,2,2,1,3,2,1]:;
  round(cross_correlation(v,w))
```

$$[2, 1, 0, 8, 9, 12, 15, 18, 13, 11, 5, 2]$$

Observe that the cross-correlation of `v` and `w` is peaking at position 8 with the value 18, indicating that the two signals are best correlated when the last sample in `v` is aligned with the eighth sample in `w`. Indeed, there is an occurrence of `v` in `w` precisely at that point.

21.2.4 Auto-correlation of a signal

The auto correlation of a vector is its cross correlation with itself (see Section 21.2.3, p. 570). The `auto_correlation` command computes the auto correlation of a vector.

- `auto_correlation` takes v, a complex vector.
- `auto_correlation(`v`)` returns the cross-correlation of v with itself, $v \star v$.

Example

```
> auto_correlation([2,3,4,3,1,4,5,1,3,1])
```

$$[2.0, 9.0, 15.0, 28.0, 37.0, 44.0, 58.0, 58.0, 68.0, 91.0, 68.0, 58.0, 58.0, 44.0, 37.0, 28.0, 15.0, 9.0, 2.0]$$

21.2.5 Convolution of two signals or functions

The convolution of two real vectors $v = [v_1, \ldots, v_n]$ and $w = [w_1, \ldots, w_m]$ is the complex vector $z = v * w$ (note the difference between $*$ and the cross-correlation operation $\star$, see Section 21.2.3, p. 570) of length $n + m - 1$ given by

$$z_k = \sum_{i=0}^{k} v_i \, w_{k-i}, \quad k = 0, 1, \ldots, N-1,$$

such that $v_j = 0$ for $j \geq n$ and $w_j = 0$ for $j \geq m$.

The convolution of two real functions $f(x)$ and $g(x)$ is the integral

$$\int_{-\infty}^{+\infty} f(t)\, g(x-t)\, \mathrm{d}t.$$

The `convolution` command finds the convolution of two vectors or two functions.

- For the convolution of two vectors, `convolution` takes two arguments: v, w, two vectors (not necessarily the same length).
- `convolution(`v, w`)` returns the convolution $v * w$.
- For the convolution of two functions, `convolution` takes two mandatory arguments and one optional argument:
  - f, g, two expressions of a variable.
  - Optionally, x, the variable name.
- `convolution(`$f, g\,\langle, x\rangle$`)` returns the convolution of f and g restricted to $[0, +\infty\rangle$.

Remark. f and g are assumed to be *causal functions*, i.e. $f(x) = g(x) = 0$ for $x < 0$. Therefore both f and g are multiplied with $\theta(x)$ prior to integration.

Examples

To find the convolution of two lists $[1, 2, 3]$ and $[1, -1, 1, -1]$, enter:

```
> convolution([1,2,3],[1,-1,1,-1])
```

$$[1.0, 1.0, 2.0, -2.0, 1.0, -3.0]$$

Compute the convolution of $f(x) = 25\,\mathrm{e}^{2\,x}\,\theta(x)$ and $g(x) = x\,\mathrm{e}^{-3\,x}\,\theta(x)$, where θ is the Heaviside function:

```
> convolution(25*exp(2x),x*exp(-3x))
```

$$\left(-5x\mathrm{e}^{-3x} - \mathrm{e}^{-3x} + \mathrm{e}^{2x}\right)\theta\,(x)$$

Compute the convolution of $f(t) = \ln(1+t)\,\theta(t)$ and $g(t) = \frac{1}{\sqrt{t}}$.

```
> convolution(ln(1+t),1/sqrt(t),t)
```

$$-\frac{\theta\,(t)\left(2t\ln\left(\frac{-\sqrt{t}+\sqrt{t+1}}{\sqrt{t}+\sqrt{t+1}}\right)+4\sqrt{t}\sqrt{t+1}+2\ln\left(\frac{-\sqrt{t}+\sqrt{t+1}}{\sqrt{t}+\sqrt{t+1}}\right)\right)}{\sqrt{t+1}}$$

Application

A "dry" signal can be reverberated by convolving it with the *impulse response* of a particular acoustic space. The latter is typically obtained by recording a short, popping sound at the location, such as firing a starter gun.

The following demonstration requires two files: a music mono recording `msmn4.wav` downloaded from here and a two-channel impulse response `French 18th Century Salon.wav` downloaded from here. To load the files in XCAS, enter:

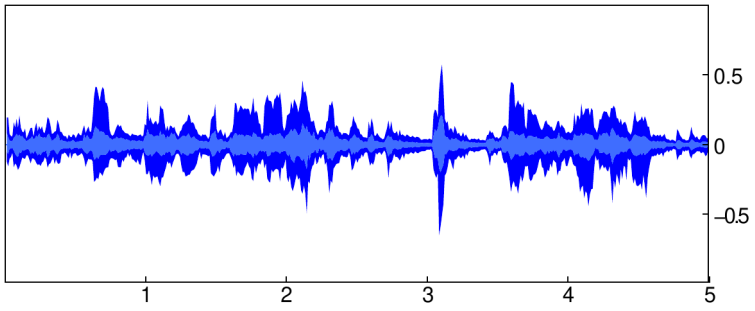
```
> clip:=readwav("/home/luka/Downloads/msmn4.wav")
```

a sound clip with 110250 samples at 22050 Hz (16 bit, mono)

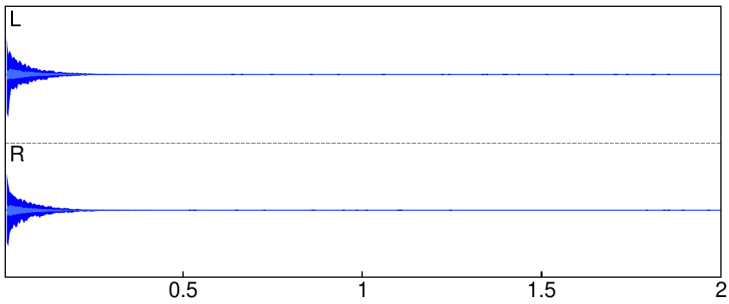
```
> ir:=readwav("/home/luka/Downloads/French 18th Century Salon.wav")
```

a sound clip with 88300 samples at 44100 Hz (16 bit, stereo)

```
> plotwav(clip)
```




```
> plotwav(ir)
```



Convolving data from `clip` with both channels in `ir` produces a reverberated variant of the recording, in stereo. Since the two clips have different sample rates, `clip` should be up-sampled to 44100 Hz before convolving (see Section 28.2.11, p. 829).

```
> data:=channel_data(resample(clip,samplerate(ir))):;
  L:=convolution(data,channel_data(ir,left)):;
  R:=convolution(data,channel_data(ir,right)):;
```

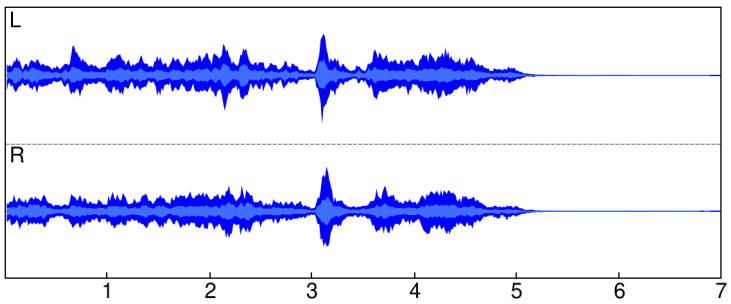
Evaluation time: 35.87

The convolved signals `L` and `R` now become the left and right channel of a new audio clip, respectively. The `normalize` option is used because convolution usually results in a huge increase of sample values.

```
> spatial:=createwav([L,R],normalize=-3):; playsnd(spatial)
```

The music being played back appears as it has been recorded in the same salon as the impulse response. Moreover, it is a true stereo sound at 44100 Hz as opposed to the original mono recording at 22050 Hz. To visualize it, enter:

```
> plotwav(spatial)
```



The length of the resulting audio equals the length of the original clip plus the length of the impulse response minus 1. Indeed:

```
> length(spatial)==length(data)+length(ir)-1
```

true

21.3 Filters

21.3.1 Low-pass filtering

The `lowpass` command applies a simple first-order lowpass RC filter to a signal or audio clip.

- `lowpass` takes two mandatory arguments and one optional argument:
  - A, an audio clip or a real vector representing the sampled signal.
  - c, a real number specifying the cutoff frequency.
  - Optionally, r, a samplerate (by default 44100).
- `lowpass(`$A, c\,\langle, r\rangle$`)` applies a simple first-order lowpass RC filter to input data and returns the result.

Example

To generate a synthetic signal, enter:

```
> f:=unapply(periodic(sign(x),x,-1/880,1/880),x):;
  s:=apply(f,soundsec(3)):;
```

Then:

```
> playsnd(lowpass(createwav(s),1000))
```

Output: the sound of the periodic signal after applying a lowpass RC filter with cutoff at 1000 Hz.

21.3.2 High-pass filtering

The `highpass` command applies a simple first-order lowpass RC filter to a signal or audio clip.

- `highpass` takes two mandatory arguments and one optional argument:
  - A, an audio clip or a real vector representing the sampled signal.
  - c, a real number specifying the cutoff frequency.
  - Optionally, r, a samplerate (by default 44100).
- `highpass`$(A, c\,\langle, r\rangle)$ applies a simple first-order highpass RC filter to input data and returns the result.

Example

To generate a synthetic signal, enter:

```
> f:=unapply(periodic(sign(x),x,-1/880,1/880),x);
  s:=apply(f,soundsec(3)):;
```

Then:

```
> playsnd(highpass(createwav(s),5000))
```

Output: the sound of the periodic signal after applying a highpass RC filter with cutoff at 5000 Hz.

21.3.3 Moving-average filter

The `moving_average` command applies a moving-average filter to a signal.

- `moving_average` takes two arguments:
  - A, an array of numeric values representing a sampled signal.
  - n, a positive integer.
- `moving_average`(A, n) returns an array B obtained by applying a moving-average filter of length n to A. The elements of B are defined by

$$B[i] = \frac{1}{n}\sum_{j=0}^{n-1} A[i+j]$$

  for $i = 0, 1, \ldots, L-n$, where L is the length of A.

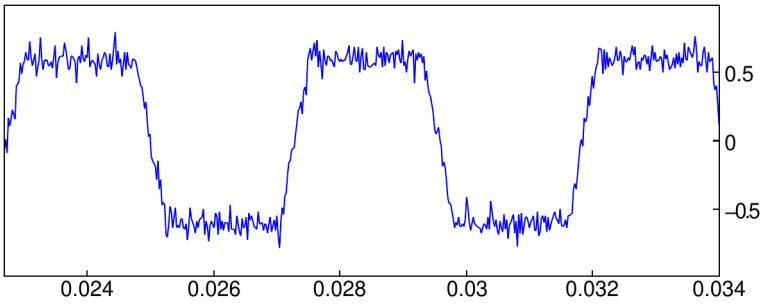
Moving-average filtering is fast and useful for smoothing time-encoded signals.

Example

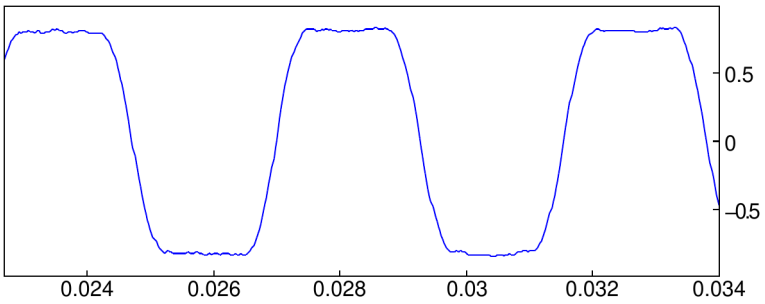
```
> snd:=soundsec(2):;
  noise:=randvector(length(snd),normald,0,0.05):;
  data:=0.5*threshold(3*sin(2*pi*220*snd),[-1.0,1.0])+noise:;
  noise_clip:=createwav(data);
```

a sound clip with 88200 samples at 44100 Hz (16 bit, mono)

```
> plotwav(noise_clip,range=[1000,1500])
```




```
> filtered_clip:=createwav(moving_average(data,25));
  plotwav(filtered_clip,range=[1000,1500])
```



You could also listen to the two waveforms by using `playsnd` with `noise_clip` and `filtered_clip` (see Section 28.2.14, p. 833).

21.3.4 Performing thresholding operations on an array

The `threshold` command changes data in an array by raising (or lowering) the values to meet a given threshold.

- `threshold` takes two mandatory arguments and two optional arguments:
  - v, an array (vector or matrix) of real or complex numbers.
  - a bound specification, which can be one of: This can be either:
    - $b = b_0$ for real numbers b and b_0.
    - b, a real number (equivalent to $b = b$).
    - $[l = l_0, u = u_0]$, for real numbers l, l_0, u, u_0.
    - $[l, u]$, a list of two real numbers (equivalent to $[l = l, u = u]$).
  - Optionally `'`$\prec$`'`, a quoted comparison operator, one of `'<'`, `'<='`, `'>'`, `'>='` (by default `'<'`).
  - Optionally, `abs=`*bool*, where *bool* is either `true` or `false` (by default `abs=false`). If `abs=true`, then the components of v must be real.
- `threshold(`$v, b = b_0\ \langle,' \prec' \rangle$`)` returns the array w of the same dimensions as v whose kth component is:

$$w_k = \begin{cases} b_0, & v_k \prec b, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is real and

$$w_k = \begin{cases} b_0 \frac{v_k}{|v_k|}, & |v_k| \prec b, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is complex; for $k = 0, 1, \ldots,$`size`$(v) - 1$.

- `threshold`$(v, b = b_0 \langle, \prec \rangle,$`abs=true`$)$ returns the vector w whose kth component is:

$$w_k = \begin{cases} b_0, & |v_k| \prec b \text{ and } v_k > 0, \\ -b_0, & |v_k| \prec b \text{ and } v_k < 0 \\ v_k, & \text{otherwise} \end{cases}$$

- `threshold`$(v, [l = l_0, u = u_0] \langle, \prec \rangle)$ returns the vector w whose kth component is:

$$w_k = \begin{cases} u_0, & u \prec v_k, \\ l_0, & v_k \prec l \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is real and

$$w_k = \begin{cases} u_0 \frac{v_k}{|v_k|}, & u \prec |v_k|, \\ l_0 \frac{v_k}{|v_k|}, & |v_k| \prec l, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is complex; for $k = 0, 1, \ldots,$`size`$(v) - 1$.

Examples

```
> threshold([2,3,1,2,5,4,3,7],3)
```

$$[3, 3, 3, 3, 5, 4, 3, 7]$$

```
> threshold([2,3,1,2,5,4,3,7],3=a,'>=')
```

$$[2, a, 1, 2, a, a, a, a]$$

```
> threshold([-2,-3,1,2,5,-4,3,-1],3=0,abs=true)
```

$$[0, -3, 0, 0, 5, -4, 3, 0]$$

```
> threshold([-2,-3,1,2,5,-4,3,-1],3=0,'<=',abs=true)
```

$$[0, 0, 0, 0, 5, -4, 0, 0]$$

```
> threshold([-120,-11,-3,0,7,27,111,234],[-100,100])
```

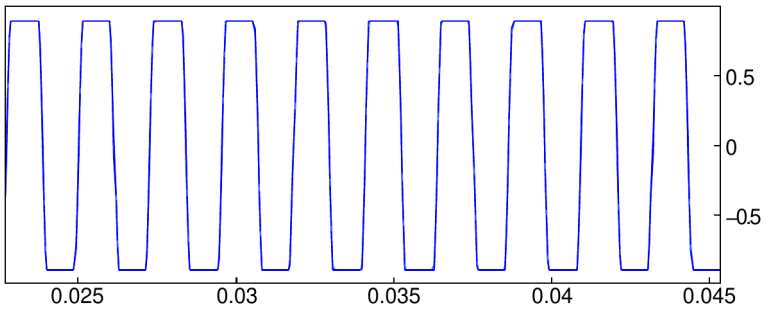
$$[-100, -11, -3, 0, 7, 27, 100, 100]$$

```
> threshold([-120,-11,-3,0,7,27,111,234],[-100=-inf,100=inf])
```

$$[-\infty, -11, -3, 0, 7, 27, +\infty, +\infty]$$

In the following example, a square-like signal is created from a sine wave by clipping sample values.

```
> data:=threshold(3*sin(2*pi*440*soundsec(2)),[-1.0,1.0]):;
  snd:=createwav(data):;
  plotwav(snd,range=[1000,2000])
```



21.4 Transforms

21.4.1 Fourier coefficients

Let f be a T-periodic continuous function on $\mathbb{R}$ except perhaps at a finite number of points. One can prove that if f is continuous at x, then;

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{+\infty} a_n \cos\left(\frac{2\pi n x}{T}\right) + b_n \sin\left(\frac{2\pi n x}{T}\right)$$
$$= \sum_{n=-\infty}^{+\infty} c_n \mathrm{e}^{\frac{2\mathrm{i}\pi n x}{T}}$$

where the coefficients a_n, b_n, $n \in N$, (or c_n, $n \in Z$) are the Fourier coefficients of f. The `fourier_an` and `fourier_bn` or `fourier_cn` commands compute these coefficients.

- `fourier_an` takes four mandatory and one optional argument:
  - *expr*, an expression depending on a variable.
  - x, the name of this variable.
  - T, the period.
  - n, a non-negative integer.
  - Optionally, a, a real number (by default $a = 0$).
- `fourier_an(`*expr*$, x, T, n\,\langle, a\rangle$`)` returns the Fourier coefficient a_n of a function f of variable x defined on $[a, a+T)$ by $f(x) = expr$ and such that f is periodic with period T:

$$a_n = \frac{2}{T}\int_a^{a+T} f(x)\cos\left(\frac{2\pi n x}{T}\right)\,\mathrm{d}x$$

- `fourier_bn` takes four mandatory and one optional argument:
  - *expr*, and expression depending on a variable.
  - x, the name of this variable.
  - T, the period.
  - n, an integer.
  - Optionally, a, a real number (by default $a = 0$).

- `fourier_bn(`*expr*, $x, T, n\,\langle, a\rangle$`)` returns the Fourier coefficient b_n of a function f of variable x defined on $[a, a+T)$ by $f(x) = expr$ and such that f is periodic with period T:

$$b_n = \frac{2}{T}\int_a^{a+T} f(x)\sin\left(\frac{2\pi n x}{T}\right)\,\mathrm{d}x$$

- `fourier_cn` takes four mandatory and one optional argument:
  - *expr*, and expression depending on a variable.
  - x, the name of this variable.
  - T, the period.
  - n, an integer.
  - Optionally, a, a real number (by default $a = 0$).
- `fourier_cn(`*expr*, $x, T, n\,\langle, a\rangle$`)` returns the Fourier coefficient c_n of a function f of variable x defined on $[a, a+T)$ by $f(x) = expr$ and such that f is periodic with period T:

$$c_n = \frac{1}{T}\int_a^{a+T} f(x)\,\mathrm{e}^{\frac{-2\mathrm{i}\pi n x}{T}}\,\mathrm{d}x$$

To simplify the computations, you should input `assume(n,integer)` (see Section 3.3.8, p. 39) before calling the above commands with an unspecified `n` to specify that it is an integer.

Examples

Let the function f, with period $T = 2$, be defined on $[-1, 1)$ by $f(x) = x^2$. To obtain the coefficient a_0, input:

```
> fourier_an(x^2,x,2,0,-1)
```

$$\frac{1}{3}$$

To obtain the coefficient a_n $(n \neq 0)$, input:

```
> assume(n,integer):;
  fourier_an(x^2,x,2,n,-1)
```

$$\frac{4\,(-1)^n}{n^2\pi^2}$$

Let the function f, with period $T = 2$, defined on $[-1, 1)$ by $f(x) = x^2$. To get the coefficient b_n $(n \neq 0)$, input:

```
> assume(n,integer):;
  fourier_bn(x^2,x,2,n,-1)
```

$$0$$

Let the function f, with period $T = 2$, defined on $[-1, 1)$ by $f(x) = x^3$. To get the coefficient b_1, input:

```
> fourier_bn(x^3,x,2,1,-1)
```

$$\frac{2\pi^2 - 12}{\pi^3}$$

Find the Fourier coefficients c_n of the periodic function f of period 2 and defined on $[-1, 1)$ by $f(x) = x^2$. To get c_0, input:

```
> fourier_cn(x^2,x,2,0,-1)
```

$$\frac{1}{3}$$

Input (to get c_n):

```
> assume(n,integer);
  fourier_cn(x^2,x,2,n,-1)
```

$$\frac{2\,(-1)^n}{n^2\pi^2}$$

Find the Fourier coefficients c_n of the periodic function f, of period 2, and defined on $[0, 2)$ by $f(x) = x^2$. To get c_0, input:

```
> fourier_cn(x^2,x,2,0)
```

$$\frac{4}{3}$$

To get c_n, input:

```
> assume(n,integer):;
  fourier_cn(x^2,x,2,n)
```

$$\frac{2\pi\mathrm{i}\,n + 2}{n^2\pi^2}$$

Find the Fourier coefficients c_n of the periodic function f of period 2π and defined on $[0, 2\pi)$ by $f(x) = x^2$.

```
> assume(n,integer):;
  fourier_cn(x^2,x,2*pi,n)
```

$$\frac{2\pi\mathrm{i}\,n + 2}{n^2}$$

You must also compute c_n for $n = 0$:

```
> fourier_cn(x^2,x,2*pi,0)
```

$$\frac{4}{3}\pi^2$$

Remarks.

- Input `purge(n)` (see Section 3.3.9, p. 41) to remove the hypothesis done on n.
- Input `about(n)` or `assume(n)`, to know the hypothesis done on the variable n.

21.4.2 Continuous Fourier Transform

The Fourier transform of a function f is defined by

$$F(\omega) = \int_{-\infty}^{+\infty} \mathrm{e}^{-\mathrm{i}\,\omega\,t}\, f(t)\,\mathrm{d}t = \mathcal{F}\left\{f(t)\right\}(\omega),$$

where $\omega \in \mathbb{R}$. The `fourier` command computes the Fourier transform by table lookup and application of transform properties.

- `fourier` takes one mandatory argument and several optional arguments:
  - *expr*, an expression which defines a function $f(t) = expr$.
  - Optionally, t, the variable for f (by default `t`).
  - Optionally, ω, the variable for the Fourier transform (by default `omega`).
  - Optionally, a sequence of arguments in form `opt`=*bool*, where *bool* is a boolean value and `opt` is one of the following:
    - `int` or `integrate` or `Int` or `Integrate`, for specifying whether symbolic integration should be attempted for unknown transform pairs (by default, this is disabled).
    - `diff` or `derive`, for specifying whether to attempt transforming the derivatives of unknown transform pairs (by default, this is disabled).
- `fourier(`*expr* $\langle, t, \omega \rangle$`)` returns the Fourier transform $F(\omega)$ of $f(t)$.

The inverse Fourier transform, as its name implies, takes a Fourier transform $F(\omega)$ and returns the original function $f(t)$. It is given by:

$$f(t) = \frac{1}{2\,\pi} \int_{-\infty}^{+\infty} \mathrm{e}^{\mathrm{i}\,\omega\,t}\, F(\omega)\,\mathrm{d}\omega = \mathcal{F}^{-1}\left\{F(\omega)\right\}(t) = \frac{1}{2\pi}\mathcal{F}\left\{F(\omega)\right\}(-t).$$

The `ifourier` command computes the inverse Fourier transform by using the above identity.

- `ifourier` takes one mandatory argument and several optional arguments:
  - *expr*, an expression which defines a function $F(\omega) = expr$.
  - Optionally, ω, the variable for F (by default `omega`).
  - Optionally, t, the variable for the original function f (by default `t`).
  - Optionally, a sequence of arguments in form `opt`=*bool*, where *bool* is a boolean value and `opt` is one of the following:
    - `int` or `integrate` or `Int` or `Integrate`, for specifying whether symbolic integration should be attempted for unknown transform pairs (by default, this is disabled).
    - `diff` or `derive`, for specifying whether to attempt finding unknown transform pairs by transforming the derivative of the original (by default, this is disabled).
- `ifourier(`*expr* $\langle, \omega, t \rangle$`)` returns the inverse Fourier transform $f(t)$ of $F(\omega)$.

The `Fourier` command is the inert form of `fourier`. It is used by `fourier` and `ifourier` when returning unknown transforms.

Transforming rational functions. An arbitrary rational function can be transformed as long as its full partial fraction decomposition can be found.

Examples

Find the Fourier transform of $f(t) = \frac{t}{t^3-19\,t+30}$.

```
> F:=fourier(t/(t^3-19t+30))
```

$$\frac{\pi\left(5\mathrm{i}\mathrm{e}^{5\mathrm{i}\omega} - 21\mathrm{i}\mathrm{e}^{-3\mathrm{i}\omega} + 16\mathrm{i}\mathrm{e}^{-2\mathrm{i}\omega}\right)\mathrm{sign}\,(\omega)}{56}$$

Indeed:

```
> ifourier(F)
```

$$\frac{t}{t^3 - 19t + 30}$$

Find the transform of $f(t) = \frac{t^2+1}{t^2-1}$.

```
> F:=fourier((t^2+1)/(t^2-1))
```

$$2\pi\left(\delta\left(\omega\right) - \operatorname{sign}\left(\omega\right)\sin\omega\right)$$

Indeed:

```
> ifourier(F)
```

$$\frac{t^2+1}{t^2-1}$$

Find the transform of $f(t) = \frac{2}{1+4t^4}$.

```
> F:=fourier(2/(1+4t^4))
```

$$\pi\left(\sin\left(\frac{|\omega|}{2}\right) + \cos\left(\frac{|\omega|}{2}\right)\right)\mathrm{e}^{-\frac{|\omega|}{2}}$$

Indeed:

```
> ifourier(F)
```

$$\frac{2}{4t^4+1}$$

As a special case, you can transform all poylnomials. For example:

```
> fourier(3x^2+2x+1,x,s)
```

$$2\pi\left(\delta\left(s\right) + 2\mathrm{i}\,\delta\left(s,1\right) - 3\,\delta\left(s,2\right)\right)$$

Transforming general functions. A range of other (generalized) functions and distributions can be transformed, as demonstrated in the following examples.

Examples

```
> fourier(Dirac(x-1)+Dirac(x+1),x)
```

$$2\cos\omega$$

```
> fourier(exp(-2*abs(x-1)),x,s)
```

$$\frac{4\mathrm{e}^{-\mathrm{i}s}}{s^2+4}$$

```
> fourier(BesselJ(3,x),x,s)
```

$$-\frac{s\left(4s^2-3\right)\left(-\mathrm{isign}\left(s+1\right)+\mathrm{isign}\left(s-1\right)\right)}{\sqrt{-s^2+1}}$$

```
> F:=fourier(sin(x)*sign(x),x,s)
```

$$-\frac{2}{s^2-1}$$

```
> ifourier(F,s,x)
```

$$\operatorname{sign}\left(x\right)\sin x$$

```
> fourier(log(abs(x)),x,s)
```

$$-\frac{\pi}{|s|}-2\pi\gamma\,\delta\left(s\right)$$

```
> F:=fourier(abs(2t-3)^(-3/4))
```

$$\frac{\Gamma\left(\frac{1}{4}\right)\sqrt{\sqrt{2}+2}\left(|\omega|^{\frac{1}{4}}\right)^{3}\mathrm{e}^{-\frac{3\mathrm{i}\omega}{2}}}{\left(2^{\frac{1}{4}}\right)^{3}|\omega|}$$

```
> ifourier(F)
```

$$\frac{1}{\left(|-2t+3|^{\frac{1}{4}}\right)^{3}}$$

```
> fourier(rect(x),x,s)
```

$$\frac{2\sin\left(\frac{s}{2}\right)}{s}$$

```
> fourier(exp(-abs(x))*sinc(x),x,s)
```

$$\arctan\left(s+1\right)-\arctan\left(s-1\right)$$

```
> fourier(1/sqrt(abs(x)),x,s)
```

$$\frac{\sqrt{2}\sqrt{\pi}}{\sqrt{|s|}}$$

```
> F:=fourier(1/cosh(2x),x,s)
```

$$\frac{\pi}{\mathrm{e}^{-\frac{1}{4}\pi s}+\mathrm{e}^{\frac{1}{4}\pi s}}$$

```
> ifourier(F,s,x)
```

$$\frac{2}{\mathrm{e}^{-2x}+\mathrm{e}^{2x}}$$

```
> F:=fourier(Gamma(1+i*x/3),x,s)
```

$$6\pi\mathrm{e}^{-\left(3s+\mathrm{e}^{-3s}\right)}$$

```
> ifourier(F,s,x)
```

$$\Gamma\left(\frac{1}{3}\mathrm{i}x+1\right)$$

```
> fourier(atan(x/4)/x,x,s)
```

$$\pi\mathrm{ugamma}\left(0,4\,|s|\right)$$

```
> assume(a>0):;
  fourier(exp(-a*x^2+b),x,s)
```

$$\frac{\sqrt{a}\sqrt{\pi}\mathrm{e}^{-\frac{s^2}{4a}+b}}{a}$$

Piecewise functions can be transformed if defined as

$$\texttt{piecewise}(t < a_1, f_1, x < a_2, f_2, \ldots, t < a_n, f_n, f_0)$$

where $f_0, \ldots, f_n$ are expressions and $a_1, a_2, \ldots, a_n$ are real numbers such that $a_1 < a_2 < \cdots < a_n$. Inequalities may be strict or non-strict. For example:

```
> f:=piecewise(t<-1,exp(t+1),t<1,1,exp(2-2t)):;
  F:=fourier(f)
```

$$\frac{-\mathrm{i}\omega \sin\omega + 3\omega\cos\omega + 4\sin\omega}{\omega\,(\omega - 2\mathrm{i})\,(\omega + \mathrm{i})}$$

```
> ifourier(F)
```

$$\theta\,(-t+1) - \theta\,(-t-1) + \theta\,(t-1)\,\mathrm{e}^{-2t+2} + \theta\,(-t-1)\,\mathrm{e}^{t+1}$$

You can verify that the above expression coincides with $f(t)$ by plotting them together with the `plot` command. Alternatively, you can apply `piecewise` to the result (see Section 5.1.3, p. 50):

```
> piecewise(ans())
```

$$\begin{cases} \mathrm{e}^{t+1}, & t < -1 \\ 1, & t < 1 \\ \mathrm{e}^{-(2t-2)}, & \text{otherwise} \end{cases}$$

Convolution Theorem and applications. The Fourier transform behaves nicely when applied to convolutions. Recall that the *convolution* $f * g$ (see Section 21.2.5, p. 571) of two functions f and g is defined by

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(u)\, g(t-u)\, \mathrm{d}u.$$

The Convolution Theorem states that $\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$.

As an example, we use this result for computing the convolution of $f(t) = \mathrm{e}^{-|t|}$ with itself. Note that Xcas is unable to compute $f * f$ from the definition:

```
> f(t):=exp(-abs(t)):;
  int(f(u)*f(t-u),u=-inf..inf)
```

undef

Using Fourier transform:

```
> F:=fourier(f(t))
```

$$\frac{2}{\omega^2 + 1}$$

```
> collect(ifourier(F^2))
```

$$(|t| + 1)\,\mathrm{e}^{-|t|}$$

The above result is the desired convolution $(f * f)(t) = \int_{-\infty}^{+\infty} f(u)\, f(t-u)\, \mathrm{d}u$.

21.4.3 Defining symbolic transform pairs

Some algebraic transformations behave predictably under the Fourier or Laplace transform. For example, if $g(x) = f(x-a)$, then $\mathcal{F}\{g\}(s) = \mathrm{e}^{-2\pi i a s}\mathcal{F}\{f\}(s)$. The `addtable` command lets you assign a function name to the Fourier (see Section 21.4.2, p. 579) or Laplace (see Section 13.4.2, p. 296) transform of another function name, without specifying the either function. This allows you to alter the original function and see the effect on the transform.

- `addtable` takes five arguments:
  - *transform*, which can be `fourier` or `laplace` and indicates the type of transform.
  - $f(x)$, where f is a symbol representing an unspecified function of the variable x.
  - $F(s)$, where F is a symbol representing the transform of f and s is the new variable.
  - x, the variable used by f.
  - s, the variable used by F.
- `addtable(`*transform*$, f(x), F(s), x, s$`)` returns 1 if F is assigned as the transform of f, and 0 otherwise. In the case that F is assigned as the transform of f, then the transform (`fourier` or `laplace`) of manipulations of f will be returned in terms of F and conversely.

Examples

```
> addtable(fourier,y(x),Y(s),x,s)
```

$$1$$

```
> F:=fourier(y(a*x+b),x,s)
```

$$\frac{\mathrm{e}^{\frac{ibs}{a}} Y\left(\frac{s}{a}\right)}{|a|}$$

```
> ifourier(F,s,x)
```

$$y\left(ax+b\right)$$

```
> fourier(Y(x),x,s)
```

$$2\pi y\left(-s\right)$$

```
> fourier(y(x)+Int(y(t),t=x..inf),x,s)
```

$$\frac{Y\left(s\right)\left(\pi s\,\delta\left(s\right)+s+\mathrm{i}\right)}{s}$$

```
> addtable(fourier,g(x,t),G(s,t),x,s)
```

$$1$$

```
> fourier(g(x/2,3*t),x,s)
```

$$2G\left(2s,3t\right)$$

ODE solving with Fourier transforms. Fourier transforms can be used for solving (partial) differential equations. For example, to obtain a particular solution to the equation

$$y(x) + y''(x) = x^2\,\theta(x),$$

where θ is the Heaviside function, you can first transform both sides of the above equation. Assuming that the symbolic transform pair $\mathcal{F}\{y(x)\} = Y(s)$ has been defined by using `addtable` as above, then:

```
> L:=fourier(y(x)+diff(y(x),x,2),x,s);
  R:=fourier(x^2*Heaviside(x),x,s)
```

$$-Y(s)\,(s+1)\,(s-1)\,,\ -\pi\,\delta\,(s,2) + \frac{2\mathrm{i}}{s^3}$$

Then you can solve the equation $L = R$ for $Y(s)$. Generally, you should apply `csolve` instead of `solve`.

```
> sol:=csolve(L=R,Y(s))[0]
```

$$\frac{\pi s^3\,\delta\,(s,2) - 2\mathrm{i}}{s^3\,(s+1)\,(s-1)}$$

Finally, you can apply `ifourier` to obtain $y(x)$.

```
> expand(ifourier(sol,s,x))
```

$$\frac{x^2 + x^2\mathrm{sign}\,(x) + 2\mathrm{sign}\,(x)\cos x - 2\mathrm{sign}\,(x) - 2}{2}$$

The above solution can be combined with solutions of the corresponding homogeneous equation to obtain the general solution.

As another example, solve the Airy equation $y'' - xy = 0$. Its Fourier transform is a first-order linear differential equation in Y:

```
> eq:=fourier(diff(y(x),x,2)-x*y(x),x,s)=0
```

$$-\mathrm{i}\frac{\mathrm{d}}{\mathrm{d}s}Y(s) - s^2 Y(s) = 0$$

This equation can be solved by using `dsolve`:

```
> dsolve(eq,s,Y)
```

$$c_0 \mathrm{e}^{\frac{1}{3}\mathrm{i}s^3}$$

The above result is the Fourier transform of y, i.e. $y(x) = c_0\mathcal{F}^{-1}\left\{\mathrm{e}^{\frac{1}{3}\mathrm{i}s^3}\right\}(x)$. For $c_0 = 1$ this is the Airy function of the first kind. Indeed:

```
> fourier(Airy_Ai(x),x,s)
```

$$\mathrm{e}^{\frac{1}{3}\mathrm{i}s^3}$$

21.4.4 Discrete Fourier Transform and Fast Fourier Transform

For any integer N, the Discrete Fourier Transform (DFT) is a transformation F_N defined on the set of periodic sequences of period N; it depends on a choice of a primitive nth root of unity ω_N. For sequences with complex coefficients, we take $\omega_N = \mathrm{e}^{\frac{2\mathrm{i}\pi}{N}}$. If x is a periodic sequence of period N, defined by the vector $x = [x_0, x_1, \ldots x_{N-1}]$ then $F_N(x) = y$ is a periodic sequence of period N, defined by:

$$(F_N(x))_k = y_k = \sum_{j=0}^{N-1} x_j \omega_N^{-kj}, \quad k = 0, \ldots, N-1.$$

F_N is bijective with inverse

$$F_N^{-1} = \frac{1}{N}\overline{F_{N,\omega_N}} \quad \text{i.e.} \quad \left(F_{N,\omega_N}^{-1}(x)\right)_k = \frac{1}{N}\sum_{j=0}^{N-1} x_j \omega_N^{kj}.$$

The Fast Fourier Transform (FFT) is an efficient way to compute the discrete Fourier transform; faster than computing each term individually. Xcas implements the FFT algorithm to compute the DFT when the period of the sequence is a power of 2.

The `fft` command computes the discrete Fourier transform.

- `fft` takes x, a list or sequence regarded as one period of a periodic sequence.
- `fft(`x`)` returns $F_N(x)$, the discrete Fourier transform of x.

  If x has length which is a power of 2, then $F_N(x)$ is computed with the Fast Fourier Transform.

The `ifft` command computes the inverse discrete Fourier transform.

- `ifft` takes x, a list or sequence regarded as one period of a periodic sequence.
- `ifft(`x`)` returns $F_N^{-1}(x)$, the inverse discrete Fourier transform of x.

  If x has length which is a power of 2, then $F_N^{-1}(x)$ is computed with the Fast Fourier Transform.

Examples

```
> fft(0,1,1,0)
```

$$[2.0, -1.0 - \mathrm{i}, 0.0, -1.0 + \mathrm{i}]$$

```
> ifft([2,-1-i,0,-1+i])
```

$$[0.0, 1.0, 1.0, 0.0]$$

Applications

Value of a polynomial. Define a polynomial $P(x) = \sum_{j=0}^{N-1} c_j x^j$ by the vector of its coefficients $c := [c_0, c_1, \ldots, c_{N-1}]$, where zeroes may be added so that N is a power of 2 (so the Fast Fourier Transform can be used).

Let us compute the values of $P(x)$ at $x = a_k = \omega_N^{-k} = \mathrm{e}^{\frac{-2\mathrm{i}k\pi}{N}}$, $k = 0, \ldots, N-1$. This is just the DFT of c since

$$P(a_k) = \sum_{j=0}^{N-1} c_j (\omega_N^{-k})^j = F_N(c)_k.$$

For example, let $P(x + x^2)$ and $x = 1, i, -1, -i$. Here the coefficients of P are $[0, 1, 1, 0]$, $N = 4$ and $\omega = \mathrm{e}^{2\mathrm{i}\pi/4} = i$.

```
> fft([0,1,1,0])
```

$$[2.0, -1.0 - \mathrm{i}, 0.0, -1.0 + \mathrm{i}]$$

Hence $P(1) = 2$, $P(-\mathrm{i}) = P(\omega^{-1}) = -1 - \mathrm{i}$, $P(-1) = P(\omega^{-2}) = 0$, $P(\mathrm{i}) = P(\omega^{-3}) = -1 + \mathrm{i}$.

Let us now compute the values of $P(x)$ at $x = b_k = \omega_N^k = \mathrm{e}^{\frac{2\mathrm{i}k\pi}{N}}$ for $k = 0, \ldots, N-1$. This is the inverse DFT of c since

$$P(a_k) = \sum_{j=0}^{N-1} c_j (\omega_N^{k})^j = N F_N^{-1}(c)_k.$$

Use this method to find the values of $P(x+x^2)$ at $x = 1, \mathrm{i}, -1, -\mathrm{i}$. Again, the coefficients of P are $[0,1,1,0]$, $N = 4$ and $\omega = \mathrm{e}^{2\mathrm{i}\pi/4} = \mathrm{i}$.

```
> 4*ifft([0,1,1,0])
```

$$[2.0, -1.0 + \mathrm{i}, 0.0, -1.0 - \mathrm{i}]$$

Hence $P(1) = 2$, $P(\mathrm{i}) = P(\omega^1) = -1 + \mathrm{i}$, $P(-1) = P(\omega^2) = 0$, $P(-\mathrm{i}) = P(\omega^3) = -1 - \mathrm{i}$. You find of course the same values as above.

Trigonometric interpolation. Let f be periodic function of period 2π and let $f_k = f(2k\pi/N)$ for $k = 0, \ldots, N-1$. Find a trigonometric polynomial p that interpolates f at $x_k = 2k\pi/N$, that is find p_j for $j = 0, \ldots, N-1$ such that

$$p(x) = \sum_{j=0}^{N-1} p_j x^j, \quad p(x_k) = f_k.$$

Replacing x_k by its value in $p(x)$ we get:

$$\sum_{j=0}^{N-1} p_j \mathrm{e}^{\mathrm{i}\frac{j2k\pi}{N}} = f_k.$$

In other words, (f_k) is the inverse DFT of (p_k), hence

$$(p_k) = \frac{1}{N} F_N\left((f_k)\right).$$

If the function f is real then $p_{-k} = \overline{p}_k$, hence

$$p(x) = \begin{cases} p_0 + 2\,\mathrm{Re}\left(\sum_{k=0}^{\frac{N}{2}-1} p_k \mathrm{e}^{\mathrm{i}kx}\right) + \mathrm{Re}\left(p_{\frac{N}{2}} \mathrm{e}^{\mathrm{i}\frac{Nx}{2}}\right), & N \text{ even}, \\ p(x) = p_0 + 2\,\mathrm{Re}\left(\sum_{k=0}^{\frac{N-1}{2}} p_k \mathrm{e}^{\mathrm{i}kx}\right), & N \text{ odd}. \end{cases}$$

For an example, see Section 21.4.4, p. 587.

Fourier series. Let f be a periodic function of period 2π and let $y_k = f(x_k)$ where $x_k = \frac{2k\pi}{N}$ for $k = 0, \ldots, N-1$. Suppose that the Fourier series of f converges to f (this will be the case if for example f is continuous). If N is large, a good approximation of f will be given by:

$$\sum_{-\frac{N}{2} \le n < \frac{N}{2}} c_n \mathrm{e}^{\mathrm{i}nx}.$$

Hence we want a numeric approximation of

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(t) \mathrm{e}^{-\mathrm{i}nt}\, \mathrm{d}t.$$

The numeric value of the integral $\int_0^{2\pi} f(t)\mathrm{e}^{-\mathrm{i}nt}\,\mathrm{d}t$ can be computed by the trapezoidal rule (note that the Romberg algorithm would not work here because the Euler-Maclaurin development has its coefficients equal to zero, since the integrated function is periodic, hence all its derivatives have the same value at 0 and at 2π). If $\tilde{c}_n$ is the numeric value of c_n obtained by the trapezoidal rule, then

$$\tilde{c}_n = \frac{1}{2\pi} \frac{2\pi}{N} \sum_{k=0}^{N-1} y_k \mathrm{e}^{-2\mathrm{i}\frac{nk\pi}{N}}, \quad -\frac{N}{2} \le n < \frac{N}{2}.$$

Indeed, since $x_k = 2k\pi/N$ and $f(x_k) = y_k$:

$$f(x_k)\mathrm{e}^{-\mathrm{i}nx_k} = y_k\mathrm{e}^{-2\mathrm{i}\frac{nk\pi}{N}},$$
$$f(0)\,\mathrm{e}^0 = f(2\pi)\mathrm{e}^{-2\mathrm{i}\frac{nN\pi}{N}} = y_0 = y_N.$$

Hence $[\tilde{c}_0, \ldots, \tilde{c}_{\frac{N}{2}-1}, \tilde{c}_{\frac{N}{2}+1}, \ldots, c_{N-1}] = \frac{1}{N}F_N([y_0, y_1, \ldots, y_{(N-1)}])$, since

- if $n \geq 0$, then $\tilde{c}_n = y_n$.
- if $n < 0$, then $\tilde{c}_n = y_{n+N}$.
- $\omega_N = \mathrm{e}^{\frac{2\mathrm{i}\pi}{N}}$, hence $\omega_N^n = \omega_N^{n+N}$.

Several properties are listed below.

- The coefficients of the trigonometric polynomial that interpolates f at $x = 2k\pi/N$ are

$$p_n = \tilde{c}_n, \quad -\frac{N}{2} \leq n < \frac{N}{2}.$$

- If f is a trigonometric polynomial of degree $m \leq \frac{N}{2}$, then

$$f(t) = \sum_{k=-m}^{m-1} c_k\mathrm{e}^{2\mathrm{i}k\pi t}.$$

  The trigonometric polynomial that interpolates f is f itself, the numeric approximation of the coefficients are in fact exact ($\tilde{c}_n = c_n$).

- More generally, you can compute $\tilde{c}_n - c_n$.

  Suppose that f is equal to its Fourier series, i.e. that:

$$f(t) = \sum_{m=-\infty}^{+\infty} c_m\mathrm{e}^{2\mathrm{i}\pi mt}, \quad \sum_{m=-\infty}^{+\infty} |c_m| < \infty.$$

  Then:

$$f(x_k) = f\left(\frac{2k\pi}{N}\right) = y_k = \sum_{m=-\infty}^{+\infty} c_m\omega_N^{km}, \quad \tilde{c}_n = \frac{1}{N}\sum_{k=0}^{N-1} y_k\omega_N^{-kn}.$$

  Replace y_k by its value in $\tilde{c}_n$:

$$\tilde{c}_n = \frac{1}{N}\sum_{k=0}^{N-1}\sum_{m=-\infty}^{+\infty} c_m\omega_N^{km}\omega_N^{-kn}.$$

  If $m \neq n \pmod N$, ω_N^{m-n} is an nth root of unity different from 1, hence:

$$\omega_N^{(m-n)N} = 1, \quad \sum_{k=0}^{N-1}\omega_N^{(m-n)k} = 0.$$

  Therefore, if $m - n$ is a multiple of N ($m = n + l \cdot N$) then $\sum_{k=0}^{N-1}\omega_N^{k(m-n)} = N$, otherwise $\sum_{k=0}^{N-1}\omega_N^{k(m-n)} = 0$. By reversing the two sums, you get

$$\tilde{c}_n = \frac{1}{N}\sum_{m=-\infty}^{+\infty} c_m\sum_{k=0}^{N-1}\omega_N^{k(m-n)}$$

$$= \sum_{l=-\infty}^{+\infty} c_{(n+l\cdot N)}$$
$$= \cdots + c_{n-2N} + c_{n-N} + c_n + c_{n+N} + c_{n+2N} + \cdots$$

Conclusion: if $|n| < N/2$, then $\tilde{c}_n - c_n$ is a sum of c_j with large indices (at least $N/2$ in absolute value), hence is small (depending on the rate of convergence of the Fourier series).

For example, input:

```
> f(t):=cos(t)+cos(2*t):;
  x:=f(2*k*pi/8)$(k=0..7)
```

$$2, \frac{\sqrt{2}}{2}, -1, -\frac{\sqrt{2}}{2}, 0, -\frac{\sqrt{2}}{2}, -1, \frac{\sqrt{2}}{2}$$

```
> fft(x)
```

$$[0.0, 4.0, 4.0, 0.0, 0.0, 0.0, 4.0, 4.0]$$

Dividing by $N = 8$, you get

$$c_0 = 0, c_1 = 0.5, c_2 = 0.5, c_3 = 0.0,$$
$$c_{-4} = 0.0, c_{-3} = 0.0, c_{-2} = 0.5, c_{-1} = 0.5.$$

Hence $b_k = 0$ and $a_k = c_{-k} + c_k$ equals 1 for $k = 1, 2$ and 0 otherwise.

Convolution Product. If $P(x) = \sum_{j=0}^{n-1} a_j x^j$ and $Q(x) = \sum_{j=0}^{m-1} b_j x^j$ are given by the vectors of their coefficients $a = [a_0, a_1, \ldots, a_{n-1}]$ and $b = [b_0, b_1, \ldots, b_{m-1}]$, you can compute the product of these two polynomials using the DFT. The product of polynomials is the convolution product of the periodic sequence of their coefficients if the period is greater or equal to $(n + m)$. Therefore we complete a (resp. b) with $m + p$ (resp. $n + p$) zeros, where p is chosen such that $N = n + m + p$ is a power of 2. If $a = [a_0, a_1, \ldots, a_{n-1}, 0, \ldots, 0]$ and $b = [b_0, b_1, \ldots, b_{m-1}, 0, \ldots, 0]$, then:

$$P(x)Q(x) = \sum_{j=0}^{n+m-1} (a * b)_j x^j.$$

If you know $F_N(a)$ and $F_N(b)$, then $a * b = F_N^{-1}(F_N(a) \odot F_N(b))$, where $\odot$ denotes the Hadamard (elementwise) product.

Noise removal with spectral subtraction. We use XCAS to implement a simple algorithm for static noise removal based on the spectral subtraction method[1].

The efficiency of the spectral subtraction method largely depends on a good noise spectrum estimate. Below is the code for a function `noiseprof` that takes `data` and `wlen` as its arguments. These are, respectively, a signal chunk containing only noise and the window length for signal segmentation (the best values are powers of two, such as 256, 512 or 1024). The function returns an estimate of the noise power spectrum obtained by averaging the power spectra of a (not too large) number of distinct chunks of `data` of length `wlen`. The Hamming window is applied prior to FFT.

```
noiseprof(data,wlen):={
  local N,h,dx,x,v,cnt;
  N:=length(data);
```

[1]For a theoretical overview see "Noise Reduction Based on Modified Spectral Subtraction Method" by Ekaterina Verteletskaya and Boris Simak (2011), *International Journal of Computer Science*, 38:1 (PDF).

```
  h:=wlen/2;
  dx:=min(h,max(1,(N-wlen)/100));
  v:=[0$wlen];
  for (x:=h,cnt:=0;x<N-h;x+=dx,cnt++)
    v+=abs(fft(hamming_window(data,floor(x)-h,wlen))).^2;
  return 1.0/cnt*v;
};:;
```

The main function is `noisered`, which takes three arguments: the input signal `data`, the noise power spectrum `np` and the "spectral floor" parameter `beta` (β, the minimum power level). The function performs subtraction of the noise spectrum in chunks of length `wlen` (the length of list `np`) using the overlap-and-add approach with the Hamming window function.

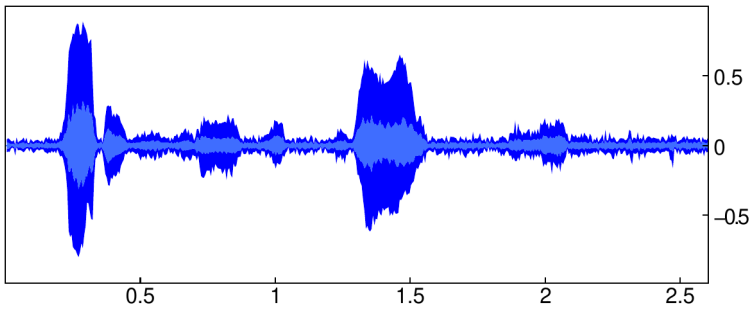
```
noisered(data,np,beta):={
  local wlen,h,N,L,padded,out,j,k,s,ds,r,alpha;
  wlen:=length(np);
  N:=length(data);
  h:=wlen/2;
  L:=0;
  repeat L+=wlen; until L>=N;
  padded:=concat(data,[0$(L-N)]);
  out:=[0$L];
  for (k:=0;k<L-wlen;k+=h) {
    s:=fft(hamming_window(padded,k,wlen));
    alpha:=max(1,4-3*sum(abs(s).^2)/(20*sum(np)));
    r:=ifft(zip(max,abs(s).^2-alpha*np,beta*np).^(1/2).*exp(i*arg(s)));
    for (j:=0;j<wlen;j++) out[k+j]+=re(r[j]);
  };
  return mid(out,0,N);
};:;
```

You can test the algorithm on a small speech sample with an audible amount of static noise (you can download the audio from here).

```
> clip:=normalize(readwav("/home/luka/Downloads/sf1_n1L.wav"),-1)
```

a sound clip with 41677 samples at 16000 Hz (16 bit, mono)

```
> plotwav(clip)
```

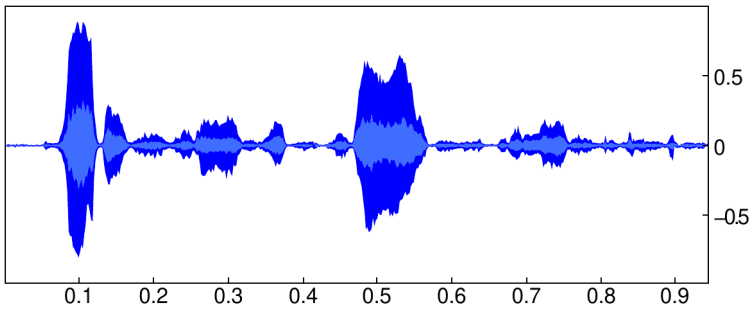


Speech starts after approximately 0.2 seconds of pure noise. You can use that part of the clip for obtaining an estimate of the noise power spectrum with `wlen` set to 256.

```
> noise:=channel_data(clip,range=0.0..0.15):; np:=noiseprof(noise,256):;
```

Now call the `noisered` function with $\beta = 0.02$:

```
> c:=noisered(channel_data(clip),np,0.02):;
  cleaned:=createwav(c):;
  plotwav(cleaned)
```



Observe that the noise level is significantly lower than in the original clip. You could use the `playsnd` command to compare the input with the output by hearing, which would reveal that the noise is still present but in a lesser degree. The parameter β controls how much noise is left in.

21.4.5 Short-time Fourier transform

The `stft` command computes the short-time Fourier transform (STFT) of a real signal. Input data is split into chunks of the specified size with half-overlap, and FFT is computed on each chunk separately, producing a time-evolving spectrum.

- `stft` takes one mandatory argument and one or two optional arguments:
  - L, a list of n real numbers.
  - Optionally, m, a positive integer which is a power of 2 (by default, $m = 128$). This is the half-size of a chunk.
  - Optionally, *wf*, a window function (by default, *wf* = `hamming_window`).
- `stft(`L $\langle, m, wf\rangle$`)` returns a matrix with $N = n/m - 1$ rows and m columns, where n is first snapped to the smallest multiple of m greater than or equal to n (the list L is padded with zeros). The kth row of the matrix is the left half of the FFT spectrum (since the signal L is real, we only go up to the Nyquist frequency) of the kth chunk of length $2m$ starting at offset km for $k = 0, 1, \ldots, N - 1$. The window function *wf* is applied to each chunk before computing FFT.

The `istft` command computes the inverse short-time Fourier transform by restoring the individual chunks and summing them up to restore the original signal.

- `istft` takes S, a matrix of complex numbers with N rows and m columns.
- `istft` returns L, a list of $(m + 1)N$ real numbers represeting the signal with STFT equal to S. Note that the restoration is generally not perfect: the restored signal will have a fade-in and fade-out of length m, resulting from applying the window function when perfoming STFT, and possibly trailing zeros at the end. Also, the window function for STFT must be chosen so that it sums to a constant on the half-overlap (e.g. Hamming, Hann, and Bartlett-Hann window, see Section 21.5, p. 607).

The STFT is a tool for spectral manipulation of a signal. Individual chunk spectra can be modified (e.g. in order to remove the noise or do a sharp lowpass/bandpass/highpass filtering) after calling `stft` and before calling `istft`. Overlap summation in inverse STFT provides for seamless crossfading between the modified chunks.

Note that by choosing the window size m you also set the number of bins for frequency spectra to m. Thus smaller m means finer time resolution and coarser frequency resolution, while larger m means coarser time resolution and finer frequency resolution.

Example

With the file `CantinaBand60.wav` downloaded from here, enter:

```
> music:=readwav("/home/luka/Downloads/CantinaBand60.wav")(0.0..15.0)
```

a sound clip with 330750 samples at 22050 Hz (16 bit, mono)

To hear the loaded audio, enter:

```
> playsnd(music)
```

The clip contains 15 seconds of a band playing. To hear only the bass part and hi-hat drum, you can use `stft` to transform the original clip, apply a spectral band-reject filter between 180 and 6000 Hz, and obtain the filtered version with `istft`. In this example the filter decays over time, so it appears as if the rest of the band enter with a fade-in.

A window of half-size $m = 8192$ is used in order to obtain a good frequency resolution. Enter:

```
> data,N,lo,hi,sr:=channel_data(music),8192,180,6000,samplerate(music):;
```

Boundary bin indices for the band-reject filter are obtained by multiplying m resp. n with the quotient of `lo` resp. `hi` and the upper frequency bound of the spectrum (i.e. the Nyquist frequency `sr/2`):

```
> m,n:=round(2*N*lo/sr,2*N*hi/sr)
```

134, 4458

To compute the STFT:

```
> spctrm:=stft(data,N):;
```

To attenuate the bins between m and n from zero to one over time, enter:

```
> tmp:=tran(spctrm):;
  alpha:=linspace(0.0,1.0,length(spctrm)).^4:;
  band:=mid(tmp,m,n-m):;
  for k from 0 to n-m-1 do band[k]=<band[k].*alpha; od:;
> spctrm2:=tran(concat(left(tmp,m),band,right(tmp,N-n+m))):;
```

The last commandline assembles the modified spectrum. Now compute the inverse STFT and create an audio clip from the obtained data:

```
> fadein:=createwav(istft(spctrm2),samplerate=sr,normalize=-1)
```

a sound clip with 341366 samples at 22050 Hz (16 bit, mono)

Now by entering

```
> playsnd(fadein)
```

you can clearly hear the bass part accompanied by the hi-hat rhythm, but no other instruments in the beginning. About halfway into the excerpt, other instruments fade in gradually, reaching the full intensity in the end.

21.4.6 Hilbert transform

Hilbert transform of a function f of a single real variable x is defined by:

$$\mathcal{H}\{f\}(x) = \frac{1}{\pi}\int_{-\infty}^{+\infty} \frac{f(\xi)}{x-\xi}\,\mathrm{d}\xi, \tag{21.1}$$

with the integral being taken as a Cauchy principal value. (Note that this implies $\mathcal{H}\{\text{const}\} = 0$.) The operator $\mathcal{H}$ is anti-involution, i.e. the inverse transform can be obtained by using the identity

$$\mathcal{H}\left\{\mathcal{H}\{f\}\right\} = -f. \tag{21.2}$$

Other basic properties include, letting $g = \mathcal{H}\{f\}$:

- linearity: $\mathcal{H}\{\alpha_1 f_1(x) + \alpha_2 f_2(x)\} = \alpha_1 \mathcal{H}\{f_1\}(x) + \alpha_2 \mathcal{H}\{f_2\}(x)$,
- time shifting: $\mathcal{H}\{f(x-a)\} = g(x-a)$,
- scaling: $\mathcal{H}\{f(ax)\} = \operatorname{sgn}(a) g(ax)$,
- derivative: $\mathcal{H}\left\{\frac{\mathrm{d}f}{\mathrm{d}x}\right\} = \frac{\mathrm{d}g}{\mathrm{d}x}$,
- orthogonality: $\int_{-\infty}^{+\infty} f(x)g(x)\mathrm{d}x = 0$.

The `hilbert` command finds Hilbert transforms. See Section 14.2.2, p. 325 and Section 21.4.7, p. 594 for other uses of `hilbert`.

- to compute (21.1), `hilbert` takes one mandatory argument and one optional argument:
  - *expr*, an expression defining the function f.
  - Optionally, x, a (real) variable (by default, $x =$ `x`).
- `hilbert(`*expr*$\langle$`,`$x\rangle$`)` returns the Hilbert transform $\mathcal{H}\{expr\}$ of *expr* with respect to the variable x. If `hilbert` unable to find the transform, the inert variant `Hilbert` will appear in the result. If the transform does not exist, then `undef` is returned.

Examples

There is no dedicated Xcas command for the inverse Hilbert transform, so you have to use the identity (21.2). Indeed:

```
> Hy:=hilbert(y(x),x)
```

$$\mathcal{H}_x\left\{y\left(x\right)\right\}$$

```
> -hilbert(Hy,x)
```

$$y\left(x\right)$$

For example:

```
> h:=hilbert(cos(a*x))
```

$$\operatorname{sign}\left(a\right)\sin\left(ax\right)$$

```
> -hilbert(h)
```

$$\cos\left(ax\right)$$

Other examples of Hilbert transform pairs follow.

Proper rational functions without poles can be transformed (since they have zero limit at infinity) as long as their denominator factorizations are simple enough. For example:

```
> assume(a>0):;
  hilbert(a/(t^2+a^2),t)
```

$$\frac{t}{a^2+t^2}$$

```
> rf:=sqrt(2)*(x-x^3+1)/(1+x^4)
```

$$\frac{\sqrt{2}\left(-x^3+x+1\right)}{x^4+1}$$

```
> hrf:=hilbert(rf,x)
```

$$\frac{x^3+2x^2+x}{x^4+1}$$

To demonstrate the orthogonality property, enter:

```
> int(rf*hrf,x=-inf..inf)
```

$$0$$

To demonstrate the derivative invariance, enter:

```
> ratnormal(hilbert(diff(rf))-diff(hrf))
```

$$0$$

Examples with transcedental functions:

```
> purge(a):;
  hilbert(exp(-i*a*x),x)
```

$$\mathrm{isign}\,(a)\,\mathrm{e}^{-\mathrm{i}ax}$$

```
> hilbert(sinc(t),t)
```

$$\frac{-\cos t+1}{t}$$

```
> hilbert(sin(t)/t^3,t)
```

Output (after expansion):

$$-\pi\,\delta\,(t,1)-\frac{\cos t}{t^3}-\frac{1}{2t}+\frac{1}{t^3}$$

Characteristic function transform:

```
> hilbert(boxcar(-2,1,x))
```

$$\frac{-\ln|-x+1|+\ln|-x-2|}{\pi}$$

All of the above transforms can be transformed back by applying `hilbert` again and changing the sign.

21.4.7 Analytic representation of a real signal

Hilbert transform is the key component of the *analytic representation* of a real signal (see here for details). Briefly, given a real signal $u(t)$, its analytic representation is $u_a(t)=u(t)+\mathrm{i}\mathcal{H}\{u\}(t)$, the main property of which is that its imaginary part is the Hilbert transform of the real part, the latter being exactly the original signal. As a consequence, the spectrum of u_a does not contain negative frequency components.

The `hilbert` command can find analytic representation u_a of a discrete real signal u by using the algorithm of Marple (1999) which uses FFT as a subroutine. See Section 14.2.2, p. 325 and Section 21.4.6, p. 592 for other uses of `hilbert`.

- To find u_a, `hilbert` takes u, a list $[u_1,u_2,\ldots,u_n]$ of real numbers.
- `hilbert(`u`)` returns the analytic representation u_a of u as a list of complex numbers $z_i=u_i+\mathrm{i}v_i$ for $i=1,2,\ldots,n$, where $v=[v_1,v_2,\ldots,v_n]$ is the Hilbert transform of u.
- Note that, in order to obtain a well-behaved Hilbert transform of the input signal, the latter should have the mean zero (or close to it). Namely, since $\mathcal{H}\{c\}=0$ for $c\in\mathbb{R}$, the inverse transform is unable to restore the original signal if the latter is shifted by a nonzero amount.

Examples

Given a signal defined by:

```
> f:=[0.7,0.2,-0.5,-0.6,-0.2,0.3,0.5,-0.1,-0.3]:;
```

you get its analytic representation with

```
> h:=hilbert(f):;
```

which is valid since

```
> mean(f)
```

$$1.97372982156 \times 10^{-16}$$

Indeed:

```
> re(h)
```

and

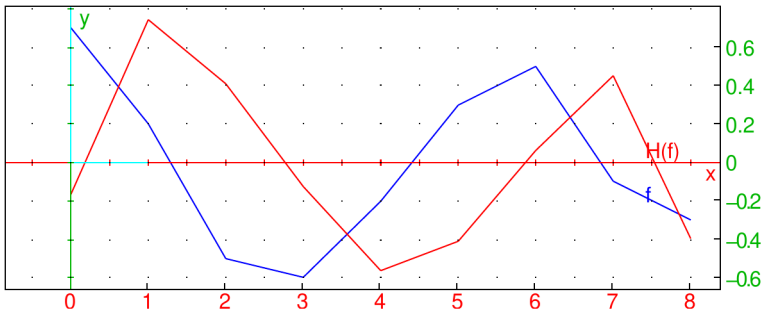
```
> -im(hilbert(im(h)))
```

both return

$$[0.7, 0.2, -0.5, -0.6, -0.2, 0.3, 0.5, -0.1, -0.3]$$

To visualize the input signal and its Hilbert transform together, enter:

```
> listplot(f,color=blue,legend="f");
  listplot(im(h),color=red,legend="H(f)");
```



The above image illustrates the fundamental property of the Hilbert transform: it shifts the phase of the input signal by $\pm 90°$.

Instantaneous amplitude, frequency and phase

You can get instantaneous amplitude and frequency of the original signal from its analytic representation. Since $u_a(t) = A(t)e^{i\varphi(t)}$ for some A and φ, the instantaneous amplitude is $A(t)$ and the instantaneous (angular) frequency is given by $\omega(t) = \frac{d\varphi}{dt}$. Indeed, this applies to the original signal because $u(t) = \Re(u_a(t)) = A(t)\cos(\varphi(t))$ by Euler's formula. For sampled signals, $\omega(t)$ is obtained in radians per sample and, assuming that r is the sample rate of u, the instantaneous frequency in Hertz is thus given by $f(t) = \frac{r}{2\pi}\omega(t)$. Since $f(t) \leq \frac{r}{2}$ by Nyquist theorem and $\omega(t) \geq 0$, it follows that $0 \leq \omega(t) \leq \pi$.

For a continuous, differentiable signal u, the instantaneous frequency can be obtained as

$$\omega(t) = -iN^*(t)N'(t) = \Im\left(N^*(t)N'(t)\right), \tag{21.3}$$

where $N^*(t)$ denotes the complex conjugate of the normalized analytic signal $N(t) = u_a(t)/A(t)$ and N' the time derivative of N (Vesnaver, 2017).

The phase $\varphi(t)$ is "wrapped" to $[-\pi, \pi]$ because of the periodicity of $e^{i\varphi(t)}$. Therefore it may contain an arbitrary number of first-order discontinuities when defined by $\varphi(t) = \arg(N(t))$. One way of defining the "unwrapped phase" for a causal signal u is:

$$\varphi(t) = \varphi_0 + \int_0^t \omega(\tau)\mathrm{d}\tau, \quad t \geq 0, \tag{21.4}$$

where $\varphi_0 = \arg(N(0))$ and ω is computed from (21.3). Indeed, φ in (21.4) is non-decreasing because $\omega(t) \geq 0$. Note that unwrapping the phase introduces uncertainties and requires user-defined parameters; in the above definition, φ_0 can have infinitely many values.

The instantaneous amplitude of u is obtained simply by calling the `abs` command with u_a as its argument. The vector of squared amplitudes is called the *energy* of u.

The `instfreq` command finds instantaneous frequency of a discrete signal u given its analytic representation u_a.

- `instfreq` takes one mandatory argument and one optional argument:
  - u_a, the analytic representation of a real signal u, which may be a vector (as returned by `hilbert`) or a symbolic expression.
  - Optionally, x, a variable (by default `x`).
- `instfreq(`u_a $\langle, x\rangle$`)` returns ω as a vector of phase advancements of u_a in the discrete case and the expression (21.3) in the continuous case. Precisely, if $u_a = [z_1, z_2, \ldots, z_n]$, then ω is a vector of length $n-1$ with components

  $$\omega_i = \arg\left(z_i^* z_{i+1}\right) \in [0, \pi], \quad i = 1, 2, \ldots, n-1.$$

  The above formula handles phase wrapping.
- In the discrete case, ω_i is the angular frequency of u at offset $i+1$, where $i = 1, 2, \ldots, n-1$.

The `instphase` command finds instantaneous phase of a discrete signal u given its analytic representation u_a, without wrapping.

- `instphase` takes one mandatory argument and one optional argument:
  - u_a, the analytic representation of a real signal u, which may be a vector (as returned by `hilbert`) or a symbolic expression.
  - Optionally, x, a variable (by default `x`).
- `instphase` returns φ as a cummulative-sum vector of `instfreq(`u_a`)` (adjusted so that the first element matches the initial phase) in the discrete case and the expression (21.4) in the continuous case.
- In the discrete case, φ_i is the phase of u at offset i, where $i = 1, 2, \ldots, n$.

Examples

Sample the function $x(t) = \sin(20\pi \sin(t))$ in the segment $\left[0, \frac{\pi}{2}\right]$ and apply the Hamming window:

```
> x:=evalf(linspace(0,pi/2,1000)):;
  y:=hamming_window(apply(t->sin(20*pi*sin(t)),x)):;
```

Compute the analytic representation of y and the relative error (in percentages) made by reconstructing the original signal from h:

```
> h:=hilbert(y):;
  norm(y+im(hilbert(im(h))))/norm(y)*100
```

2.08546075156

The error is relatively small (about 2%), hence y is applicable for further analysis (its analytic representation is well-behaved).

The instantaneous amplitude is obtained by:

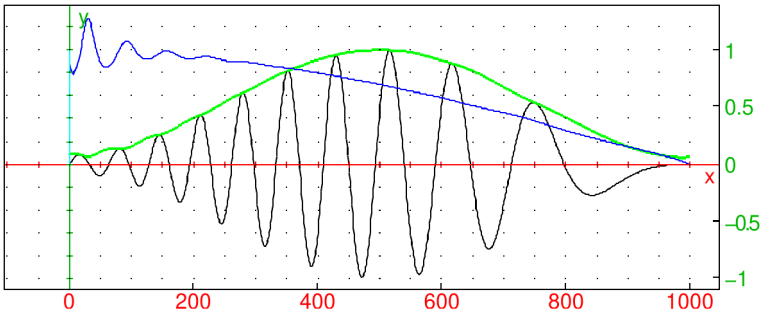
```
> a:=abs(h):;
```

and the instantaneous frequency by

```
> f:=instfreq(h):;
```

To plot y, a and f together, you can enter (f is scaled arbitrarily to be visible in the amplitude axis):

```
> listplot(y);
  listplot(a,color=green+line_width_2);
  listplot(10*f,color=blue)
```



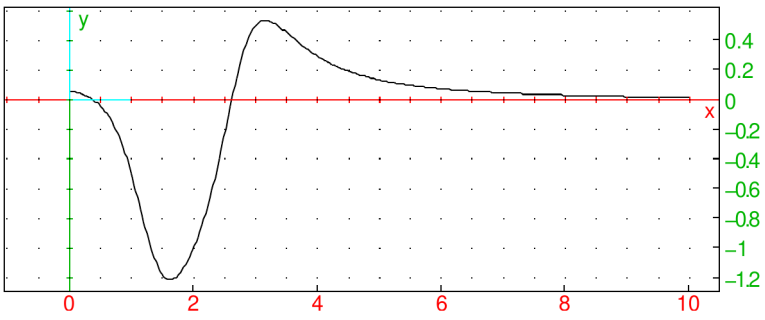
As you can see in the image above, the instantaneous amplitude (shown in green) forms an envelope of the original signal (shown in black), while the instantaneous frequency (shown in blue) drops as the oscillations in the original signal are slowing down (note that its values are nonnegative).

Continuous analytic signal. To obtain the instantaneous frequency of a continuous signal you can use (21.3) or the `instfreq` command. For example:

```
> x:=(t-3+(t-2)^2)/(1+(t-2)^4)
```

$$\frac{(t-2)^2+t-3}{(t-2)^4+1}$$

```
> plot(x,t=0..10)
```



To compute $y = \mathcal{H}\{x\}$, enter:

```
> y:=normal(hilbert(x,t))
```

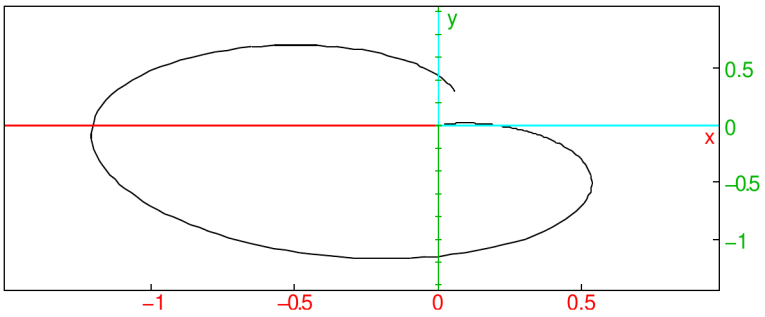
$$\frac{t^2-6t+7}{\sqrt{2}t^4-\left(8\sqrt{2}\right)t^3+24\sqrt{2}t^2-\left(32\sqrt{2}\right)t+17\sqrt{2}}$$

The analytic representation of x and its normalization are obtained by:

```
> z:=x+i*y:;
```

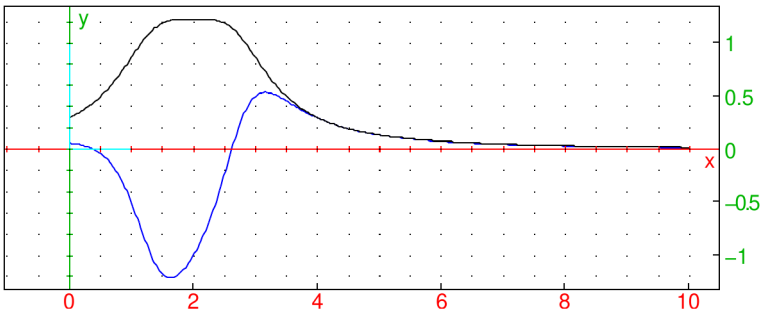
To animate unfolding of $z(t)$ in the complex plane for $0 \le t \le 10$, enter:

```
> animation(seq([plotparam(z,t=0..s/5),vector(subs(z,t,s/5),color=blue)],s=1..50))
```



To plot the instantaneous amplitude $A(t)$ against $x(t)$, enter:

```
> plotfunc([x,abs(z)],t=0..10,color=[blue,black])
```



The instantaneous frequency of x is obtained by entering:

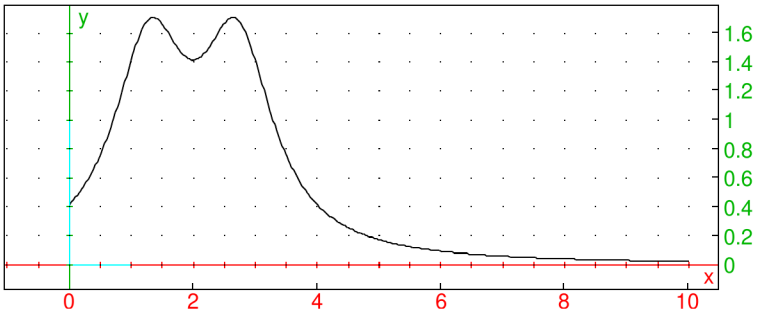
```
> ifreq:=instfreq(z,t)
```

$$\frac{t^2\sqrt{2} - 4t\sqrt{2} + 5\sqrt{2}}{t^4 - 8t^3 + 24t^2 - 32t + 17}$$

```
> ifreq>=0
```

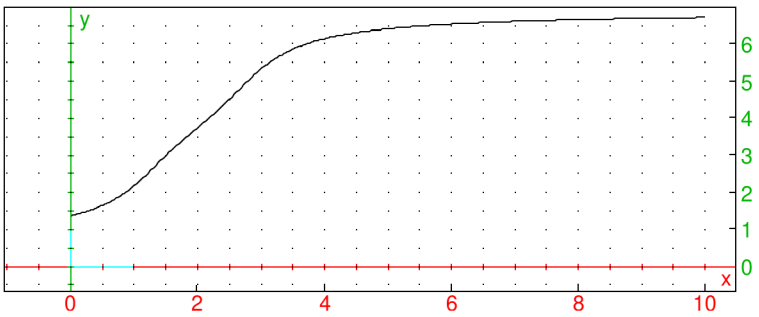
true

```
> plot(ifreq,t=0..10)
```



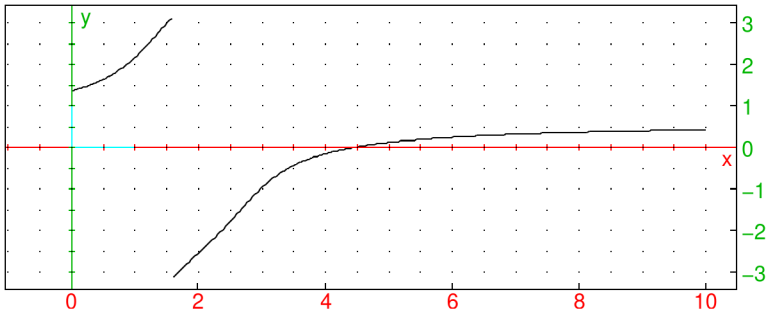
The unwrapped phase, according to the definition (21.4), is now obtained by entering:

```
> iphas:=instphase(z,t):;
  plot(iphas,t=0..10)
```



The wrapped phase, on the other hand, suffers from a discontinuity at $t = 3 - \sqrt{2}$:

```
> plot(arg(z),t=0..10)
```



As another example, show that the function $u(t) = \frac{a}{a^2+t^2}$ for $a > 0$ is itself its instantaneous frequency and, in case $a = 1$, also its energy. To verify, first compute the (normalized) analytic representation by entering:

```
> purge(a):; assume(a>0):;
  u:=a/(a^2+t^2):;
  z:=u+i*hilbert(u,t):;
  N:=simplify(z/abs(z))
```

$$\frac{\sqrt{a^2+t^2}}{a-it}$$

Then compute the instantaneous frequency and energy of u, respectively, by entering:

```
> instfreq(z,t),simplify(abs(z)^2)
```

$$\frac{a}{a^2+t^2}, \frac{1}{a^2+t^2}$$

21.4.8 Empirical mode decomposition

Empirical mode decomposition (EMD) is an adaptive method for decomposing a nonstationary real signal into a sequence of *intrinsic mode functions* (IMFs). Each IMF will represent an intrinsic oscillation of the input signal and satisfies the following two conditions:

1. The number of local extrema must be equal to the number of zero crossings or differ from it by at most one.

2. The mean value of two envelopes interpolating local minima and local maxima, respectively, must be zero at any time.

The sequence of IMFs represents a nearly orthogonal basis for the input signal.

An IMF is extracted from the input signal by a technique referred to as *sifting*. In each iteration, the mean of the upper and lower envelope is subtracted from the input signal. The process ends when the variance of the mean vector is below a predefined threshold v_0. Then the obtained IMF is subtracted from the original signal and the procedure is iteratively repeated until the number of local extrema goes below a predefined threshold (usually 2); the rest of the signal is called the *residue* or *trend* (it is not an IMF).

The `emd` command computes empirical mode decomposition (GSL is required for cubic spline interpolation).

- `emd` takes one mandatory argument and a sequence of optional arguments:
  - x, a vector of real numbers representing the input signal.
  - Optionally, *opts*, a sequence of options each of which is one of:
    - `threshold=`t, where t is a positive real number specifying the variance threshold v_0 (by default, $th = 0.1$).
    - `limit=`n, where n is a positive integer specifying the maximum number of IMFs to be extracted (by default, n is unset, meaning that all IMFs are extracted).
    - `extrema=`m, where m is a positive integer specifying the minimal number of local extrema that an IMF must contain (by default, $m = 2$).
    - `residue=`*bool*, where *bool* is a boolean value which specifies whether to return the residue (*bool* = `true`, the default) or not (*bool* = `false`).
- If `residue` = `true`, then `emd(`x ⟨, *opts*⟩`)` returns a sequence of two objects: the list of IMFs and the residue, respectively. If `limit` option is set, then the second element of the returned sequence is the unprocessed part of the input signal x. The returned IMFs and the residue always sum up to x.
- If `residue` = `false`, then `emd(`x ⟨, *opts*⟩`)` returns only the list of computed IMFs.

A convenient way to visualize intrinsic mode functions is to use the `plotimf` command, which plots (a selection of) IMFs in a stacked layout, optionally including the input signal and/or the residue. `imfplot` is a synonym for `plotimf`.

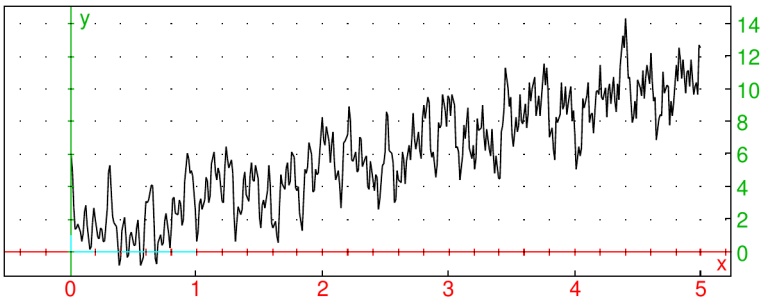
- `plotimf` takes one mandatory argument and a sequence of optional arguments:
  - *imfs*, a list of intrinsic mode functions obtained by the `emd` command.
  - Optionally, either n, a positive integer specifying that only the first n IMFs should be displayed, or *opts*, a sequence of options each of which is one of the following:
    - `count=`n, an alternative way to specify the parameter n (see above).
    - `index=`*ind*, where *ind* is an integer k, a range $k_1..k_2$ where k_1 and k_2 are integers, or a list of integers and/or ranges, specifying the indices determining a selection of IMFs. If this option is set, only IMFs with specified indices are shown.
    - `max=`*bool*, where *bool* is a boolean value specifying whether to display the upper y-bound for each IMF plot (*bool* = `true`, the default) or not (*bool* = `false`).
    - `residue=`*res*, where *res* is the residual signal. If this option is set, then *res* is plotted below the IMFs.
    - `input=`*orig*, where *orig* is the original signal. If this option is set, then *orig* is plotted above the IMFs.
    - `display=`*disp* or `color=`*disp*, where *disp* specifies individual graphic attributes which are to be used in each plot (such as the color, line style, and the like, see Section 19.1.2, p. 454). By default, plots are drawn in blue.

- `plotimf(`*imfs*`⟨,` *n*`⟩)` and `plotimf(`*imfs*, *opts*`)` return a graphical object containing plots of the selected EMD components.

Example

To define a synthetic non-stationary signal, enter:

```
> f(t):=2t+cos(10t^2+100t)+cos(60t)+cos(40t)+cos(t^2+20t)+cos(t^2/2+5t)+1:;
  plot(f(t),t=0..5)
```



To discretize the signal, enter:

```
> synth:=apply(f,linspace(0,5,500)):;
```

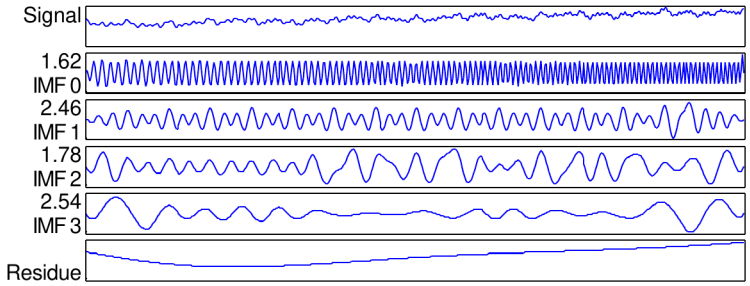
Now compute EMD and output the total number of intrinsic mode functions:

```
> imf,res:=emd(synth):; size(imf)
```

9

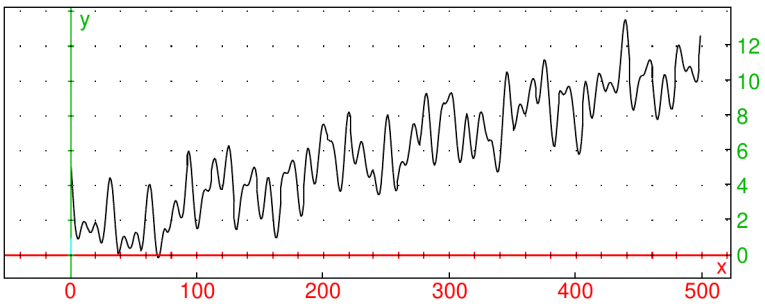
To show the first four IMFs together with the residue and the original signal, enter:

```
> imfplot(imf,residue=res,input=synth,count=4)
```



The first IMF captures the high-frequency oscillations which can be filtered by removing that component. Effectively, you sum all but the first IMF and also add the residue to obtain the filtered signal. The first IMF can be discarded by using the `tail` command.

```
> listplot(res+sum(tail(imf)))
```



21.4.9 Hilbert-Huang transform

The `hht` command finds Hilbert spectra by applying the *Hilbert-Huang transform* (HHT) which obtains instantaneous frequency data from intrinsic mode functions (IMFs) extracted from nonstationary and/or nonlinear signal data. The Hilbert spectrum of a signal reveals its time-dependent features.

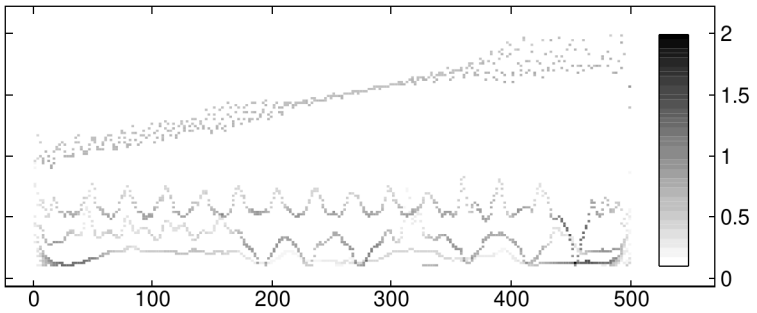
- `hht` takes one mandatory argument and a sequence of optional arguments:
  - *data*, a list of real numbers (representing the signal) or a matrix in which each row corresponds to an IMF as obtained by empirical mode decomposition (see Section 21.4.8, p. 599).
  - Optionally, *opts*, a sequence of optional arguments each of which is one of:
    - `frequencies=`*frng*, where *frng* is either a list `[`*fmin*, *fmax*`]` or an interval *fmin*`..`*fmax*, where *fmin* and *fmax* specify the desired frequency range. The values are in Hertz if sample-rate is provided, otherwise in radians per sample. If *frng* is a list, then *fmin* and *fmax* can also be strings specifying MIDI notes such as e.g. `"C4"` and `"D#5"`. By default, *fmin* corresponds to the zero frequency and *fmax* to the Nyquist frequency.
    - `range=`*trng*, where *trng* is either a list `[`*tmin*, *tmax*`]` or an interval *tmin*`..`*tmax*, where *tmin* and *tmax* specify the desired time range. If sample-rate is provided, then the unit is one second, otherwise the unit is one sample. By default, the entire signal is analyzed.
    - `samplerate=`*sr*, where *sr* is a positive integer representing the sample-rate of the (decomposed) signal (by default, unset).
    - `output=`*out*, where *out* is one of:
      - `image` (the default), which displays the spectrum as an image.
      - `plot`, which draws the spectrum with Xcas graphic commands (in lower resolution).
      - `matrix`, which outputs the spectrum as a matrix in which rows correspond to frequency bins and columns to spectra at individual samples.
    - `bins=`*m*, where *m* is a positive integer specifying the desired number of bins for the frequency range discretization. By default, $m = 200$ for the image output and $m = 100$ for the plot and matrix outputs.
    - `log=`*islog*, where *islog* is a boolean value specifying whether the frequency axis is logarithmic or not (by default, the scale is linear). If *islog* = `true`, then the frequency-axis unit is one semitone, with zero corresponding to the MIDI note 0. Instead of `log=true`, you can simply enter `log`.
    - `legend=`*haslegend*, where *haslegend* is a boolean value specifying whether to show the colormap or not (by default, the colormap is shown on the right of the spectrum).
    - `tran=`*hasalpha*, where *hasalpha* is a boolean value specifying whether to use the alpha channel in the image output (this value is ignored for other outputs). When used, the alpha channel values are smaller for smaller amplitudes, making the lower-amplitude components less visible. By default, `tran=true`.
    - `color=`*cmap*, where *cmap* is the colormap index (nonnegative integer) as returned by the `colormap` command (see Section 19.1.3, p. 460). *cmap* can also be a boolean value, with `true` corresponding to the default rainbow colormap and `false` corresponding to the grey ramp with 24 colors from the FLTK colormap. A suitable choice of colormap can help detecting important signal features in the graphic output. By default, `color=true`.
- `hht(`*data*⟨, *opts*⟩`)` returns (the selected part of) the Hilbert spectrum associated with *data* in the specified form. If graphic output is selected, then the colormap is used to color-encode amplitude data. The axis units are seconds/samples resp. Hertz/radians per second/semitones, depending on the values of `samplerate` and `log` options.

- With matrix output, you can use `hht` to compute the signal energy as a function of time. The element at position (i, j) in the resulting matrix, when squared, represents the energy of the ith frequency bin at the jth sample. The total energy at time j is the sum of squares of elements in the jth column of the matrix.

Examples

Synthetic signal. To see the spectrum of the `synth` signal defined in Section 21.4.8, p. 599 with the FLTK grayscale ramp, enter:

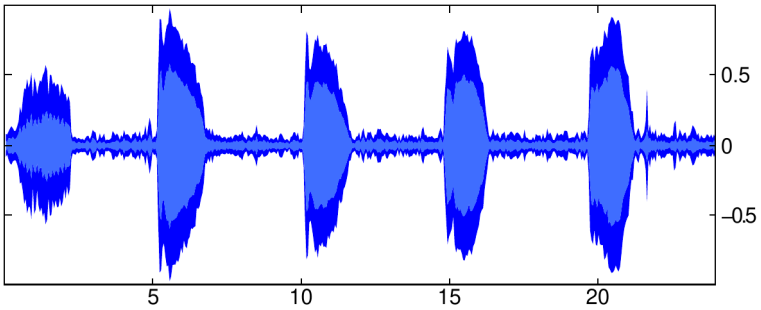
```
> hht(synth,color=false,frequencies=0.1..2.0,output=plot)
```



You may also pass the list of already computed IMFs as the first argument.

Whale song. The file `nepblue24s10x.wav` downloaded from here contains sounds produced by a blue whale in the northeast Pacific. Since the frequency of moans is very low (below 20 Hz, which is inaudible for humans), the file is played back at 10x speed. To load and show the audio, enter:

```
> whale:=readwav("/home/luka/Downloads/nepblue24s10x.wav"):; plotwav(whale)
```



You can hear the whale song by entering (see Section 28.2.14, p. 833):

```
> playsnd(whale)
```

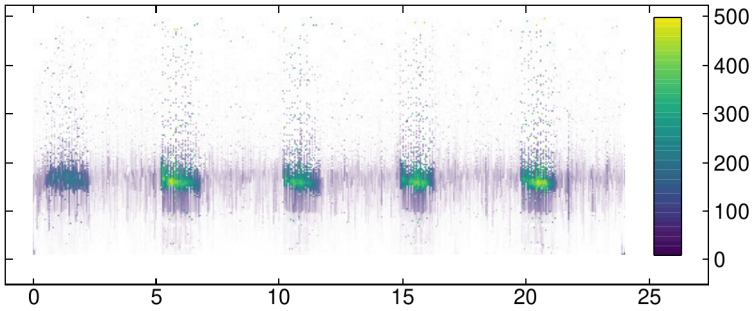
To perform the empirical mode decomposition (see Section 21.4.8, p. 599), enter:

```
> imf:=emd(channel_data(whale),residue=false):;
```

Evaluation time: 10.01

(Note that the residue is discarded since it is not relevant for the spectrum.) To display the Hilbert spectrum of the whale song, enter:

```
> sr:=samplerate(whale):;
  hht(imf,frequencies=10..500,samplerate=sr,color=colormap("viridis"))
```



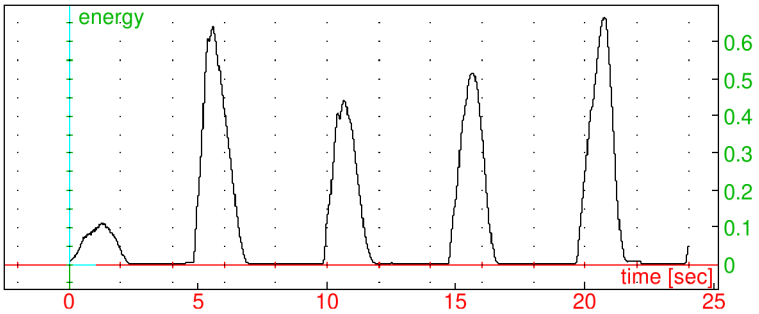
Whale moans occur at about 5, 10, 15 and 20 seconds from the beginning of the recording (in reality, these intervals are ten times larger). Also, the spectrum reveals that moans have the same frequency, about 160 Hz in the recording and 16 Hz in reality. The magnitude of the first moan is smaller, as suggested by both the waveform and spectrum.

To obtain the Hilbert spectrum as a matrix, enter:

```
> hs:=hht(imf,frequencies=10..500,samplerate=sr,output=matrix):;
```

To plot the energy of the whale song, sum the columns of the Hilbert spectrum matrix in which every element is squared. Energy data can be smoothed out by applying a moving-average filter.

```
> y:=moving_average(sum(hs.^2),4000):;
  x:=linspace(0,duration(whale),length(y)):;
  labels=["time [sec]","energy"];
  listplot(tran([x,y]))
```



Spectral analysis of music. With the file `CantinaBand3.wav` downloaded from here:

```
> music:=readwav("/home/luka/Downloads/CantinaBand3.wav")
```

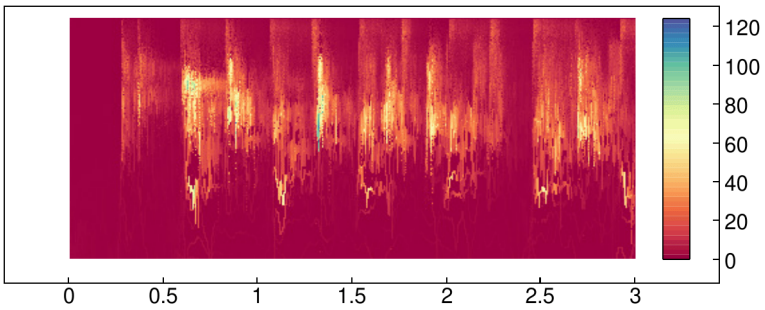
a sound clip with 66150 samples at 22050 Hz (16 bit, mono)

To obtain instrinsic mode functions, enter (residue is discarded):

```
> imf:=emd(channel_data(music),residue=false):;
```

To display the Hilbert spectrum of `music`, enter:

```
> hht(imf,samplerate=22050,log,color=colormap("spectral"),tran=false)
```



In the upper part of the spectrum you can see individual beats of the song, and in the lower part you can see six bass notes appearing every 2 beats. The first and fifth notes are stressed more than the others, suggesting the 4/4 time signature. The percussive pickup measure is also clearly visible. The first beat of the first measure starts after about 0.65 seconds. In the example in Section 21.4.5, p. 591 it is explained how to extract the rhythmic section by cutting out the middle band of the spectrum.

As another example, download the files `violin-C4.wav` and `flute-G6.wav` from here and load them:

```
> violin:=readwav("/home/luka/Downloads/violin-C4.wav");
  flute:=readwav("/home/luka/Downloads/flute-G6.wav")
```

a sound clip with 38500 samples at 11025 Hz (16 bit, mono),
a sound clip with 40250 samples at 11025 Hz (16 bit, mono)

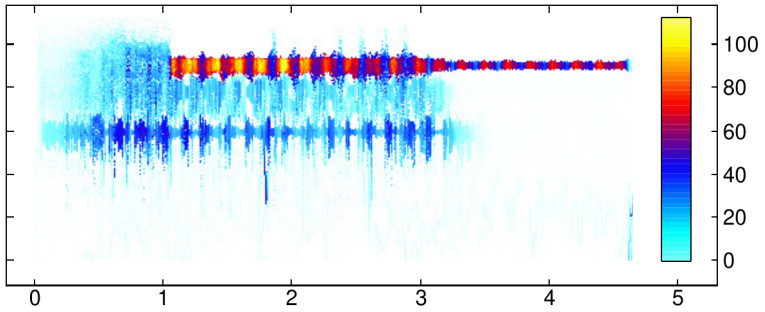
Mix the two audio clips with one-second delay for the flute (see Section 28.2.18, p. 835):

```
> snd:=mixdown(violin,[],flute,[1.0])
```

a sound clip with 51275 samples at 11025 Hz (16 bit, mono)

You can hear the result by using the `playsnd` command (see Section 28.2.14, p. 833). To display the Hilbert spectrum, enter (in this case, EMD is computed automatically by `hht`):

```
> data,sr:=channel_data(snd),samplerate(snd):;
  hht(data,log,samplerate=sr,color=colormap("jet",inv,reverse));
```



The specified colormap helps you to see two individual tones at 60 resp. 91 semitones from the MIDI note 0, corresponding to C4 resp. G6, as expected. The flute part of the spectrum (yellow-reddish) is much more concentrated than the violin part due to the inherent sinusoidal quality of the flute sound.

21.4.10 Discrete wavelet transform

The `dwt` and `idwt` commands are used for computing the discrete wavelet transform (DWT) of a signal and for computing the inverse transformation, i.e. reconstructing the original signal from the transform. These commands require the GSL library.

- `dwt` and `idwt` both take one mandatory argument and up to four optional arguments (the order of optional arguments is irrelevant):
  - *data*, a numeric vector of length n or matrix with dimensions $m \times n$.
  - Optionally, "*wtype*", a string which defines the wavelet family. Supported values of *wtype* are: `daubechies` (the default), `haar` and `bspline`.
  - Optionally, k, a positive integer which selects the specified member of the wavelet family. For the Daubechies wavelet family, $k \in \{4, 8, \ldots, 20\}$ (even numbers). For the Haar wavelet, only $k = 2$ is supported. For the biorthogonal B-spline wavelet family, the implemented values of k are 103, 105, 202, 204, 206, 208, 301, 303, 305 307 and 309. By default, $k = 4$ is selected for the Daubechies wavelet, $k = 2$ for the Haar wavelet and $k = 103$ for the B-spline wavelet.

  - Optionally, `center`, the symbol which makes use of the centered forms of the wavelets which align the coefficients of the various sub-bands on edges, making the resulting visualization of the coefficients of the wavelet transform in the phase plane easier to understand.
  - Optionally, `image`, the symbol which specifies the "non-standard" form of the 2D DWT.

  Note that the input array *data* will be enlarged, if necessary, such that its size is a power of two. If *data* is a matrix, it is augmented to a square matrix of size equal to the first power of 2 not smaller than $\max\{m, n\}$. Input data is extended by zero-padding.

- If *data* is a vector, then `dwt(`*data* $\langle$, *options*$\rangle$`)` returns the list of length n which contains the DWT of the signal *data*, packed in a triangular storage layout:

$$\left(s_{-1,0}, d_{0,0}, d_{1,0}, d_{1,1}, d_{2,0}, d_{2,1}, d_{2,2}, \ldots, d_{i,j}, \ldots, d_{N-1,2^{N-1}-1}\right),$$

  where $i = 0, \ldots, N-1$ is the level index, $j = 0, \ldots, 2^i - 1$ is the index of the coefficient within each level, and $N = \log_2(n)$. The first element $s_{-1,0}$ is the smoothing coefficient, which is followed by detail coefficients $d_{i,j}$ for each level i.

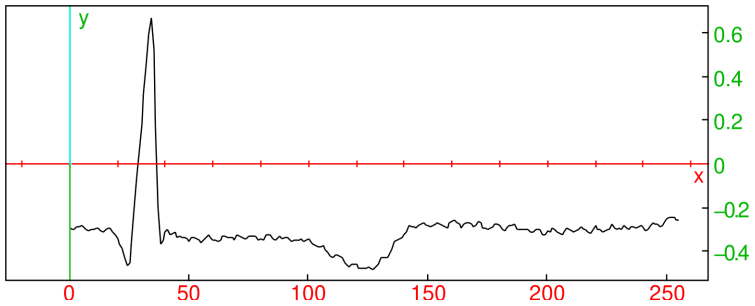
- If *data* is a matrix, then the "non-standard" 2D DWT is performed in interleaved passes on the rows and columns for each level of the transform. The non-standard form of DWT is typically used in image analysis. By default, the "standard" form is used, which performs a complete DWT on the rows of the matrix, followed by a separate complete DWT on the columns of the resulting row-transformed matrix.

- `idwt` takes the DWT and returns the original data. The *data* parameter may be modified prior to calling `idwt` for the sake of e.g. signal smoothing or data compression.

Examples

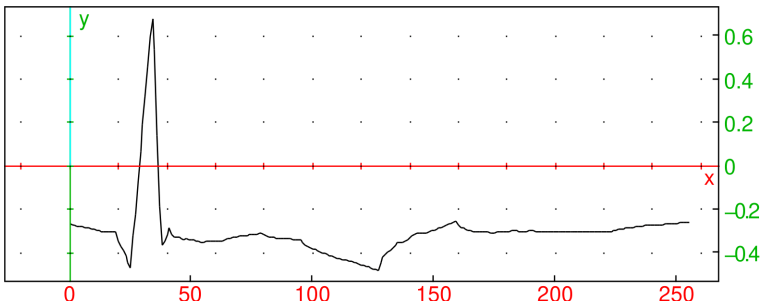
Signal compression. In the first example (adapted from GSL documentation), we load a sample of size 256 from the MIT-BIH Arrhythmia Database (see here) and attempt to remove the noisy information.

```
> data:=mid(col(csv2gen("/home/luka/Documents/MIT-BIH.csv","\t"),2),300,256):;
  listplot(data)
```



Now we transform the signal by using `dwt` and select the 20 largest components. We set other elements to zero and transform the result back using `idwt`.

```
> tdata:=dwt(data):;
  p:=reverse(sortperm(abs(tdata))):;
  for k from 20 to 255 do tdata[p[k]]:=0; od:;
  res:=idwt(tdata):;
  listplot(res)
```



As you can see, the fast, low-amplitude oscillations are gone, but important features such as the spike are left practically intact. As a beneficial side-effect of denoising, you get the signal compressed by, in this case, about 92% in terms of memory usage.

Noise reduction in images. Assuming that an image `noisy.jpg` is stored in `Pictures` folder, enter:

```
> img:=image("/home/luka/Pictures/noisy.jpg")(grey)
```

an image of size 300×300 (grayscale)

Use DWT to transform the image matrix and subsequently replace each component smaller than 55 (by absolute value) in the resulting matrix to zero, before transforming it back to obtain a noise-reduced image. (The threshold value was obtained by trial and error.) Note that `dwt` enlarges the input matrix to the size 512×512, so be sure to crop the output of `idwt` back to 300×300. To make sure that no component gets below zero or above 255, use the `threshold` command (see Section 21.3.4, p. 575).

```
> tdata:=dwt(img[0],image):;
  sdata:=threshold(tdata,55.0=0,'<=',abs=true):;
  nr:=image(1,threshold(round(subMat(idwt(sdata,image),0,0,299,299)),[0,255]))
```

an image of size 300×300 (grayscale)

To display the original image and the processed image next to each other for comparison, enter:

```
> display(img,0); display(nr,320);
  legend(-20i,"original"); legend(320-20i,"denoised")
```

21.5 Window functions

21.5.1 Bartlett-Hann window

The `bartlett_hann_window` command applies the Bartlett-Hahn window to a sequence or its segment.

- `bartlett_hann_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).

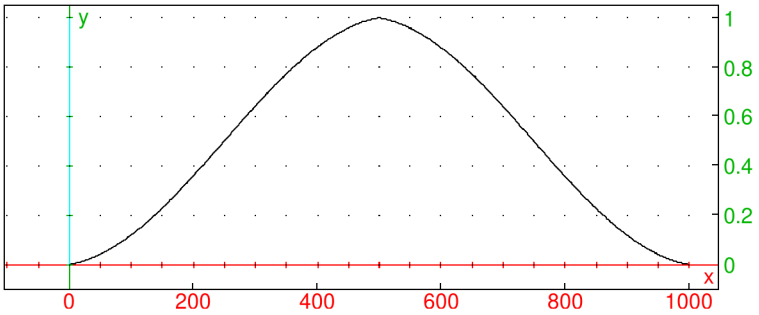
- `bartlett_hann_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = a_0 - a_1 \left| \frac{k}{N-1} - \frac{1}{2} \right| - a_2 \cos\left(\frac{2\,k\,\pi}{N-1}\right)$$

  for $k = 0, 1, \ldots, N-1$, where $a_0 = 0.62$, $a_1 = 0.48$ and $a_2 = 0.38$.

Example

```
> listplot(bartlett_hann_window([1$1000]))
```



21.5.2 Blackman-Harris window

The `blackman_harris_window` command applies the Blackman-Harris window to a sequence or its segment.

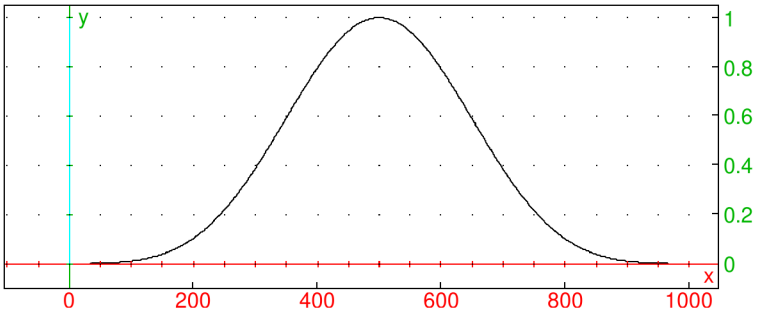
- `blackman_harris_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `blackman_harris_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = a_0 - a_1 \cos\left(\frac{2\,k\,\pi}{N-1}\right) + a_2 \cos\left(\frac{4\,k\,\pi}{N-1}\right) - a_3 \cos\left(\frac{6\,k\,\pi}{N-1}\right)$$

  for $k = 0, 1, \ldots, N-1$, where $a_0 = 0.35875, a_1 = 0.48829, a_2 = 0.14128$ and $a_3 = 0.01168$.

Example

```
> listplot(blackman_harris_window([1$1000]))
```



21.5.3 Blackman window

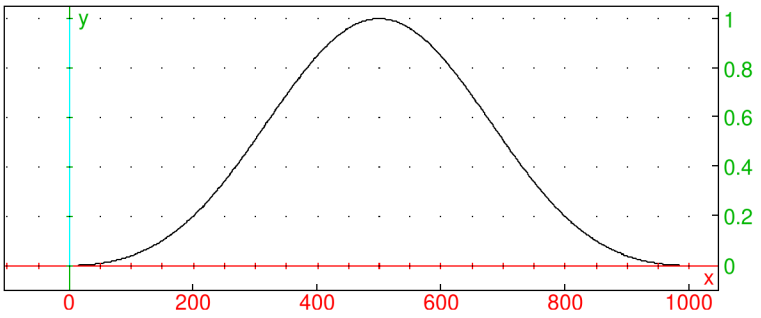
The `blackman_window` command applies the Blackman window function to a sequence or its segment.

- `blackman_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - α, a real number (by default, $\alpha = 0.16$).
    - $m, N \langle, \alpha\rangle$, a sequence of two integers and optionally a real number α (by default, $m = 0$ and $N = n$).
- `blackman_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \frac{1-\alpha}{2} - \frac{1}{2}\cos\left(\frac{2\,k\,\pi}{N-1}\right) + \frac{\alpha}{2}\cos\left(\frac{4\,k\,\pi}{N-1}\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(blackman_window([1$1000]))
```



21.5.4 Bohman window

The `bohman_window` command applies the Bohman window function to a sequence or its segment.

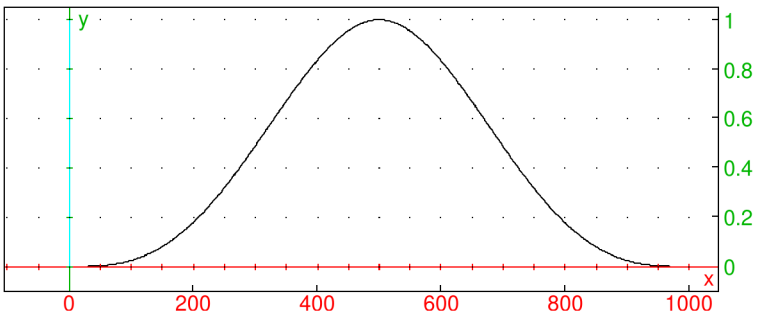
- `bohman_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `bohman_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = (1 - x_k)\cos(\pi\,x_k) + \frac{1}{\pi}\sin(\pi\,x_k),$$

  where $x_k = \left|\frac{2\,k}{N-1} - 1\right|$ for $k = 0, 1, \ldots, N-1$.

Example

```
> listplot(bohman_window([1$1000]))
```



21.5.5 Cosine window

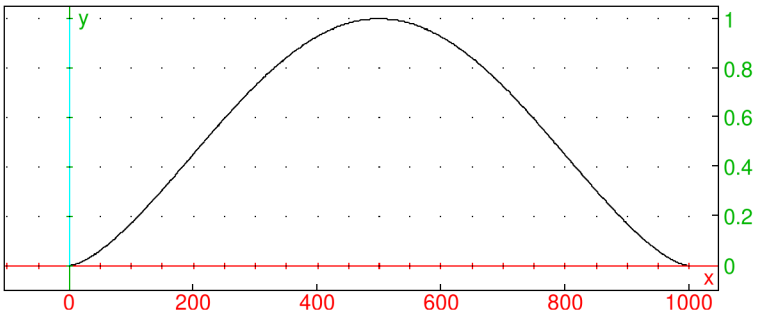
The `cosine_window` command applies the cosine window function to a sequence or its segment.

- `cosine_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - α, a real number (by default, $\alpha = 1$).
    - $m, N \langle, \alpha\rangle$, a sequence of two integers and optionally a real number α (by default, $m = 0$ and $N = n$).
- `cosine_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \sin^\alpha\left(\frac{k\,\pi}{N-1}\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(cosine_window([1$1000],1.5))
```



21.5.6 Gaussian window

The `gaussian_window` command applies the Gaussian window function to a sequence or its segment.

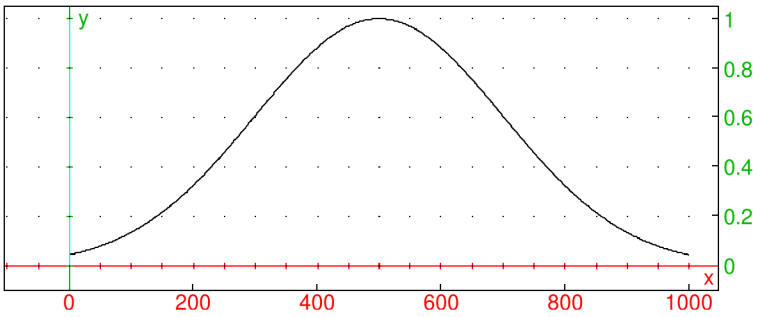
- `gaussian_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - α, a real number not larger than 0.5 (by default, $\alpha = 0.1$).
    - $m, N \langle, \alpha\rangle$, a sequence of two integers and optionally a real number $\alpha \leq 0.5$ (by default, $m = 0$ and $N = n$).

- `gaussian_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \exp\left(-\frac{1}{2}\left(\frac{k-(N-1)/2}{\alpha\,(N-1)/2}\right)^2\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(gaussian_window([1$1000],0.4))
```



21.5.7 Hamming window

The `hamming_window` command applies the Hamming window function to a sequence or its segment.

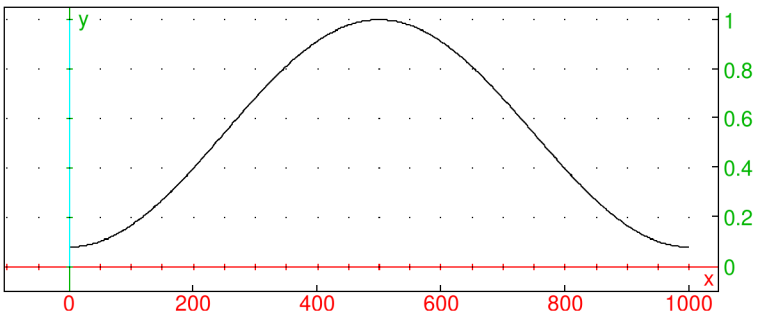
- `hamming_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `hamming_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \alpha - \beta\,\cos\left(\frac{2\,k\,\pi}{N-1}\right)$$

for $k = 0, 1, \ldots, N-1$, where $\alpha = 0.54$ and $\beta = 1 - \alpha = 0.46$.

Example

```
> listplot(hamming_window([1$1000]))
```



21.5.8 Hann-Poisson window

The `hann_poisson_window` command applies the Hann-Poisson window to a sequence or its segment.

- `hann_poisson_window` takes one mandatory argument and up to three optional arguments:

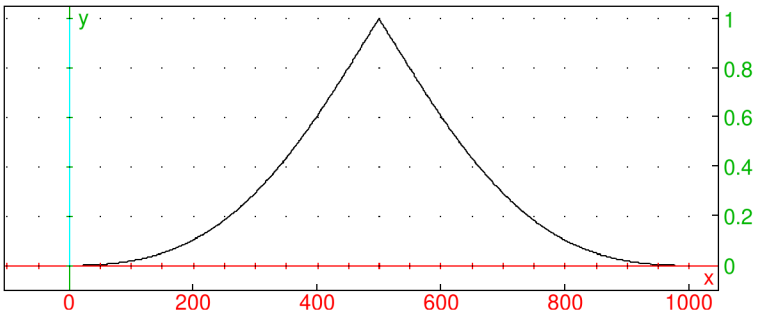
  - **v**, a real vector with length n.
  - Optionally, one of:
    * α, a real number (by default, $\alpha = 1$).
    * $m, N \langle, \alpha\rangle$, a sequence of two integers and optionally a real number α (by default, $m = 0$ and $N = n$).

- `hann_poisson_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \frac{1}{2}\left(1 - \cos\frac{2\,k\,\pi}{N-1}\right)\exp\left(-\frac{\alpha\,|N-1-2\,k|}{N-1}\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(hann_poisson_window([1$1000],2))
```



21.5.9 Hann window

The `hann_window` command applies the Hann window function to a sequence or its segment.

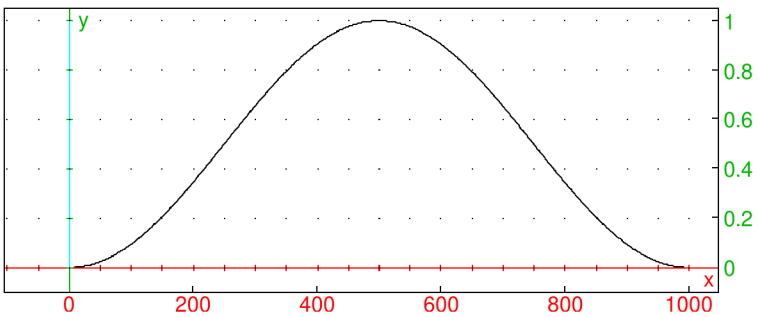
- `hann_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).

- `hann_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \sin^2\left(\frac{k\,\pi}{N-1}\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(hann_window([1$1000]))
```



21.5.10 Parzen window

The `parzen_window` command applies the Parzen window function to a sequence or its segment.

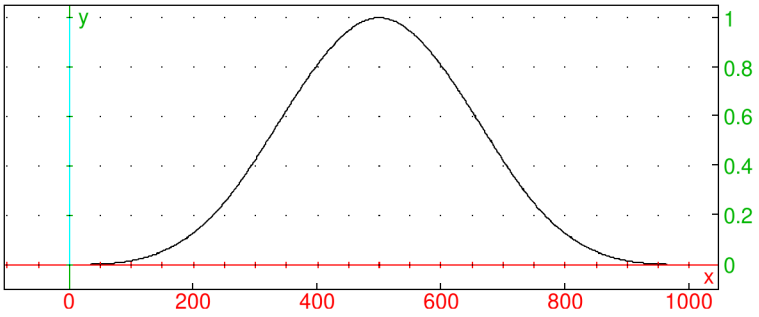
- `parzen_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `parzen_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \begin{cases} 1 - 6\,x_k^2\,(1 - x_k)\,, & \left|\frac{N-1}{2} - k\right| \le \frac{N-1}{4}, \\ 2\,(1 - x_k)^3\,, & \text{otherwise,} \end{cases}$$

  where $x_k = \left|1 - \frac{2\,k}{N-1}\right|$ for $k = 0, 1, \ldots, N-1$.

Example

```
> listplot(parzen_window([1$1000]))
```



21.5.11 Poisson window

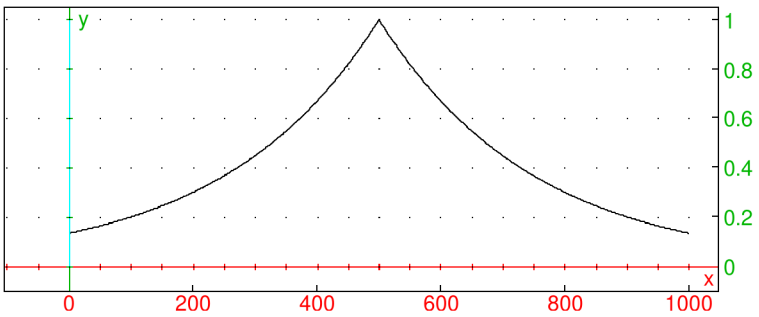
The `poisson_window` command applies the Poisson window function to a sequence or its segment.

- `poisson_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - α, a real number (by default, $\alpha = 1$).
    - $m, N\,\langle, \alpha\rangle$, a sequence of two integers and optionally a real number α (by default, $m = 0$ and $N = n$).
- `poisson_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \exp\left(-\alpha\left|\frac{2\,k}{N-1} - 1\right|\right), \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(poisson_window([1$1000],2))
```



21.5.12 Riemann window

The `riemann_window` command applies the Riemann window function to a sequence or its segment.

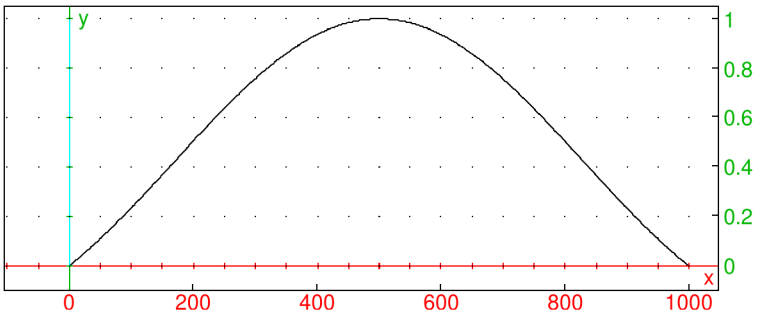
- `riemann_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `riemann_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \begin{cases} 1, & k = \frac{N-1}{2}, \\ \frac{\sin(\pi\, x_k)}{\pi\, x_k}, & \text{otherwise}, \end{cases}$$

  where $x_k = \frac{2\,k}{N-1} - 1$ for $k = 0, 1, \ldots, N-1$.

Example

```
> listplot(riemann_window([1$1000]))
```



21.5.13 Triangular window

The `triangle_window` command applies the triangle window function to a sequence or its segment.

- `triangle_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - d, an integer in $\{-1, 0, 1\}$ (by default, $d = 0$).
    - $m, N\,\langle, d\rangle$, a sequence of two integers and optionally an integer $d \in \{-1, 0, 1\}$ (by default, $m = 0$ and $N = n$).

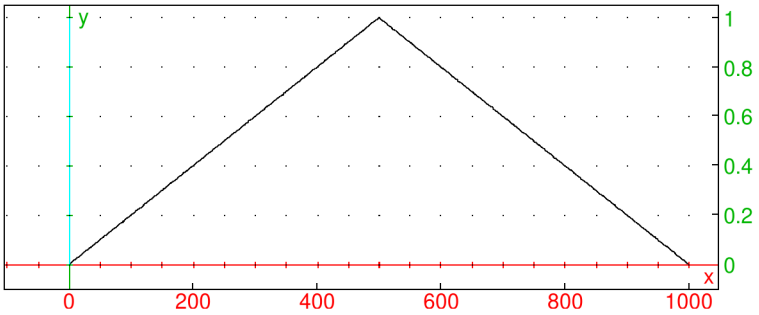
- `triangle_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = 1 - \left| \frac{k - \frac{N-1}{2}}{\frac{N+d}{2}} \right|, \quad k = 0, 1, \ldots, N-1.$$

The case $d = -1$ is called the Bartlett window function.

Example

```
> listplot(triangle_window([1$1000],1))
```



21.5.14 Tukey window

The `tukey_window` command applies the Tukey window function to a sequence or its segment.

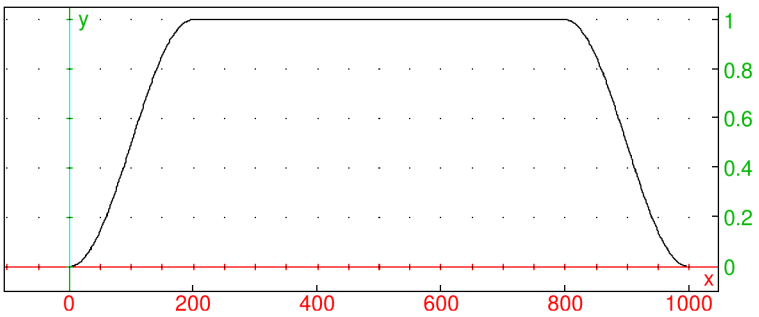
- `tukey_window` takes one mandatory argument and up to three optional arguments:
  - **v**, a real vector with length n.
  - Optionally, one of:
    - α, a real number in $[0, 1]$ (by default, $\alpha = 0.5$).
    - $m, N \langle, \alpha \rangle$, a sequence of two integers and optionally a real number $\alpha \in [0, 1]$ (by default, $m = 0$ and $N = n$).
- `tukey_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = \begin{cases} \frac{1}{2}\left(1 + \cos\left(\pi\left(\frac{k}{\beta} - 1\right)\right)\right), & k < \beta, \\ 1, & \beta \le k \le (N-1)\left(1 - \frac{\alpha}{2}\right), \\ \frac{1}{2}\left(1 + \cos\left(\pi\left(\frac{k}{\beta} - \frac{2}{\alpha} + 1\right)\right)\right), & \text{otherwise,} \end{cases}$$

where $\beta = \frac{\alpha(N-1)}{2}$, for $k = 0, 1, \ldots, N-1$.

Example

```
> listplot(tukey_window([1$1000],0.4))
```



21.5.15 Welch window

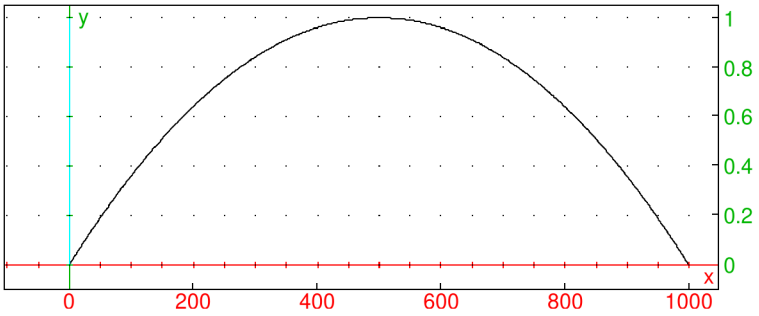
The `welch_window` command applies the Welch window function to a sequence or its segment.

- `welch_window` takes one mandatory argument and two optional arguments:
  - **v**, a real vector with length n.
  - Optionally, m, N, a sequence of two integers (by default, $m = 0$ and $N = n$).
- `welch_window` returns the elementwise product of $[v_m, v_{m+1}, \ldots, v_{m+N-1}]$ and the vector w of length N defined by

$$w_k = 1 - \left(\frac{k - \frac{N-1}{2}}{\frac{N-1}{2}} \right)^2, \quad k = 0, 1, \ldots, N-1.$$

Example

```
> listplot(welch_window([1$1000]))
```



22 Data analysis and machine learning

22.1 Data conversion routines

22.1.1 Converting between geodetic and ECEF coordinates

You can convert between geodetic (LLA: Latitude, Longitude, Altitude) and Earth-Fixed Earth-Centered (ECEF) coordinates with respect to the WGS84 ellipsoid by using the `geodetic2ecef` and `ecef2geodetic` commands.

For more information about geographic coordinate systems and conversion, see e.g. here.

- `geodetic2ecef` takes three arguments:
  - ϕ, the latitude (in decimal degrees).
  - λ, the longitude (in decimal degrees).
  - h, the height (i.e. altitude, in metres).

  Alternatively, you can pass the list `[`ϕ, λ, h`]` as a single argument.
- `geodetic2ecef(`ϕ, λ, h`)` returns the list $[x, y, z]$ in the ECEF Cartesian coordinate system (all components are in metres). For invalid locations, `undef` is returned.

The `ecef2geodetic` command works the opposite way. With ECEF coordinates x, y and z supplied as input arguments, you get the list $[\phi, \lambda, h]$ of the corresponding geodetic coordinates.

Examples

```
> xyz:=geodetic2ecef(45.1832745,15.7852367,89.541)
```

$$[4333403.44478, 1225023.67407, 4501791.18176]$$

```
> ecef2geodetic(xyz)
```

$$[45.1832745, 15.7852367, 89.5409998717]$$

22.2 Clustering

Clustering is a form of unsupervised machine learning. XCAS can perform hierarchical clustering and k-means clustering.

22.2.1 Hierarchical clustering

The `cluster` command is used for hierarchical agglomerative clustering of custom data by using a custom distance function.

- `cluster` takes one or more arguments:
  - *data*, a list of data points.

– Optionally, *opt*, a sequence of options which may contain the following:
  * k or `count=`k, where k is a positive integer specifying the number of clusters (unset by default).
  * `count_inf=`k_l, where k_l is a positive integer specifying the minimal number of clusters (unset by default).
  * `count_sup=`k_u, where k_u is a positive integer specifying the maximal number of clusters (by default, $k_u = 16$).
  * `type=`*linkage*, where *linkage* is a string specifying the linkage method. Avaliable methods are: `single` (the default), `complete`, `average`, `weighted`, and `ward`.
  * `distance=`*dist*, where *dist* is a distance function. By default, the squared Euclidean distance function `distance2` is used unless *data* is a list of strings, in which case Levenshtein distance is used. For example, the taxicab distance function could be defined by typing:

    ```
    > taxicab:=(p1,p2)->l1norm(p1-p2)
    ```

  * `index=`*ind*, where *ind* is a string or a list of strings specifying the index function(s) used for selecting the optimal number of clusters (unset by default). Available index functions are: `silhouette`, `mcclain-rao`, `dunn`, and `all`. *ind* may also take a boolean value, in which case no index is used (*ind*`=false`) or the silhouette index is used (*ind*`=true`).
  * `output=`*out*, where *out* is one of the following:
    · `part`, for outputting the partition of data into clusters.
    · `list`, for outputting the list of cluster indices for data points (the default).
    · `plot`, for outputting a colored visualization of data points with additional specifications given in `display` option (see below). Note that this is possible only with two- and 3D numerical data.
    · `count`, for outputting only the number of clusters.
    · `index`, for outputting the list of values of the (first) used index.
    · `tree`, for outputting a dendrogram drawing which visualizes the linking process.
  * `display=`*disp*, where *disp* is a configuration for plot output (by default, points are drawn as dots of width 2).
  * `color=`*colors*, where *colors* is a list of colors that will be used for cluster coloring in the plot output mode (by default, the standard palette of first 16 colors is used, with white and yellow replaced by more visible colors).
  * `labels=`*lab*, where *lab* is a boolean value which specifies whether to show data points in the dendrogram.

- `cluster(`*data*⟨, *opt*⟩`)` returns the result as specified by the `output` argument (see above).
- Hierarchical clustering is slower than the k-means algorithm (see Section 22.2.2, p. 619) but may produce a better classification of the data. Hierarchical clustering is also a method of choice for custom data types and distance functions.
- If more than one index is computed in a clustering process, then the optimal number of clusters is decided by voting: the number which was selected by most indices is used.

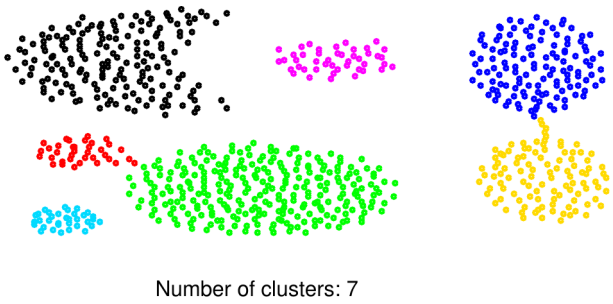
Examples

To apply `cluster` command to a 2D "aggregation" shape dataset which can be downloaded from here, first load the data by entering:

```
> data:=csv2gen("/home/luka/Downloads/Aggregation.txt","\t"):;
```

The original file contains three data columns: first are the x-coordinates, second the y-coordinates, and third are cluster indices. To cluster and plot 2D points specified by the first two columns, with the average linkage method and silhouette index (which is used by default if `index=true`), enter:

```
> cluster(delcols(data,2),type="average",index=true,output=plot)
```



Levenshtein distance (see Section 5.2.16, p. 63) is used by default for string data. For example:

```
> cluster(["cat","mouse","rat","spouse","house","cut"],output=part)
```

$$\begin{bmatrix} \text{“cat”} & \text{“rat”} & \text{“cut”} \\ \text{“mouse”} & \text{“spouse”} & \text{“house”} \end{bmatrix}$$

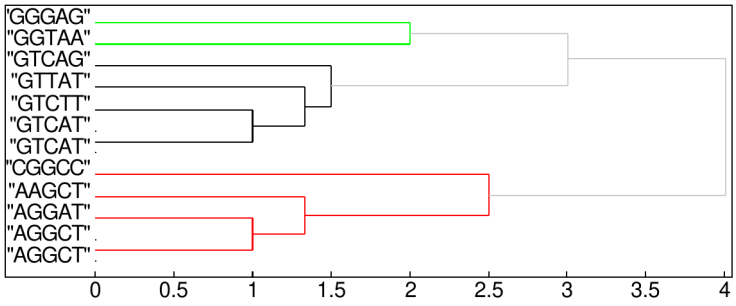
In the following example, genomic sequences are split into three clusters by using the `average` linkage and Hamming distance function.

```
> data:=["GTCTT","AAGCT","GGTAA","AGGCT","GTCAT","CGGCC",
  "GGGAG","GTTAT","GTCAT","AGGCT","GTCAG","AGGAT"]:;
  cluster(data,type="average",count=3,distance=hamdist,output=part)
```

$$\begin{bmatrix} \text{“GTCTT”}, \text{“GTCAT”}, \text{“GTTAT”}, \text{“GTCAT”}, \text{“GTCAG”} \\ \text{“AAGCT”}, \text{“AGGCT”}, \text{“CGGCC”}, \text{“AGGCT”}, \text{“AGGAT”} \\ \text{“GGTAA”}, \text{“GGGAG”} \end{bmatrix}$$

To display the dendrogram, enter:

```
> cluster(data,type="average",count=3,distance=hamdist,output=tree)
```



22.2.2 k-means clustering

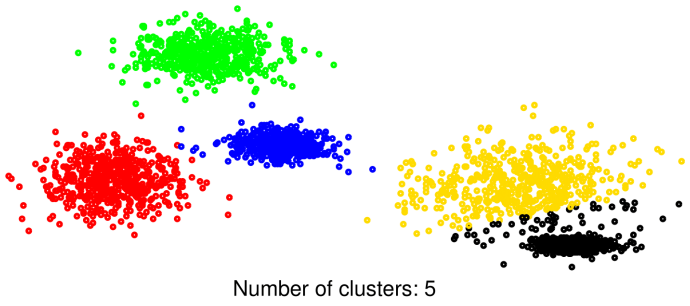
The `kmeans` command is an efficient implementation of k-means algorithm for clustering numerical multidimensional data.

- `kmeans` accepts one or more input arguments:
  - *data*, which must be a list of 2D points or m-dimensional Euclidean points entered as vectors. For n such points, *data* should be a $n \times m$ matrix.
  - Optionally, *opt*, a sequence of options which may contain the following:
    - k or `count=`k or `count=`k_l`..`k_u, where k is a positive integer specifying the number of clusters and k_l`..`k_u is an interval specifying minimal and maximal number of clusters (by default, unset).
    - `count_inf=`k_l, where k_l is a positive integer specifying the minimal number of clusters (by default, $k_l = 2$).
    - `count_sup=`k_u, where k_u is a positive integer specifying the maximal number of clusters (by default, $k_u = n - 1$).
    - `limit=`N, where N is a positive integer specifying the maximal number of iterations for the k-means algorithm (by default, $N = 100$).
    - `index=`*str*, where *str* is a string or a list of strings specifying the index functions used to find the optimal number of clusters automatically. Available indices are: `ball-hall`, `banfeld-raftery`, `calinski-harabasz`, `davies-bouldin`, `det`, `ksq-detW`, `log-det`, `log-ss`, `pbm`, `ratkowsky-lance`, `ray-turi`, `scott-symons`, `trace-W`, `trace-WiB` and `all` (note that you must surround these names by double-quotes). For details, see Desgraupes (2017) (PDF). *str* may also take a boolean value, in which case either no index is used (*str*`=false`) or the Calinski-Harabasz index is used (*str*`=true`). By default, Hartigan's criterion is used to determine the optimal number of clusters if k is not given.
    - `output=`*out*, where *out* is one of the following:
      - `part`, for outputting the partition of input data into clusters.
      - `list`, for outputting the list of cluster indices for data points (the default).
      - `plot`, for outputting a colored visualization of data points with additional specifications given in `display` option (see below). Note that this is possible only with two- and 3D data.
      - `count`, for outputting only the number of clusters.
      - `index`, for outputting the list of values of the (first) used index.
    - `display=`*disp*, where *disp* is a configuration for plot output (by default, points are drawn as dots of width 2).
    - `color=`*colors*, where *colors* is a list of colors that will be used for cluster coloring in the plot output mode (by default, the standard palette of first 16 colors is used, with white and yellow replaced for visibility).
- `kmeans(`*data* ⟨, *opt*⟩`)` returns the result as specified by the `output` argument (see above).
- `kmeans` applies the Hartigan-Wong implementation of k-means algorithm, which is fast and suitable for large datasets.
- `kmeans` does not support custom distance functions. If you want to cluster data by using a distance function other than the Euclidean distance, use the `cluster` command (see Section 22.2.1, p. 617).
- If more than one index is computed in a clustering process, then the optimal number of clusters is decided by voting: the number which was selected by most indices is used.

Example

We cluster some random gaussian data and use the Calinski-Harabasz index for choosing the optimal number of clusters.

```
> d:=5:; n:=2500:; data:=[]:;
  C:=[[5,6],[14,10],[27,5],[10,21],[30,-2]]:;
  R:=[[2,2.5],[1.2,1],[3,3],[2,1.8],[1,0.5]]:;
  for k from 1 to n do
    j:=rand(d);
    x:=sample(randvar(normal,C[j][0],R[j][0]));
    y:=sample(randvar(normal,C[j][1],R[j][1]));
    data:=append(data,[x,y]);
  od:;
  kmeans(data,index="calinski-harabasz",output=plot)
```



22.3 Artificial neural networks

This section explains how to create and train feed-forward neural networks in Xcas.

A feed-forward network, more precisely a *multilayer perceptron*, is comprised of a sequence of at least three neuron layers. The first layer is called the *input layer* and the last layer is called the *output layer*. Layers are essentially vectors of numerical data. Each component of such vector corresponds to an activated neuron. The network processes the given input by applying the formula $l_{k+1} = f_k(W_k^T l_k)$, where l_k is the kth layer of length n_k, l_{k+1} is the $(k+1)$-th layer of length n_{k+1}, W_k is a $n_k \times n_{k+1}$ weight matrix and f_k is the activation function at the $k+1$-th layer (which is applied element-wise). Here $k = 1, 2, \ldots, L-1$, where L is the number of layers. Each layer except the final one contains a *bias neuron* with linear activation. Bias neurons are auxiliary neurons created during construction of the network.

Multilayer perceptrons are trained by using *backpropagation* to compute the approximate gradient (with respect to the weight parameters in the matrices W_k) of the error function which compares the output to the expected output (or ground truth). Precisely, after training data is fed to the network, the obtained output is compared to the expected output and finally the error is backpropagated to find the weight adjustment enabling the network to predict the expected output more accurately.

The examples in Section 22.3.2, p. 625 demonstrate how to prepare training data, construct neural networks, train them and test their accuracy.

Remark. The commands in this section require that `giac` is linked to GSL.

22.3.1 Creating neural networks

The command `neural_network` is used for creating trainable feed-forward neural networks.

- `neural_network` takes one mandatory argument and several optional arguments:

– *topology* or *net*, where *topology* is a vector of positive integers defining the number of neurons in each layer and *net* is an existing neural network.
  * If *topology* is given, then a neural network is constructed from scratch. The first number in *topology* is the size of the input layer and the last number is the size of the output layer. The numbers in between are sizes of the hidden layers. If *topology* contains only two numbers m and n, then a hidden layer of size $2(m+n)/3$ is added automatically. The number of layers will be referred to as L below.
  * If *net* is given, then a copy of *net* is constructed with optional modifications of the parameters as specified by the additional arguments (see below).
– Optionally, `block_size=`*bs*, where *bs* is a positive integer specifying the amount of samples that can be processed by the network at a time. This value can be set to the batch size or some smaller number for faster execution (by default, $bs = 1$).
– Optionally, `learning_rate=`*rate*, where *rate* is a positive real number which is used as the learning rate of the network (by default, $rate = 0.001$). Alternatively, *rate* can be a two-element vector $[rinit, schedule]$ where $rinit = \eta_0$ is the initial rate (a real number) and *schedule* is a function $\eta(t)$, $t \in \mathbb{Z}$ which is used as the schedule multiplier for *rinit* (t is initially zero and increases by 1 after each weight update). In this case, the learning rate in iteration t is $\eta_t = \eta_0\eta(t)$.
– Optionally, `momentum=`*mom*, where *mom* is a number $\mu \in [0, 1)$ which specifies the momentum parameter of the network (it is usually set between 0.5 and 1.0). Alternatively, *mom* can be specified as a two-element vector $[\beta_1, \beta_2]$ in which case Adam (adaptive moment estimation) method is used. To use Adam with the default (paper) parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, simply set *mom* to `adaptive`. (By default, $\mu = 0$, i.e. no momentum is used.)
– Optionally, `func=`*act*, where *act* is the activation function used in the hidden layers. It can be specified either as an univariate function (e.g. `tanh` or `logistic`) or a list in which the first element is a univariate function $f(x, a, b, \dots)$ with parameters $a, b, \dots$, and the other elements are the fixed values $a_0, b_0, \dots$ of the parameters. In the latter case the value of f at x is obtained by computing $f(x, a_0, b_0, \dots)$. ReLU activation is specified by the symbol `ReLU`. By default, ReLU activation is used for regression models while tanh is used for classifiers (see below).
– Optionally, `output=`*outact*, where *outact* is the activation function used in the output layer. It is specified in the same way as the hidden activation function (see above). By default, identity is used in regression models, while sigmoid and softmax are used in binary and multi-class classifiers, repectively.
– Optionally, `erf=`*err*, where *err* is the error function used for network training. It is either a bivariate function $g(\mathbf{t}, \mathbf{y})$ or a list in which the first element is a function $g(\mathbf{t}, \mathbf{y}, a, b, \dots))$ with parameters $a, b, \dots$ and the other elements are the fixed values $a_0, b_0, \dots$ of the parameters. In the latter case the error is computed as $f(\mathbf{t}, \mathbf{y}, a_0, b_0, \dots)$. Here $\mathbf{t}$ is the expected output and $\mathbf{y}$ is the output read at the final layer of the network after forward-propagation of the input. By default, half mean squared distance is used in regression models while log-loss and cross-entropy are used in binary and multi-class classifiers, respectively. These error functions can be specified by using shorthands `MSE`, `log_loss` and `cross_entropy`. The corresponding definitions are:

$$\text{MSE:} \quad f(\mathbf{t}, \mathbf{y}) = \frac{1}{2n}\sum_{i=1}^{n}(t_i - y_i)^2,$$

$$\text{log-loss:} \quad f(\mathbf{t}, \mathbf{y}) = -\sum_{i=1}^{n}(t_i \log(y_i) + (1 - t_i)\log(1 - y_i)),$$

$$\text{cross-entropy:} \quad f(\mathbf{t}, \mathbf{y}) = -\sum_{i=1}^{n} t_i \log(y_i).$$

– Optionally, `labels=`*lab* or `classes=`*lab*, where *lab* is the list of labels corresponding to the output neurons. If this option is set, then the activation for hidden layers is set to `tanh`, while the activation for the output layer is set to `logistic` except with `classes` and multiple labels, in which case the softmax activation is used. (By default, *lab* is unset, i.e. the network is a regression model.) With `labels` the network becomes a multi-label classifier (multiple labels may be assigned to the output), while with `classes` it becomes a binary/multi-class classifier (exactly one label is assigned to the output).

– Optionally, `weights=`*wi*, where *wi* is either a list of matrices containing initial weights for the model or a random variable used for generating initial weights automatically.

∗ In the former case, *wi* is a list $[W^{(1)}, W^{(2)}, \ldots, W^{(L-1)}]$ where $W^{(k)}$ is the matrix specifying the weights between the kth and $(k+1)$-th layer. The element $w_{ij}^{(k)}$ is the weight corresponding to the link from the ith neuron in kth layer to the jth neuron in $(k+1)$-th layer. Optionally, the initial bias for the kth layer may be specified as an additional row in the matrix $W^{(k)}$.

∗ In the latter case, *wi* is either a constant or a random variable X as returned by the command `randvar` which may optionally contain one or two symbolic parameters: n_{in} and optionally n_{out}, which correspond to the size of the preceding layer ("fan-in") and the size of the next layer ("fan-out"). If symbols are present, then *wi* must be a list in which the first element is the random variable and other elements are the symbols (n_{in} first, then optionally n_{out}). These symbols are substituted by sizes of the kth and $(k+1)$-th layer, respectively. The commonly used initializations by He, Glorot and LeCun can be specified as strings `"he"`, `"glorot"` and `"lecun"` with the suffix `"-uniform"` resp. `"-normal"` for uniform resp. Gaussian variants.

By default, the uniform random variable $U\left(-\frac{1}{\sqrt{n_{\text{in}}}}, \frac{1}{\sqrt{n_{\text{in}}}}\right)$ is used for weights initialization. Note that bias weights are always initialized to zero.

– Optionally, `weight_decay=`*wd*, where *wd* is a positive real number α or a list of $L-1$ such numbers $[\alpha_1, \alpha_2, \ldots, \alpha_{L-1}]$. These numbers are the L^2-regularization coefficients used in the model (by default, they are all equal to zero, i.e. no regularization is performed). If a single number α is given, then the coefficients for all layers are set to that value. If the list is given, then α_k is used for the kth layer (α_k is a weight decay coefficient for the weights in $W^{(k)}$). Note that regularization is not applied to bias weights.

– Optionally, `title=`*str*, where *str* is a string holding the name of the newtork which is printed alongside its one-line description (by default, network is not named).

- `neural_network(`*topology*⟨, *options*⟩`)` or `neural_network(`*net*⟨, *options*⟩`)` returns the network object `net` which can be trained by using the `train` command (see Section 22.3.2, p. 625).

- After the network is trained to a sufficient accuracy, you can feed it with some input `inp` by calling the command `net(inp)`, which returns the list of values in the final layer or output label(s) if the network is a classifier. `inp` can also be a matrix, in which case it is processed row by row and the list of results is returned. Alternatively, you can pass two arguments to `net` as in the command `net(inp,res)` where `res` is (the list of) expected output(s). The return value in this case is the (list of) error(s) made by the network in attempt to predict `res`.

- Hyperparameters and other properties of the network `net` can be fetched by using the command `net[`*property*`]`, where *property* is one of:

- `block_size`, for obtaining the block size,
- `learning_rate`, for obtaining the learning rate and possibly the schedule multiplier,
- `labels`, for obtaining the list of output labels,
- `momentum`, for obtaining the momentum/Adam parameter(s),
- `title`, for obtaining the network name,
- `topology`, for obtaining the network topology, i.e. the list of layer sizes,
- `weight_decay`, for obtaining the list of L^2-regularization parameters,
- `weights`, for obtaining the list of weight matrices (bias weights are contained in the last row in each of these matrices).

To fetch the contents of the kth layer neurons, the command `net[k]` can be used, where $k \in \{0, 1, \ldots, L-1\}$. This returns the layer as a matrix with the number of rows equal to the block size in which the ith row corresponds to the ith sample passed forward through the network. This is useful for e.g. obtaining hidden representations from autoencoders.

- Although `neural_network` is flexible when it comes to custom activation and error functions, the resulting network is optimized for speed only when the options `func`, `output` and `erf` are unset, i.e. only when the default activation/error function(s) are used.
- Networks can be saved to disk by using the command `write` (see Section 4.1.1, p. 44) and loaded by using the command `read` (see Section 4.1.3, p. 45). This is useful for storing networks after training and loading them on demand.

Examples

To create a network with three layers of size 2, 3, and 1, input:

```
> neural_network([2,3,1])
```

a neural network with input of size 2 and output of size 1

To use GELU activation $x \mapsto x \cdot \Phi(x)$ for hidden neurons:

```
> neural_network([2,3,1],func=unapply(x*normal_cdf(x),x))
```

a neural network with input of size 2 and output of size 1

To define a penguin classifier:

```
> net:=neural_network([10,15,7,3],classes=["adelie","chinstrap","gentoo"])
```

a classifier with input of size 10 and 3 classes

Now we create a copy of `net` with block size changed to 10:

```
> netcopy:=neural_network(net,block_size=10)
```

a classifier with input of size 10 and 3 classes

```
> netcopy[block_size]
```

10

22.3.2 Training a neural network

The command `train` is used for training neural networks created by the command `neural_network` (see Section 22.3.1, p. 621).

- `train` takes three mandatory arguments and one optional argument:
  - *net*, a neural network.
  - *input*, an input vector or a list of input vectors.
  - *expout*, the expected output(s) from *input*.
  - Optionally, *batchsize*, a positive integer specifying the batch size. Weight updates are applied only after processing each batch of samples. By setting this value to a negative number, weight deltas are accumulated but the weights are not updated. (By default, *batchsize* is equal to the number of input vectors, i.e. weights will be updated after processing the whole input.)
- `train` returns a copy of the input network containing weight modifications. If the weights are not modified (depending on the *batchsize*), the returned newtork contains the accumulated weight deltas which will be applied alongside any subsequent modification in the next update. Before calling `train` again, the resulting so-far trained network can be tested for accuracy.
- `train` applies batch/minibatch/stochastic gradient descent (depending on the batch size) in attempt to optimize the weight parameters. Optionally, it takes advantage of the weight decay, classical momentum or Adam (adaptive moment estimation) methods which usually make the learning process faster and the resulting model more accurate. These tools are enabled for the network during the construction (see Section 22.3.1, p. 621).
- It is advisable to set the `block_size` option to the batch size when constructing the network (see Section 22.3.1, p. 621). This makes the forward passing and backpropagation more efficient thanks to the fast BLAS level 3 routines.

Examples

Predicting values of a nonlinear function. To demonstrate learning a nonlinear function, let

$$f(x_1, x_2) = x_1 \sin(3x_2) \cos(2x_1 x_2).$$

We create a neural network with three hidden layers and train it to predict the value $f(x_1, x_2)$ given the input vector $\mathbf{x} = (x_1, x_2)$ in the square $S = [-1, 1]^2$. We use Adam with the default parameters and set the weight decay factor to 10^{-4}. The block size is set to 100 (we will use full-batch gradient descent, meaning that the network will process training samples in bulks of 100 samples at once). The network has $30 + 20 + 10$ neurons in hidden layers (not counting the bias neurons). We store the network topology in `t`.

```
> t:=[2,30,20,10,1]:;
  net:=neural_network(t,momentum=adaptive,weight_decay=1e-4,block_size=100)
```

a neural network with input of size 2 and output of size 1

Next we create 5000 training samples in S by using the uniform random variable $U(-1, 1)$.

```
> f(x):=x[0]*sin(3*x[1])*cos(2*x[0]*x[1]):;
```

```
> data:=ranm(5000,2,uniformd(-1,1)):; res:=apply(f,data):;
```

Now we have data points in `data` and the corresponding function values in `res`. In a similar manner, we create a collection of another 100 samples, which will be kept unseen by the network and used for testing its accuracy.

```
> test_data:=ranm(100,2,uniformd(-1,1)):; test_res:=apply(f,test_data):;
```

Next we train the network using 2500 epochs. We test the accuracy in intervals of 250 epochs.

```
> for epoch from 1 to 2500 do
    net:=train(net,data,res);
    if irem(epoch,250)==0 then print(mean(net(test_data,test_res))); fi;
  od:;
```

```
0.00211048030912
0.000199757341385
8.70954607301e-05
6.21486919568e-05
5.22746108944e-05
5.0011469063e-05
4.91138941048e-05
4.81631000381e-05
4.86611973063e-05
4.79773288935e-05
```

Evaluation time: 16.85

Note that half-MSE is used as the error function by default (this is a regression model). Now we generate a random point $\mathbf{x}_0$ in S and compute the predicted and exact value of $f(\mathbf{x}_0)$.

```
> x0:=ranv(2,uniformd(-1,1))
```

$$[-0.402978600934, -0.836934269406]$$

```
> net(x0),f(x0)
```

$$0.18592080555, 0.185619807512$$

To plot the learned surface, use the command:

```
> plot3d(quote(net([x1,x2])),x1=-1..1,x2=-1..1)
```

Nonlinear separation of data. Let

$$f(t) = \frac{2}{5} + \frac{3}{2}\left(t - \frac{1}{2}\right)^2,$$

which defines a parabola that splits the unit square $S = [0,1]^2$ into two regions. We generate 1024 random points in S and label them either as `below` or `above`, depending on whether they are located below or above the parabola.

```
> f(t):=0.4+1.5*(t-0.5)^2:;
  g:=unapply(x[1]<f(x[0])?"below":"above",x):;
  pts:=ranm(1024,2,uniformd(0,1)):;
  lab:=apply(g,pts):;
```

Next we create a neural network with four hidden layers which we train to label random points in S. The error function used by default is the log-loss function since we have a binary classifier.

```
> params:=seq[momentum=adaptive,weight_decay=1e-4,block_size=128]:;
  net:=neural_network([2,10$4,1],classes=["below","above"],params)
```

a classifier with input of size 2 and 2 classes

We train on the generated data with batch size 128 and 500 epochs. Training data is shuffled before each epoch in order to avoid symmetry.

Figure 22.1: Handwritten digits from the MNIST dataset

```
> for epoch from 1 to 500 do
    p:=randperm(size(pts));
    net:=train(net,sortperm(pts,p),sortperm(lab,p),128);
  od:;
```

Now we test the accuracy of the classifier by using 1000 random test samples which we store in `tst`. The number of misses is the Hamming distance of the vector of predicted labels `net(tst)` from the vector of correct labels which we obtain by using the command `apply(g,tst)`.

```
> tst:=ranm(1000,2,uniformd(0,1)):;
  (1-hamdist(net(tst),apply(g,tst))/size(tst))*100.0
```

99.8

Recognizing handwritten digits. Here we train a neural network on the MNIST dataset in PNG format, which can be obtained here. This dataset contains 60000 training grayscale images of handwritten digits in 28×28 resolution, anlogside 10000 testing images (see Figure 22.1). Let us assume that the contents of `mnist_png.tar.gz` are unpacked in the `Downloads` folder. Now put the files `mnist_training.csv` and `mnist_testing.csv`, which can be obtained here, into the subfolders `training` and `testing`, respectively. These CSV files contain image paths and labels. We use these files to load and encode training and testing data in Xcas.

First we load the training data, which takes several minutes. Note that we flatten and normalize the images, so that the training vectors contain numbers strictly in $[0, 1]$ (see Section 28.1, p. 812).

```
> train_path:="/home/luka/Downloads/mnist_png/training/":;
  train_csv:=csv2gen(train_path+"mnist_training.csv",","):;
  train_data:=[0$size(train_csv)]:;
  train_lab:=col(train_csv,1):;
  for k from 1 to size(train_csv) do
    train_data[k-1]=<flatten(image(train_path+train_csv[k-1,0]))/255.0;
  od:;
```

We load the testing images in a similar manner and store them in `test_data` and `test_lab`.

Now we create a neural network with three hidden layers for classification of handwritten digits which uses ReLU activation in hidden layers and He normal initialization for weights.

```
> c:=["zero","one","two","three","four","five","six","seven","eight","nine"]:;
  params:=seq[func=ReLU,weights="he-normal",momentum=adaptive,weight_decay=1e-4]:;
  net:=neural_network([28*28,500$3,10],block_size=100,classes=c,params);
```

a classifier with input of size 784 and 10 classes

We train the network with batch size 100 and 5 epochs. Training data is shuffled before each epoch and the mean error on testing data is printed after each epoch. The training takes about a minute.

```
> for epoch from 1 to 5 do
    p:=randperm(size(train_data));
    net:=train(net,sortperm(train_data,p),sortperm(train_lab,p),100);
    print(mean(net(test_data,test_lab)));
  od:;
```

```
0.110725006182
0.104843072908
0.0859572165559
0.0675629083633
0.0626279369745
```

The printed error values are computed using the cross-entropy function, which is used by default in multiclass classifiers. To test the accuracy of the network, use the following command.

```
> (1-hamdist(net(test_data),test_lab)/size(test_data))*100.0
```

```
97.88
```

23 Numerical computations

Real numbers may have an exact representation (e.g. rationals, symbolic expressions involving square roots or constants like π, ...) or approximate representation, which means that internally the real is represented by a rational (with a denominator that is a power of the basis of the representation) close to the real. Inside Xcas, the standard scientific notation is used for approximate representation; that is a *mantissa* (with a point as decimal separator) optionally followed by the letter `e` and an integer *exponent*.

Note that, for example, the real number 10^{-4} is an exact number but 1e–4 is an approximate representation of this number.

23.1 Floating point representation

This section discusses how real numbers are represented.

23.1.1 Digits

The `Digits` variable (see Section 2.5.1, p. 13) is used to control how real numbers are represented and also how they are displayed. When the specified number of digits is less or equal to 14 (for example `Digits:=14`), then hardware floating point numbers are used and they are displayed using the specified number of digits. When `Digits` is larger than 14, Xcas uses the MPFR library, the representation is similar to hardware floats (cf. infra) but the number of bits of the mantissa is not fixed and the range of exponents is much larger. More precisely, the number of bits of the mantissa of a created MPFR float is `ceil(Digits*log(10)/log(2))`.

Note that if you change the value of `Digits`, this will affect the creation of new real numbers compiled from command lines or programs or by instructions like `approx`, but it will not affect existing real numbers. Hence hardware floats may coexist with MPFR floats, and even in MPFR floats, some may have 100 bits of mantissa and some may have 150 bits of mantissa. If operations mix different kinds of floats, the most precise kind of floats are coerced to the less precise kind of floats.

23.1.2 Representation by hardware floats

A real is represented by a floating number d, that is

$$d = 2^{\alpha}\,(1+m), \quad 0 < m < 1, \quad -2^{10} < \alpha < 2^{10}.$$

If $\alpha > 1 - 2^{10}$, then $m \geq 1/2$, and d is a normalized floating point number, otherwise d is denormalized ($\alpha = 1 - 2^{10}$). The special exponent 2^{10} is used to represent plus or minus infinity and NaN (Not a Number). A hardware float is made of 64 bits:

- the first bit is for the sign of d (0 for + and 1 for −),
- the 11 following bits represents the exponent, more precisely if α denotes the integer given by the 11 bits, the exponent is $\alpha + 2^{10} - 1$,
- the 52 last bits codes the mantissa m, more precisely if M denotes the integer given by the 52 bits, then $m = 1/2 + M/2^{53}$ for normalized floats and $m = M/2^{53}$ for denormalized floats.

Examples of representations of the exponent:

- $\alpha = 0$ is coded by 011 1111 1111
- $\alpha = 1$ is coded by 100 0000 0000
- $\alpha = 4$ is coded by 100 0000 0011
- $\alpha = 5$ is coded by 100 0000 0100
- $\alpha = -1$ is coded by 011 1111 1110
- $\alpha = -4$ is coded by 011 1111 1011
- $\alpha = -5$ is coded by 011 1111 1010
- $\alpha = 2^{10}$ is coded by 111 1111 1111
- $\alpha = 2^{-10} - 1$ is coded by 000 0000 000

Remark. $2^{-52} = 0.2220446049250313\text{e}{-15}$.

Examples of representations of normalized floats

Representation of 3.1. We have

$$\begin{aligned} 3.1 &= 2 \cdot \left(1 + \frac{1}{2} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^9} + \frac{1}{2^{10}} + \cdots\right) \\ &= 2 \cdot \left(1 + \frac{1}{2} + \sum_{k=1}^{\infty} \left(\frac{1}{2^{4k+1}} + \frac{1}{2^{4k+2}}\right)\right), \end{aligned}$$

hence $\alpha = 1$ and $m = \frac{1}{2} + \sum_{k=1}^{\infty}(\frac{1}{2^{4k+1}} + \frac{1}{2^{4k+2}})$. Hence the hexadecimal and binary representation of 3.1 is:

```
40 (01000000),  8 (00001000), cc (11001100), cc (11001100),
cc (11001100), cc (11001100), cc (11001100), cd (11001101),
```

the last octet is 1101, the last bit is 1, because the following digit is 1 (upper rounding).

Representation of 3.0. We have $3 = 2 \cdot (1 + 1/2)$. Hence the hexadecimal and binary representation of 3 is:

```
40 (01000000), 8 (00001000), 0 (00000000), 0 (00000000),
0  (00000000), 0 (00000000), 0 (00000000), 0 (00000000)
```

The difference between representations of $3.1 - 3.0$ **and** 0.1**.** For the representation of 0.1:

$$\begin{aligned} 0.1 &= 2^{-4} \cdot \left(1 + \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \cdots\right) \\ &= 2^{-4} \cdot \sum_{k=0}^{\infty} \left(\frac{1}{2^{4k}} + \frac{1}{2^{4k+1}}\right), \end{aligned}$$

hence $\alpha = 1$ and

$$m = \frac{1}{2} + \sum_{k=1}^{\infty} \left(\frac{1}{2^{4k}} + \frac{1}{2^{4k+1}}\right),$$

therefore the representation of 0.1 is

```
3f (00111111), b9 (10111001), 99 (10011001), 99 (10011001),
99 (10011001), 99 (10011001), 99 (10011001), 9a (10011010),
```

the last octet is 1010, indeed the 2 last bits 01 became 10 because the following digit is 1 (upper rounding).

For the representation of $a = 3.1 - 3$: computing a is done by adjusting exponents (here nothing to do), then subtracting the mantissa and adjusting the exponent of the result to have a normalized float. The exponent is $\alpha = -4$ (that corresponds at $2 \cdot 2^{-5}$) and the bits corresponding to the mantissa begin at $1/2 = 2 \cdot 2^{-6}$: the bits of the mantissa are shifted to the left 5 positions and you get:

```
3f (00111111), b9 (10111001), 99 (10011001), 99 (10011001),
99 (10011001), 99 (10011001), 99 (10011001), a0 (10100000),
```

Therefore, $a > 0.1$ and $a - 0.1 = 1/2^{50} + 1/2^{51}$ (since $100000 - 11010 = 110$).

This is the reason why:

```
> floor(1/(3.1-3))
```

returns `9` and not `10` when `Digits:=14`.

23.1.3 Approximate evaluation

The `evalf` or `approx` command evaluates a symbolic expression to a numeric approximation, if possible (see Section 7.3.1, p. 128). The approximation is to `Digits` digits (see Section 2.5.1, p. 13), this can be changed with an optional second argument.

Examples

Assuming that in the CAS configuration Digits = 7, so hardware floats are used, and 7 digits are displayed:

```
> evalf(sqrt(2))
```

1.414214

You can change the number of digits in a command line by assigning the variable `DIGITS` or `Digits`.

```
> DIGITS:=20:;
  evalf(sqrt(2))
```

1.4142135623730950488

```
> evalf(10^-5)
```

1e−05

```
> evalf(10^15)
```

1e+15

```
> evalf(sqrt(2))*10^-5
```

1.41421356237e−05

23.2 Computing derivatives and integrals

23.2.1 Approximating derivatives

The `nDeriv` command numerically approximates the value of a derivative.

- `nDeriv` takes as arguments:
  - *expr*, expression.
  - Optionally, x, the variable used in the expression (by default `x`).
  - Optionally, h (by default 0.001).
- `nDeriv(`$ex\langle, x, h\rangle$`)` returns an approximated value of the derivative of the expression ex at the point x using the formula:
$$\frac{f(x+h)-f(x-h)}{2h}$$

For a more general command on numeric differentiation, see Section 13.2.4, p. 281.

Examples

```
> nDeriv(x^2,x)
```
$$\left((x+0.001)^2-(x-0.001)^2\right)\cdot 500.0$$

```
> subst(nDeriv(x^2,x),x=1)
```
$$2.0$$

```
> nDeriv(exp(x^2),x,0.00001)
```
$$\left(\mathrm{e}^{\left(x+1.0\times10^{-5}\right)^2}-\mathrm{e}^{\left(x-1.0\times10^{-5}\right)^2}\right)\cdot 50000.0$$

```
> subst(exp(nDeriv(x^2,x,0.00001)),x=1)
```
$$7.38905609706$$

which is an approximate value of $\mathrm{e}^2 \approx 7.38905609893$.

23.2.2 Approximating definite integrals

The `romberg` or `nInt` command finds approximate values of integrals using the Romberg method.

- `romberg` takes three mandatory arguments and one optional argument:
  - *expr*, an expression involving one variable.
  - Optionally, *var*, the variable (by default `x`).
  - a, b, two real numbers.
- `romberg(`$expr\langle, var,\rangle a, b$`)` returns an approximated value of the integral $\int_a^b expr\cdot \mathrm{d}var$. The integrand must be sufficiently regular for the approximation to be accurate, otherwise, `romberg` returns a list of real values that come from the application of the Romberg algorithm (the first list element is the trapezoid rule approximation, the next ones come from the application of the Euler-Maclaurin formula to remove successive even powers of the step of the trapezoid rule).

The `gaussquad` command finds an approximate value of an integral, calculated by an adaptive method by Ernst Hairer which uses a 15-point Gaussian quadrature.

- `gaussquad` takes four arguments:
  - *expr*, an expression.
  - *var*, the variable used by the expression.
  - a, b, two numbers.
- `gaussquad(`$expr, a, b$`)` returns an approximation of the integral $\int_a^b expr \cdot \mathrm{d}var$.

Example

```
> romberg(exp(x^2),x,0,1)
```

1.46265174591

```
> gaussquad(exp(x^2),x,0,1)
```

1.46265174591

```
> gaussquad(exp(-x^2),x,-1,1)
```

1.49364826562

23.3 Solving equations

23.3.1 Newton method

The `newton` command uses Newton's method to approximate a solution to an equation.

- `newton` takes one mandatory argument and four optional arguments:
  - *ex*, an expression.
  - Optionally, *var*, the variable used in this expression (by default `x`).
  - Optionally, a, a number (by default 0)
  - Optionally, ε, a small number (by default `1e-8`)
  - Optionally, *nbiter* (by default 12).
- `newton(`$ex\langle, var, a, \varepsilon, nbiter\rangle$`)` returns an approximate solution of the equation $ex = 0$ using the Newton algorithm with starting point $var = a$. The maximum number of iterations is *nbiter* and the precision is ε.

Examples

```
> newton(x^2-2,x,1)
```

1.41421356237

```
> newton(x^2-2,x,-1)
```

−1.41421356237

```
> newton(cos(x)-x,x,0)
```

0.739085133215

23.3.2 Finding approximate solutions of equations involving one variable

The `fsolve` or `nSolve` command can solve equations or systems of equations. This section will discuss solving equations; systems will be discussed in the next section.

The `cfsolve` command is the complex version of `fsolve`, with the same arguments. The only difference is that `cfsolve` gives numeric solutions over the complex numbers, even if Xcas is not in complex mode (see Section 2.5.5, p. 14). `fsolve` will return complex roots, but only in complex mode.

`fsolve` solves numeric equations of the form:

$$f(x) = 0, \quad x \in (a, b).$$

Unlike `solve` (Section 9.3.6, p. 184) or `proot` (Section 23.3.4, p. 637), it is not limited to polynomial equations.

- `fsolve` takes one mandatory argument and three optional arguments:
  - *eqn*, an equation involving one variable.
  - Optionally, *var*, the variable (by default `x`).
  - Optionally, *init*, an initial approximation or range.
  - Optionally, *method*, the name of an iterative method to be used by the GSL solver. The possible values are:

    `bisection_solver`. This algorithm of dichotomy is the simplest but also generally the slowest. It encloses the zero of a function on an interval. Each iteration cuts the interval into two parts, the middle point value is calculated. The function sign at this point gives you the half-interval on which the next iteration will be performed.

    `brent_solver`. The Brent method interpolates of f at three points, finds the intersection of the interpolation with the x axis, computes the sign of f at this point and chooses the interval where the sign changes. It is generally faster than bisection.

    `falsepos_solver`. The "false position" algorithm is an iterative algorithm based on linear interpolation: it computes the value of f at the intersection of the line $(a, f(a))$, $(b, f(b))$ with the x axis. This value gives us the part of the interval containing the root, and on which a new iteration is performed. The convergence is linear but generally faster than bisection.

    `newton_solver`. This is the classic Newton method. The algorithm starts at an initial value x_0, then finds the intersection x_1 of the tangent at x_0 to the graph of f, with the x axis, the next iteration is done with x_1 instead of x_0. The x_i sequence is defined by

    $$x_0 = x_0, \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

    If the Newton method converges, then the convergence is quadratic for roots of multiplicity 1.

    `secant_solver`. The secant method is a simplified version of Newton's method. The computation of x_1 is done using Newton's method, but then the computation of $f'(x_n), n > 1$ is done approximately. This method is used when the computation of the derivative is expensive:

    $$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}}, \quad f'_{est} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

    The convergence for roots of multiplicity 1 is of order $(1 + \sqrt{5})/2 \approx 1.62\ldots$.

`steffenson_solver`. The Steffenson method is generally the fastest method. It combines Newton's method with a "delta-two" Aitken acceleration: with Newton's method, you get the sequence x_i and the convergence acceleration gives the Steffenson sequence

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{(x_{i+2} - 2x_{i+1} + x_i)}.$$

- `fsolve(`*eqn* ⟨, *var*, *init*, *method*⟩`)` returns an approximate solution to the equation *eqn*.

Examples

Input in real mode:

```
> fsolve(x^3-1,x)
```

1.0

Input in complex mode:

```
> fsolve(x^3-1,x)
```

$[-0.5 - 0.866025403784\mathrm{i}, -0.5 + 0.866025403784\mathrm{i}, 1.0]$

Input in any mode:

```
> cfsolve(x^3-1,x)
```

$[-0.5 - 0.866025403784\mathrm{i}, -0.5 + 0.866025403784\mathrm{i}, 1.0]$

```
> fsolve(sin(x)=2)
```

[]

```
> cfsolve(sin(x)=2)
```

$[1.57079632679 - 1.31695789692\mathrm{i}, 1.57079632679 + 1.31695789692\mathrm{i}]$

```
> fsolve((cos(x))=x,x,-1..1,brent_solver)
```

[0.739085133215]

23.3.3 Finding approximate solutions to systems of equations

In Section 23.3.2, p. 634 it was shown how to use the `fsolve` and `cfsolve` commands to solve equations. This section will discuss systems of equations.

As before, the `cfsolve` command is the complex version of `fsolve`, with the same arguments. The only difference is that `cfsolve` gives numeric solutions over the complex numbers, even if XCAS is not in complex mode (see Section 2.5.5, p. 14). `fsolve` will return complex roots, but only in complex mode.

- For solving systems of equations, `fsolve` takes three mandatory arguments and one optional argument:
  - *eqns*, a list of equations (or expressions, considered to be equal to zero) to solve.
  - `vars`, a list of the variables.
  - Optionally, `init`, a list initial values for the variables.

– Optionally, *method*, the method to use. The possible methods are: `dnewton_solver`, `hybrid_solver`, `hybrids_solver`, `newtonj_solver`, `hybridj_solver`, and `hybridsj_solver`.

- `fsolve(`*eqns*, *vars* ⟨, *init*, *method*⟩`)` returns an approximate solution to *eqns*.
- The methods are inherited from the GSL. The methods whose names end with `j_solver` use the jacobian matrix, the rest use approximations for the derivatives. All methods use an iteration of Newton kind

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The four hybrid methods also apply the method of gradient descent when the Newton iteration would make a too large of a step. The length of the step is computed without scaling for `hybrid_solver` and `hybridj_solver` or with scaling (computed from $f'(x_n)$) for `hybrids_solver` and `hybridsj_solver`.

Examples

Input in real mode:

```
> fsolve([x^2+y+1,x+y^2-1],[x,y])
```

$$\begin{bmatrix} 0.0 & -1.0 \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

Input in complex mode:

```
> fsolve([x^2+y+1,x+y^2-1],[x,y])
```

$$\begin{bmatrix} 0.0 & -1.0 \\ 0.226698825758 - 1.46771150871\mathrm{i} & 1.1027847152 + 0.665456951153\mathrm{i} \\ 0.226698825758 + 1.46771150871\mathrm{i} & 1.1027847152 - 0.665456951153\mathrm{i} \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

Input in any mode:

```
> cfsolve([x^2+y+1,x+y^2-1],[x,y])
```

$$\begin{bmatrix} 0.0 & -1.0 \\ 0.226698825758 - 1.46771150871\mathrm{i} & 1.1027847152 + 0.665456951153\mathrm{i} \\ 0.226698825758 + 1.46771150871\mathrm{i} & 1.1027847152 - 0.665456951153\mathrm{i} \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

```
> cfsolve([x^2+y+2,x+y^2+2],[x,y])
```

$$\begin{bmatrix} 0.5 + 1.65831239518\mathrm{i} & 0.5 - 1.65831239518\mathrm{i} \\ 0.5 - 1.65831239518\mathrm{i} & 0.5 + 1.65831239518\mathrm{i} \\ -0.5 + 1.32287565553\mathrm{i} & -0.5 + 1.32287565553\mathrm{i} \\ -0.5 - 1.32287565553\mathrm{i} & -0.5 - 1.32287565553\mathrm{i} \end{bmatrix}$$

Example

```
> fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],dnewton_solver)
```

$$[1.0, 1.0]$$

23.3.4 Numeric roots of a polynomial

The `proot` command numerically finds the roots of a squarefree polynomial.

- `proot` takes P, a squarefree polynomial, either in symbolic form or as a list of polynomial coefficients (written by decreasing order).
- `proot(`P`)` returns a list of the numeric roots of P.

Examples

Find the numeric roots of $P(x) = x^3 + 1$.

```
> proot([1,0,0,1])
```

or:

```
> proot(x^3+1)
```

$$[-1.0, 0.5 - 0.866025403784\mathrm{i}, 0.5 + 0.866025403784\mathrm{i}]$$

Find the numeric roots of $x^2 - 3$.

```
> proot([1,0,-3])
```

or:

```
> proot(x^2-3)
```

$$[-1.73205080757, 1.73205080757]$$

23.4 Solving differential equations

23.4.1 Approximating solutions of $y' = f(t, y)$

The `odesolve` command can solve first order differential equations or first order systems. This section covers equations, while systems of equations are discussed in the next section.

`odesolve` finds values of the solution of a differential equation of the form $y' = f(t, y)$; specifically, it will approximate $y(t_1)$ for a specified t_1.

`odesolve` can take its arguments in various ways. Letting t and y be the independent and dependent variables, t_0 and y_0 be the initial values, t_1 the place where you want the value of y, f be the function in the differential equation, $f(t, y)$ be an expression which determines the function f (see Section 8.2.1, p. 148 for the difference between a function and an expression).

- `odesolve` takes three or four mandatory arguments and two optional arguments:
  - *mandatory*, mandatory arguments given by one of the following sequences:
    - $f(t, y), [t, y], [t_0, y_0], t_1$
    - $f(t, y), t = t_0..t_1, y, y_0$
    - $t_0..t_1, f, y_0$
    - $t_0..t_1, (t, y)$`->`$f(t, y), y_0$
  - Optionally, `tstep=`n, to set the initial tstep value to the numeric solver from the GSL. It may be modified by the solver. It is also used to control the number of iterations of the solver by $2(t_1 - t_0)/n$ (if the number of iterations exceeds this value, the solver will stops at a time $t < t_1$).
  - Optionally, `curve`, the symbol.

- `odesolve(`*mandatory* $\langle$`,tstep=`n`,curve`$\rangle$`)` returns an approximate value of $y(t_1)$ where $y(t)$ is the solution of:

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0.$$

  With an optional argument of `curve`, the list of all the $[t, [y(t)]]$ values that were computed are returned.

Examples

```
> odesolve(sin(t*y),[t,y],[0,1],2)
```

or:

```
> odesolve(sin(t*y),t=0..2,y,1)
```

or:

```
> odesolve(0..2,(t,y)->sin(t*y),1)
```

or:

```
> f(t,y):=sin(t*y);
  odesolve(0..2,f,1)
```

$$[1.82241255674]$$

```
> odesolve(0..2,f,1,tstep=0.3)
```

$$[1.82241255675]$$

```
> odesolve(sin(t*y),t=0..2,y,1,tstep=0.5)
```

$$[1.82241255674]$$

```
> odesolve(sin(t*y),t=0..2,y,1,tstep=0.5,curve)
```

$$\begin{bmatrix} 0.0 & [1.0] \\ 0.0238917513909 & [1.00028543504] \\ 0.065808814858 & [1.00216696089] \\ 0.108895370376 & [1.00594077449] \\ \vdots & \vdots \\ 1.96462490594 & [1.8389834135] \\ 1.97769352646 & [1.83297839039] \\ 1.9908403154 & [1.82679805346] \\ 2.0 & [1.82241255674] \end{bmatrix}$$

23.4.2 Approximating solutions of the system $v' = f(t, v)$

This section covers using `odesolve` to solve first order systems of differential equations; using it to solve a single first order differential equation was discussed last section.

The `odesolve` can be used to solve a system of the form

$$x' = f(t, x),$$

where $x = [x_1, \ldots, x_n]$ is a list of unknown functions and f is a function of $n + 1$ variables with an initial condition.

`odesolve` can take its arguments in various ways. Letting t be the independent variable and $x = [x_1, \ldots, x_n]$ be a vector of dependent variables, t_0 and x_0 be the initial values, t_1 the place where

you want the value of x, f be the function in the differential equation, $f(t,x)$ be a list of expressions which determines the function f (see Section 8.2.1, p. 148 for the difference between a function and an expression).

- `odesolve` takes three or four mandatory arguments and two optional arguments:
  - *mandatory*, mandatory arguments given by one of the following sequences:
    - $f(t,x), t = t_0..t_1, x, x_0$
    - $t_0..t_1, (t,y) \mapsto f(t,y), x_0$
    - $t_0..t_1, f, x_0$
  - Optionally, `curve`, the symbol.
- `odesolve(`*mandatory* $\langle$`, curve`$\rangle$`)` returns an approximate value of $x(t_1)$ with x being the particular solution of $x'(t) = f(t, x(t))$ statisfying the initial condition $x(t_0) = x_0$. With an optional argument of `curve`, the list of all the $[t, [x(t)]]$ values that were computed by the solver are returned.

Examples

Solve the system of equations $x'(t) = -y(t)$ and $y'(t) = x(t)$:

```
> odesolve([-y,x],t=0..pi,[x,y],[0,1])
```

$$\left[-1.79045146764 \times 10^{-15}, -1\right]$$

Solve the system of equations $x'(t) = -y(t)$ and $y'(t) = x(t)$:

```
> odesolve(0..pi,(t,v)->[-v[1],v[0]],[0,1])
```

or:

```
> f(t,v):=[-v[1],v[0]]:; odesolve(0..pi,f,[0,1])
```

$$\left[-1.79045146764 \times 10^{-15}, -1\right]$$

```
> odesolve(0..pi/4,f,[0,1],curve)
```

$$\begin{bmatrix} 0.0 & [0.0, 1.0] \\ 0.0165441856471 & [-0.0165434309391, 0.999863148082] \\ 0.0325491321983 & [-0.0325433851614, 0.999470323763] \\ 0.0486049854945 & [-0.0485858499906, 0.998819010222] \\ \vdots & \vdots \\ 0.747336757246 & [-0.679687679865, 0.733501641333] \\ 0.76509544295 & [-0.692605846268, 0.721316256377] \\ 0.78286231703 & [-0.705311395415, 0.708897619897] \\ 0.785398163397 & [-0.707106781186, 0.707106781186] \end{bmatrix}$$

23.4.3 Approximating solutions of boundary value problems

The `bvpsolve` command finds an approximate solution $\hat{y}(t)$ of a boundary value problem

$$y'' = f(t, y, y'), \quad y(a) = \alpha,\ y(b) = \beta$$

on the interval $[a, b]$. The procedure uses the method of nonlinear shooting which is based on Newton and Runge-Kutta methods. Values of y and its first derivative y' are approximated at points $t_k = a + k\,\delta$, where $\delta = \frac{b-a}{N}$ and $k = 0, 1, \ldots, N$.

- `bvpsolve` takes three mandatory arguments and four optional arguments:
  - $f(t, y, y')$, an expression defining y''.
  - $[t = a\,..\,b, y]$, a list specifying the variable t, its range $[a, b]$ and the sought function y.
  - $[\alpha, \beta]$, a list of the boundary values of y.
  - Optionally, A, an initial guess for $y'(a)$ (by default, $(\beta - \alpha)/(b - a)$).
  - Optionally, N, an integer greater than or equal to 2 (by default 100).
  - Optionally, `output=`*type*, or `Output=`*type*, the type of the output, where *type* can be one of: `list` (the default), `diff`, `piecewise` or `spline`.
  - Optionally, `limit=`M, a positive integer for a limit for the number of iterations before the procedure is stopped (by default there is no limit).
- `bvpsolve(`$f(t, y, y'), [t = a\,..\,b, y], [\alpha, \beta]\ \langle, A, N,$ `output=`*type*$,$ `limit=`$M\ \rangle$`)` returns:
  - with *type* = `list`, a list of pairs $[t_k, y_k]$ where $y_k \approx y(t_k)$,
  - with *type* = `diff`, a list of triples $[t_k, y_k, y'_k]$, where $y'_k \approx y'(t_k)$,
  - with *type* = `piecewise`, a piecewise linear interpolation of the points (t_k, y_k),
  - with *type* = `spline`, a piecewise spline interpolation of the points (t_k, y_k), based on the values y'_k computed in the process.

Note that the shooting method is sensitive to roundoff errors and may fail to converge in some cases, especially when y is a rapidly increasing function. In the absence of convergence or if the maximum number of iterations is exceeded, `bvpsolve` returns `undef`. However, if the output type is `list` or `piecewise` and if $N > 2$, a slower but more stable finite-difference method (which approximates only the function y) is tried first.

Sometimes setting an initial guess A for $y'(a)$ to a suitable value may help the shooting algorithm to converge or to converge faster.

Examples

Solve $y'' = \frac{1}{8}\,(32 + 2\,t^3 - y\,y')$ with $1 \le t \le 3$ and boundary conditions $y(1) = 17$ and $y(3) = \frac{43}{3}$ (the exact solution is $y(t) = t^2 + 16/t$). Use $N = 20$, which gives an t-step of 0.01. To display approximated and exact values together, enter:

```
> aprx:=bvpsolve((32+2t^3-y*y')/8,[t=1..3,y],[17,43/3],20):;
  exct:=apply(t->t^2+16/t,linspace(1.0,3.0,21)):;
  prepend(border(aprx,exct),[t,y_k,y(t_k)])
```

t	y_k	$y\,(t_k)$
1.0	17.0	17.0
1.1	15.7554961579	15.7554545455
1.2	14.7733911821	14.7733333333
1.3	13.9977543159	13.9976923077
1.4	13.388631813	13.3885714286
⋮	⋮	⋮
2.8	13.5542890043	13.5542857143
2.9	13.9272429048	13.9272413793
3.0	14.3333333333	14.3333333333

As another example, solve $t^5 y'' = t^2\,y'^2 - 9\,y^2 + 4\,t^6$ with $1 \le t \le 2$ and boundary conditions $y(1) = 0$ and $y(2) = \ln 256$. Obtain the solution as a piecewise spline interpolation for $N = 10$ and estimate the

absolute error `err` of the approximation using the exact solution $y = t^3 \ln t$ and the `romberg` command for numerical integration. You need to explicitly set an initial guess A for the value $y'(1)$ because the algorithm fails to converge with the default guess $A = \ln 256 \approx 5.545$. Therefore let $A = 1$ instead.

```
> f:=(t^2*diff(y(t),t)^2-9*y(t)^2+4*t^6)/t^5:;
  p:=bvpsolve(f,[t=1..2,y],[0,ln(256),1],10,output=spline):;
  err:=sqrt(romberg((p-t^3*ln(t))^2,t=1..2))
```

$$3.27751904973 \times 10^{-6}$$

Note that, if the output type was set to `list` or `piecewise`, the solution would have been found even without specifying an initial guess for $y'(1)$ because the algorithm would automatically apply the alternative finite-difference method, which converges.

23.5 Numerical factorization of a matrix

Numeric Cholesky, QR, LU, and SVD factorizations of a matrix can be carried out, as described in Section 15.3, p. 372.

24 Unit objects and physical constants

24.1 Unit objects

24.1.1 Notation of unit objects

A unit object has two parts: a real number and a unit expression (a single unit or a multiplicative combination of units). The two parts are linked by the character `_` ("underscore"). For example `2_m` for 2 meters. For composite units, parenthesis must be used, e.g. `1_(m*s)`. See Table 24.1, p. 643 for a list of the basic units.

If a prefix is put before the unit then the unit is multiplied by a power of 10. For example, the prefix `k` or `K`, for kilo, indicates multiplication by 10^3. See Table 24.2 for a list of the unit prefixes. You cannot use a prefix with a built-in unit if the result gives another built-in unit.

For example, `1_a` is one are, but `1_Pa` is one pascal and not `10^15_a`.

Examples

```
> 10.5_m
```

10.5 m

which is a unit object of value 10.5 meters.

```
> 10.5_km
```

10.5 km

which is a unit object of value 10.5 kilometers.

24.1.2 Computing with units

Xcas performs usual arithmetic operations (`+`, `-`, `*`, `/`, `^`) on unit objects. Different units may be used, but for `+` and `-` they must be compatible. The result is an unit object.

- For the multiplication and the division of two unit objects $\_u_1$ and $\_u_2$, the unit of the result is written $\_(u_1 \cdot u_2)$ and $\_(u_1/u_2)$.
- For an addition or a subtraction of compatible unit objects, the result is expressed with the same unit as the first term of the operation.

Examples

```
> 1_m+100_cm
```

2.0 m

```
> 100_cm+1_m
```

200.0 cm

_A	Ampere	_ha	Hectare
_Angstrom	Angstrom	_hp	Horsepower
_Bq	Becquerel	_in	Inch
_Btu	Btu British thermal unit	_inH20	Inches of water at 60° Fahrenheit
_Ci	Curie	_inHg	Inches of mercury at 0° Celsius
_F	Farad	_j	Day
_Fdy	Faraday	_kWh	Kilowatt-hour
_Gal	Gal (0.01 m/s^2)	_kg	Kilogram
_Gy	Gray	_kip	Kilopound-force
_H	Henry	_km	Kilometre
_Hz	Hertz	_knot	nautical miles per hour
_J	Joule	_kph	Kilometers per hour
_K	Kelvin	_l	Liter
_Kcal	Kilocalorie	_lam	Lambert
_MHz	Megahertz	_lb	pound (16 oz)
_MW	Megawatt	_lbf	Pound-force
_MeV	Megaelectronvolt	_lbmol	Pound-mole
_N	Newton	_lbt	Troy pound
_Ohm	Ohm	_lep	Liter of oil equivalent
_P	Poise (measures viscosity)	_liqpt	US liquid pint (1 US gallon=8 US liquid pints)
_Pa	Pascal	_lm	Lumen
_R	Roentgen	_lx	Lux
_Rankine	Degree Rankine	_lyr	Light year
_S	Siemens	_m	Metre (unit)
_St	Stokes	_mho	Mho
_Sv	Sievert	_miUS	US statute mile
_T	Tesla	_mi^2	Square international mile.
_V	Volt	_mil	Mil
_W	Watt	_mile	International mile
_Wb	Weber	_mille	Nautical mile
_Wh	Watt-hour	_ml	millilitre
_a	Are (100 m^2)	_mm	Millimetre
_acre	Acre	_mmHg	Millimeter of mercury (torr), 0 degree Celsius
_arcmin	Minute of arc	_mn	Minute
_arcs	Second of arc	_mol	Mole
_atm	Atmosphere	_mph	Miles per hour
_au	Astronomical unit	_oz	Ounce
_b	Barn	_ozUK	UK fluid ounce
_bar	Bar	_ozfl	US fluid ounce
_bbl	Barrel	_ozt	Troy ounce
_bblep	Barrel of oil equivalent	_pc	Parsec
_bu	Bushel (8 UK gallons)	_pdl	Poundal (force)
_buUS	American bushel	_ph	Phot
_cal	Calorie	_pk	US peck
_cd	Candela	_psi	Pounds per square inch
_chain	Chain (66 feet or 22 yards)	_ptUK	UK pint (1 UK gallon=8 UK pints)
_cm	Centimetre	_qt	Quart
_ct	Carat	_rad	Radian
_cu	US cup	_rd	Rad (0.01 Gy)
_d	Day	_rem	Rem
_dB	Decibel	_rod	Rod 1 (5.029215842 m)
_deg	Degree (angle)	_rpm	Revolutions per minute
_degreeF	Degree Fahrenheit	_s	Second
_dyn	Dyne	_s	second
_eV	Electron volt	_sb	Stilb
_erg	Erg	_slug	Slug
_fath	Fathom	_sr	Steradian
_fbm	Board foot	_st	Stere
_fc	Footcandle ($\approx$ 10.764 lux)	_t	Metric ton
_fermi	Fermi	_tbsp	Tablespoon
_flam	Footlambert	_tec	Tonne of coal equivalent
_fm	Fathom	_tep	Tonne of oil equivalent
_ft	International foot	_tex	tex (10^{-6} kg/m)
_ftUS	Survey foot	_therm	EEC therm
_g	Gram	_ton	Short ton (2000 pounds)
_ga	Standard freefall	_tonUK	Long (UK) ton
_galC	Canadian gallon	_torr	Torr (mmHg)
_galUK	UK gallon	_tr	tour (2π rad)
_galUS	US gallon	_tsp	Teaspoon
_gf	Gram-force	_u	Atomic mass unit
_gmol	Gram-mole	_yd	International yard
_gon	Grade	_yr	Year
_grad	Grade	_h	Hour
_grain	Grain ($\approx 0,0648$ grams)	_μ	Micron

Table 24.1: Physical units in Xcas

Prefix	Name	(*10^) n	Prefix	Name	(*10^) n
Y	yota	24	d	deci	−1
Z	zeta	21	c	cent	−2
E	exa	18	m	mili	−3
P	peta	15	μ	micro	−6
T	tera	12	n	nano	−9
G	giga	9	p	pico	−12
M	mega	6	f	femto	−15
k or K	kilo	3	a	atto	−18
h or H	hecto	2	z	zepto	−21
D	deca	1	y	yocto	−24

Table 24.2: Unit prefixes in XCAS

24.1.3 Converting units into MKSA units

The MKSA units are a system of units based on the meter, kilogram, second and ampere and usually used in scientific work. The `mksa` command converts a unit object into a unit object written with the compatible MKSA base unit.

- `mksa` takes u, a unit object.
- `mksa`(u) returns the unit object in terms of the MKSA units.

Example

```
> mksa(15_C)
```

15.0 sA

24.1.4 Converting units

The `convert` command (see Section 10.1.10, p. 195) can convert a unit object into another compatible unit.

- `convert` takes two arguments:
  - *unitobj*, a unit object.
  - u, a unit compatible with *unitobj*.
- `convert`(*unitobj*, u) returns *unitobj* in terms of u.

Recall that the `=>` operator is the infixed version of `convert`.

Examples

```
> convert(1_h,_s)
```

3600.0 s

```
> convert(3600_s,_h)
```

1.0 h

24.1.5 Converting between Celsius and Fahrenheit

The `Celsius2Fahrenheit` command converts a temperature in degrees Celsius to the equivalent temperature in degrees Fahrenheit.

- `Celsius2Fahrenheit` takes T, a number representing representing a temperature in degrees Celsius.
- `Celsius2Fahrenheit(`T`)` returns the number representing the temperature in Fahrenheit.

The `Fahrenheit2Celsius` command converts temperatures in degrees Fahrenheit to degrees Celsius.

- `Fahrenheit2Celsius` takes T, a number representing representing a temperature in degrees Fahrenheit.
- `Fahrenheit2Celsius(`T`)` returns the number representing the temperature in degrees Celsius.

Examples

```
> Celsius2Fahrenheit(a)
```

$$\frac{9}{5}a + 32$$

```
> Fahrenheit2Celsius(212)
```

$$100$$

24.1.6 Factoring a unit

The `ufactor` command factors units in unit objects.

- `ufactor` takes two arguments:
  - *unitobj*, a unit object.
  - u, the unit to factor.
- `ufactor(`*unitobj*, u`)` returns a unit object multiplied by the remaining `MKSA` units.

Examples

```
> ufactor(3_J,_W)
```

$$3.0\ \text{Ws}$$

```
> ufactor(3_W,_J)
```

$$3.0\ \text{Js}^{-1}$$

24.1.7 Simplifying units

The `usimplify` command simplifies a unit in an unit object.

- `usimplify` takes *unitobj*, a unit object.
- `usimplify(`*unitobj*`)` returns *unitobj* with the units simplified.

_F_	Faraday constant	_h_	Planck's constant
_G_	Gravitational constant	_hbar_	Dirac's constant
_I0_	Reference intensity	_k_	Boltzmann constant.
_NA_	Avogadro's number	_kq_	k/q (Boltzmann/electron charge)
PSun	Power at the surface of the Sun	_lambda0_	Photon wavelength (ch/e)
REarth	Radius of the Earth	_lambdac_	Compton wavelength
RSun	Radius of the Sun	_mEarth_	Mass of the Earth
_R_	Universal gas constant	_me_	Electron rest mass
Rinfinity	Rydberg constant	_mp_	Proton rest mass
StdP	Standard pressure	_mpme_	Quotient m_p/m_e (proton mass/electron mass)
StdT	Standard temperature	_mu0_	Permeability of vacuum
_Vm_	Molar volume	_muB_	Bohr magneton
_a0_	Bohr radius	_muN_	Nuclear magneton
alpha	Fine structure constant	_phi_	Magnetic flux quantum
angl	180 degree angle	_q_	Charge of an electron
_c3_	Wien displacement constant	_qe_	Electron charge
_c_	Speed of light in vacuum	_qepsilon0_	$q \cdot \varepsilon_0$ (electron charge*permittivity)
epsilon0	Permittivity of vacuum	_qme_	Quotient q/m_e (charge/electron mass)
epsilon0q	ε_0/q (permittivity/electron charge)	_rad_	1 radian
epsilonox	Dielectric constant of Silicon dioxide	_sd_	Duration of a sidereal day
epsilonsi	Dielectric constant	_sigma_	Stefan-Boltzmann constant
_f0_	Photon frequency (e/h)	_syr_	Duration of a sidereal year
_g_	Acceleration of gravity	_twopi_	$2\,\pi$

Table 24.3: Physical constants in Xcas

Example

```
> usimplify(3_(W*s))
```

$$3.0\ \mathrm{J}$$

24.2 Constants

24.2.1 Notation of physical constants

If you want to use a physical constants inside Xcas, put its name between two characters _ ("underscore"). Do not confuse physical constants with symbolic constants; for example, e, π are symbolic constants and _c_,_NA_ are physical constants. The physical constants are in the Phys menu, Constant sub-menu, and table 24.3 gives the Constants Library:

Example

To get the object representing the speed of light in vacuum, enter:

```
> _c_
```

$$1\ c$$

Use the mksa command (see Section 24.1.3, p. 644) to put this in terms of standard units:

```
> mksa(_c_)
```

$$299792458.0\ \mathrm{ms}^{-1}$$

25 Programming

Xcas has a programming language that allows you to write complex programs in several different syntax flavours. This section explains the details of the language, how to enter programs in Xcas (most notably new functions) and how to debug them. In the end, it is briefly explained how to extend and use `giac` from C++.

25.1 Functions, programs and scripts

25.1.1 Program editor

Xcas provides you with a program editor, which you can open with Alt+P (see Figure 25.1). This can be useful for writing small programs, but for writing larger programs you may want to use your usual editor. (Note that this requires an editor, such as Emacs, and not a word processor.) If you use your own editor, then you will need to save the program to a file, such as `myprog.cxx`, and then load it into Xcas by creating a programming entry with Alt+P and then selecting Prog ▶ Load from the entry menu.

25.1.2 Functions

You have already seen functions defined with `:=`. For example, to define a function `sumprod` which takes two inputs and returns a list with the sum and the product of the inputs, you can enter:

```
> sumprod(a,b):=[a+b,a*b]
```

Afterwards, use this new function.

```
> sumprod(3,5)
```

[8, 15]

You can define functions that are computed with a sequence of instructions by putting the instructions between braces, where each command ends with a semicolon. To use local variables, you can declare them with the `local` keyword, followed by the variable names. The value returned by the function will be indicated with the `return` keyword. For example, the above function `sumprod` could also be defined by:

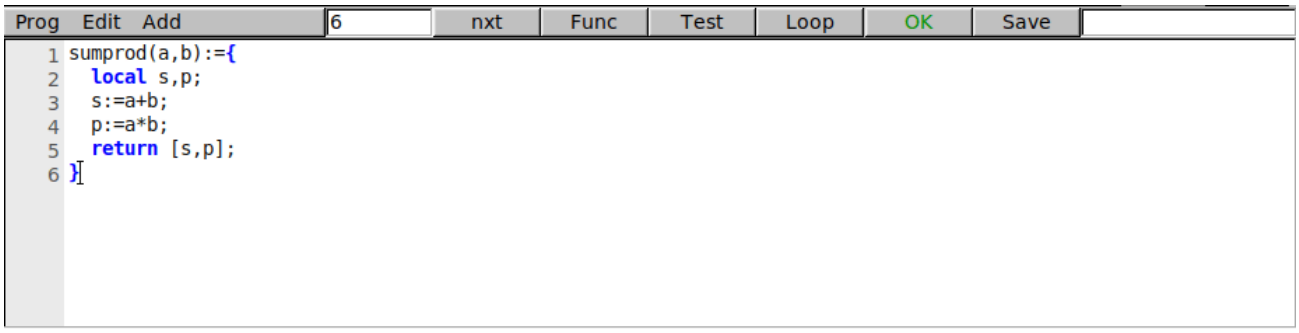


Figure 25.1: Programming editor in Xcas

```
sumprod(a,b):={
  local s,p;
  s:=a+b;
  p:=a*b;
  return [s,p];
}
```

To avoid using braces, the **proc** and **end** keywords may be used like this:

```
sumprod:=proc(a,b)
  local s,p;
  s:=a+b;
  p:=a*b;
  return [s,p];
end
```

Yet another way to define a function is by using the **function**...**endfunction** construction. With this approach, the function name and parameters follow the **function** keyword. This is otherwise like the previous approach. The `sumprod` function could be defined by:

```
function sumprod(a,b)
  local s,p;
  s:=a+b;
  p:=a*b;
  return [s,p];
endfunction
```

25.1.3 Local variables

Local variables in a function definition can be given initial values in the line they are declared in by putting their initialization in parentheses; for example,

```
local a,b;
a:=1;
```

is the same as

```
local (a:=1), b;
```

Local variables should be given values within the function definition. If you want to use a local variable as a symbolic variable, then you can indicate that with the `assume` command (see Section 3.3.8, p. 39). For example, if you define a function `myroots` by

```
myroots (a):={
  local x;
  return solve(x^2=a,x);
}
```

then calling

```
> myroots(4)
```

will simply return the empty list. You could leave `x` undeclared, but that would make `x` a global variable and could interact with other functions in unexpected ways. You can get the behavior you probably expected by explicitly assuming `x` to be a symbol;

```
myroots(a):={
  local x;
  assume(x,symbol);
```

```
  return solve(x^2=a,x);
}
```

(Alternatively, you could use `purge(x)` instead of `assume(x,symbol)`.) Now if you enter

```
> myroots(4)
```

you will get

$$[-2, 2]$$

25.1.4 Default values of the parameters

You can give the parameters of a function default values by putting *parameter*=*value* in the parameter list of the function. For example, if you define a function:

```
f(x,y=5,z):={
  return x*y*z;
}
```

then:

```
> f(1,2,3)
```

$$6$$

since the product $1 \cdot 2 \cdot 3 = 6$. If you give `f` only two values as input:

```
> f(3,4)
```

$$60$$

since the values 3 and 4 will be given to the parameters which do not have default values; in this case, `y` will get its default value `5` while `3` and `4` will be assigned to `x` and `z`, respectively. The result is $xyz = 3 \cdot 5 \cdot 4 = 60$.

25.1.5 Programs

A program is similar to a function, and is written like a function without a return value. Programs are used to display results or to create drawings. It is a good idea to turn a program into a function by putting `return 0` at the end; this way you will get a response of `0` when the program executes.

25.1.6 Scripts

A script is a file containing a sequence of instructions, each ending with a semicolon.

25.1.7 Code blocks

A code block, such as used in defining functions, is a sequence of statements delimited by braces or by `begin` and `end`. Each statement must end with a semicolon. If the block makes up a function, you can step through it one statement at a time by using the debugger (see Section 25.6, p. 670).

25.2 Basic instructions

25.2.1 Comments

The characters `//` indicate that you are writing a comment; any text between `//` and the end of the line will be ignored by Xcas.

25.2.2 Input

Creating input windows. The `input` or `Input` command prompts the user for the value of a variable.

- `input` takes an arbitrary number of commands:

  vars, a sequence of variable names, each one optionally preceded by a string.

- `input(`*vars*`)` brings up a box where the user can enter a value for each variable.

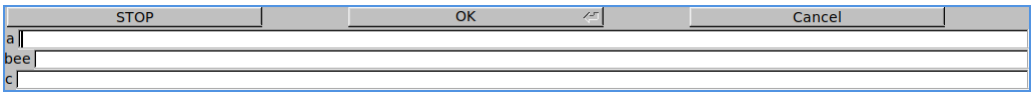
  If a variable is preceded with a string, then that string will be the prompt for the variable, otherwise the variable name will be the prompt.

Examples

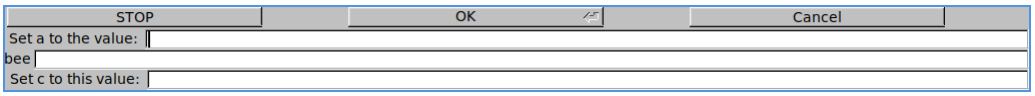
```
> input(a)
```


```
> input("Set a to the value: ",a)
```


```
> input(a,bee,c)
```




```
> input("Set a to the value: ",a,bee,"Set c to this value: ",c)
```



If the value that you enter for `input` is a string, it should be between quotes. If you want the user to enter a string without having to use the quotes, use the `InputStr` or `textinput` command, which is just like `input` except that it will assume any input is a string and so the user won't need to use quotes.

Creating output windows. The `output` or texttttOutput command creates message windows.

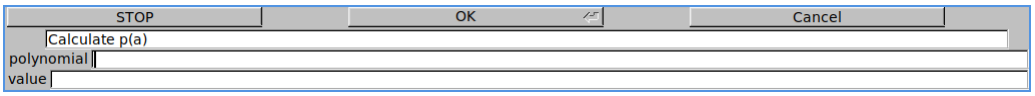
- `output` takes `strs`, a sequence of strings or variables which represent strings.
- `output(`*strs*`)` creates a message window displaying the concatentation of the strings.

Examples

```
> s:="message"
  output("This is a ",s)
```

Use `output` to add information to the input window:

```
> input(output("Calculate p(a)"),"polynomial",p,"value",a)
```



25.2.3 Reading a single keystroke

The `getKey` command gets the next keystroke.

- `getKey` takes no arguments.
- `getKey()` returns the ASCII code of the next keystroke.

For example, if you enter

```
> asciicode:=getKey()
```

and hit the `A` key, then the variable `asciicode` will have the value 65, which is the ASCII code of capital A.

25.2.4 Checking required conditions

The `assert` command breaks out of a function with an error.

- `assert` takes *bool*, a boolean.
- `assert(`*bool*`)` does nothing if *bool* is true, it returns from the function with an error if *bool* is false.

Example

Define the function:

```
> sqofpos(x):={ assert(x>0); return x^2; }
```

then:

```
> sqofpos(4)
```

16

```
> sqofpos(-4)
```

"assert failure: x>0 Error: Bad Argument Value"

since $-4 > 0$ is false.

25.2.5 Checking the type of the argument

The `type` command finds the type of its input.

- `type` takes one argument: *arg*, an object.
- `type(`*arg*`)` returns an integer indicating the type of *arg*.

  The integer is given as a constant symbol which is equal to the integer. The possible values are:

  - `1`, equivalently `real`, `double` or `DOM_FLOAT`.
  - `2`, equivalently `integer` or `DOM_INT`.

  - 4, equivalently `complex` or `DOM_COMPLEX`.
  - 6, equivalently `identifier` or `DOM_IDENT`.
  - 7, equivalently `vector` or `DOM_LIST`.
  - 8, equivalently `expression` or `DOM_SYMBOLIC`.
  - 10, equivalently `rational` or `DOM_RAT`.
  - 12, equivalently `string` or `DOM_STRING`.
  - 13, equivalently `func` or `DOM_FUNC`.

The `getType` command is similar to `type` in that it takes an object and returns the type, but it has different possible return values. It is included for compatibility reasons.

- `getType` takes *obj*, an object.
- `getType(`*obj*`)` returns the type of *obj*, which in this case means one of: `NUM`, `VAR`, `STR`, `EXPR`, `NONE`, `PIC`, `MAT` or `FUNC`.

Examples

```
> type(4)
```

integer

```
> type(3.1)==DOM_FLOAT
```

true

```
> getType(3.14)
```

NUM

```
> getType(x)
```

VAR

Subtypes

XCAS has various types of lists; the `subtype` command can determine what kind of list it is.

- `subtype` takes L, a list (in `DOM_LIST`).
- `subtype(`L`)` returns an integer indicating what type of list L is.

  The possible values are:
  - 1 is L is a sequence.
  - 2 if L is a set.
  - 10 if L is a polynomial represented by a list (see Section 11.1, p. 211).
  - 0 if L is not one of the above types of list.

Example

```
> subtype(1,2,3)
```

1

Object comparison

The `compare` operator compares two objects taking their type into account.

- `compare` takes two arguments: a, b, two objects.
- `compare`(a, b) returns
  - 1 (`true`) if `type`$(a) <$ `type`(b) or if `type`$(a) =$ `type`(b) and a is less than b in the ordering of their type.
  - 0 (`false`) otherwise.

Examples

```
> compare("a","b")
```

1

since `"a"` and `"b"` have the same type (`string`) and `"a"` is less than `"b"` in the string ordering.

If `b` is a formal variable:

```
> compare("a",b)
```

0

since the type of `"a"` is `string` (the integer 12) while the type of `b` is `identifier` (the integer 6) and `12` is not less than `6`.

25.2.6 Printing

The `print` or `Disp` command prints in a special pane called message area, located between input and output panes of a command line entry (the text is shown in green).

- `print` takes *seq*, a sequence of objects.
- `print`(*seq*) returns 1 and prints the *seq* in a special pane just above the output line.

Examples

```
> print("Hello")
```

Hello

1

```
> a:=12
  print("a=",a)
```

“a=,12”

1

The `ClrIO` command erases printing on the level it was typed.

- `ClrIO` takes no arguments and no parentheses.
- `ClrIO` returns 1 and erases any printing on the special pane above the output line on the level it was typed.

Example

```
> print("Hello"); ClrIO
```

(1, 1)

25.2.7 Displaying exponents

The `printpow` command determines how the `print` command will print exponents in the special pane above the output line.

- `printpow` takes n, which is either -1, 0 or 1 (by default 1).
- `printpow(`n`)` sets the style for printing exponents with the `print` command.
  - If $n = -1$, `print(`a`^`b`)` will subsequently print a`**`b`)`.
  - If $n = 0$, `print(`a`^`b`)` will subsequently print `pow(`a, b`)`.
  - If $n = 1$, `print(`a`^`b`)` will subsequently print a`^`b.

Examples

```
> print(x^3):;
x^3

> printpow(-1):; print(x^3):;
x**3

> printpow(0):; print(x^3):;
pow(x,3)

> printpow(1):; print(x^3):;
x^3
```

25.2.8 Infixed assignments

The infixed operators `=>`, `:=`, and `=<` can all store a value in a variable, but their arguments are in different order. (See Section 3.3.2, p. 36 and Section 3.3.3, p. 37.) Also, `:=` and `=<` have different effects when the first argument is an element of a list stored in a variable, since `=<` modifies list elements by reference (see section 25.2.10).

`=>` is the infixed version of `sto`, it stores the value in the first argument in the variable in the second argument. Both

```
> 4=>a
```

and

```
> sto(4,a)
```

store the value 4 in the variable `a`.

`:=` and `=<` both have a variable as the first argument and the value to store in the variable as the second argument. Both

```
> a:=4
```

and

```
> a=<4
```

store the value 4 in the variable a.

However, suppose you have entered:

```
> A:=[0,1,2,3,4]:; B:=A
```

and you want to change A[3], then the commandline

```
> A[3]=<33
```

will change both A and B:

```
> A,B
```

$$[0,1,2,33,4],[0,1,2,33,4]$$

Here, A pointed to the list [0,1,2,3,4] and after setting B to A, B also pointed to [0,1,2,3,4]. Changing an element of A by reference changes the list that A points to, which B also points to.

Note that multiple assigments can be made using sequences or lists. Both

```
> [a,b,c]:=[1,2,3]
```

and

```
> (a,b,c):=(1,2,3)
```

assign a the value 1, b the value 2, and c the value 3. If multiple assignments are made this way and variables are on the right hand side, they will be replaced by their values before the assignment. If a contains 5 and you enter:

```
> (a,b):=(2,a)
```

then b will get the previous value of a, 5, and not the new value of a, 2.

25.2.9 Assignment by copying

The copy command creates a copy of its argument, which is typically a list of some type. If B is a list and A:=B, then A and B point to the same list, and so changing one will change the other. But if A:=copy(B), then A and B will point to different lists with the same values, and so can be changed individually.

Example

```
> B:=[[4,5],[2,6]]:;
  A:=B:;
  C:=copy(B):;
  A,B,C
```

$$\begin{bmatrix}4&5\\2&6\end{bmatrix},\begin{bmatrix}4&5\\2&6\end{bmatrix},\begin{bmatrix}4&5\\2&6\end{bmatrix}$$

```
> B[1]=<[0,0]:;
  A,B,C
```

$$\begin{bmatrix}4&5\\0&0\end{bmatrix},\begin{bmatrix}4&5\\0&0\end{bmatrix},\begin{bmatrix}4&5\\2&6\end{bmatrix}$$

25.2.10 Difference between operators := and =<

The := and =< assignment operators have different effects when they are used to modify an element of a list contained in a variable, since =< modifies the element by reference. Otherwise, they will have the same effect.

Example

```
> A:=[1,2,3]
```

Now

```
> A[1]:=5
```

and

```
> A[1]=<5
```

both change `A[1]` to 5:

```
> A
```

$$[1,5,3]$$

but they do it in different ways. The command `A[1]=<5` changes the middle value in the list that `A` originally pointed to, and so any other variable pointing to the list will be changed, but `A[1]:=5` will create a duplicate list with the middle element of 5, and so any other variable pointing to the original list won't be affected.

Examples

```
> A:=[0,1,2,3,4]:;
  B:=A:;
  B[3]=<33:;
  A,B
```

$$[0,1,2,33,4]\,,[0,1,2,33,4]$$

```
> A:=[0,1,2,3,4]:;
  B:=A:;
  B[3]:=33:;
  A,B
```

$$[0,1,2,3,4]\,,[0,1,2,33,4]$$

If `B` is set equal to a copy of `A` instead of `A`, then changing `B` won't affect `A`.

```
> A:=[0,1,2,3,4];
  B:=copy(A);
  B[3]=<33;
  A,B
```

$$[0,1,2,3,4]\,,[0,1,2,33,4]$$

25.3 Control structures

25.3.1 Conditional statements

The Xcas language has different ways of writing "if... then" statements (see Section 5.1.3, p. 50). The standard version of such a statement consists of the **if** keyword, followed by a boolean expression (see Section 5.1, p. 50) in parentheses, followed by a statement block (see Section 25.1.7, p. 649) which will be executed if the boolean is true. You can optionally add an **else** keyword followed by a statement block which will be executed if the boolean is false.

if (*boolean*) *true-block* ⟨**else** *false-block*⟩

Recall that the blocks need to be delimited by braces or by **begin** and **end**.

Examples

```
> a:=3:; b:=2:;
  if (a>b) { a:=a+5; b:=a-b; }:;
  a,b
```

$$8, 6$$

since `a>b` evaluates to true, and so the variable `a` resets to `8` and `b` resets to the value `6`.

```
> a:=3:; b:=2:;
  if (a<b) { a:=a+5; b:=a-b; } else { a:=a-5; b:=a+b; }:;
  a,b
```

$$-2, 0$$

since `a>b` evaluates to false, and so the variable `a` resets to `-2` and `b` resets to the value `0`.

The "if... then... else... end" structure. An alternate way to write an **if** statement is to enclose the code block in **then** and **end** instead of braces:

if (*boolean*) **then** *true-block* ⟨**else** *false-block*⟩ **end**

- In this case, it is usually not necessary to enclose the boolean in parentheses.
- Instead of the keyword **end**, you can also use **fi**.

Examples

```
> a:=3:;
  if a>1 then a:=a+5; end
```

$$8$$

```
> a:=8:;
  if a>10 then a:=a+10; else a:=a-5; end
```

$$3$$

This input can also be written as:

```
> si a>10 alors a:=a+10; sinon a:=a-5; fsi
```

Nesting conditional statements. Several **if** statements can be nested. For example:

```
  if a>1 then a:=1; else if a<0 then a:=0; else a:=0.5; end; end
```

A simpler way is to replace the **else if** by **elif** and combine the **end**s:

```
  if a>1 then a:=1; elif a<0 then a:=0; else a:=0.5; end
```

In general, such a combination can be written

if (*boolean1*) **then**
  block1;
elif (*boolean2*) **then**
  block2;
...
elif (*booleanN*) **then**
  blockN;
else
  last block;
end

where the last **else** is optional. For example, to define a function f by

$$f(x) = \begin{cases} 8 & \text{if } x > 8 \\ 4 & \text{if } 4 < x \leq 8 \\ 2 & \text{if } 2 < x \leq 4 \\ 1 & \text{if } 0 < x \leq 2 \\ 0 & \text{if } x \leq 0 \end{cases}$$

you may enter:

```
f(x):={
  if (x>8) then
  return 8;
  elif (x>4) then
  return 4;
  elif (x>2) then
  return 2;
  elif (x>0) then
  return 1;
  else
  return 0;
  fi;
}
```

25.3.2 Switch statement

The **switch** statement can be used when you want the value of a block to depend on an integer. It takes one argument, an expression which evaluates to an integer. It should be followed by a sequence of **case** statements, which takes the form **case** followed by an integer and then a colon, which is followed by a code block to be executed if the expression equals the integer. At the end is an optional **default** statement, which is followed by a code block to be executed if the expression does not equal any of the given integers.

switch(*n*) {
  case n_1: *block* n_1
  case n_2: *block* n_2
  ...
  case n_k: *block* n_k
  default: *default_block*

Recall that the blocks need to be delimited by braces or by **begin** and **end**.

Example

As an example of a program which performs an operation on the first two variables depending on the third, you could enter (see Section 25.1.1, p. 647):

```
oper(a,b,c):={
  switch (c) {
    case 1:  { a:=a+b; break; }
    case 2:  { a:=a-b; break; }
    case 3:  { a:=a*b; break; }
    default: { a:=a^b; }
```

```
    }
    return a;
  }
```

Then:

```
> oper(2,3,1)
```

$$5$$

since the third argument is `1`, and so `oper(a,b,c)` will return $a+b$, and:

```
> oper(2,3,2)
```

$$-1$$

since the third argument is `2` and so `oper(a,b,c)` will return $a-b$.

25.3.3 For loop

The for-loop has three different forms, each of which uses an index variable. If the for-loop is used in a program, the index variable should be declared as a local variable. (Recall that `i` represents the imaginary unit, and so cannot be used as the index.)

The first form. For the first form, the `for` keyword is followed by the starting value for the index, the end condition, and the increment step, separated by semicolons and in parentheses. Afterwards is a block of code to be executed for each iteration, where j is the index and j_0 is the starting value of j:

$$\texttt{for (}j\texttt{:=}j_0\texttt{;}\ \ end\_cond\texttt{;}\ \ increment\_step\texttt{)}\ \ block$$

Example

To add the even numbers less than 100, you can start by setting the running total to 0:

```
> S:=0
```

Then use a `for` loop to do the summation:

```
> for (j:=0;j<100;j:=j+2) { S:=S+j }
```

$$2450$$

The second form. The second form of a for-loop has a fixed increment for the index. It is written out with the `for` keyword followed by the index, then `from`, the initial value, `to`, the ending value, `step`, the size of the increment, and finally the statements to be executed between `do` and `end_for`:

$$\texttt{for}\ j\ \texttt{from}\ j_0\ \texttt{to}\ j_{max}\ \texttt{step}\ k\ \texttt{do}\ statements\ \texttt{end\_for}$$

where j is the index, j_0 is the initial value of j, j_{max} is the ending value of j, k is the step size of j, and *statements* are executed for each value of j.

Example

Again, to add the even numbers less than 100, you can start by setting the running total to 0:

```
> S:=0
```

then use the second form of the `for` loop to do the summation:

```
> for j from 2 to 98 step 2 do S:=S+j; end_for
```

or (a French version of this syntax):

```
> pour j de 2 jusque 98 pas 2 faire S:=S+j; fpour
```

2450

The third form. The third form of a for-loop lets you iterate over the values in a list (or a set or a range). In this form, the `for` keyword is followed by the index, then `in`, the list, and then the instructions between `do` and `end_for` or `od`:

`for` *j* `in` *L* `do` *statements* `end_for`

where *j* is the index and *L* is the list to iterate over.

Example

To add all integers from 1 to 100, you can again set the running total `S` to 0:

```
> S:=0
```

then use the third form of the `for` loop to add the integers:

```
> for j in 1..100 do S:=S+j; end_for
```

or:

```
> pour j in 1..100 faire S:=S+j; fpour
```

5050

25.3.4 Repeat loop

The repeat-loop allows you to repeat statements until a given condition is met. To use it, enter `repeat`, the statements, the keyword `until` followed by the condition, a boolean:

`repeat` *statements* `until` *bool*

Example

If you want the user to enter a value for a variable `x` which is greater than 4, you could use:

```
repeat
  input("Enter a value for x (greater than 4)",x);
until (x>4);
```

which could also be written as

```
repeter
  input("Enter a value for x (greater than 4)",x);
jusqua (x>4);
```

25.3.5 While loop

The while-loop is used to repeat a code block as long as a given condition holds. To use it, enter `while`, the condition in parentheses, and then a code block.

`while` (*boolean*) *block*

Example

Add the terms of the harmonic series $1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\cdots$ until a term is less than 0.05.

You can initialize the sum `S` to 0 and let `j` be the first term 1. Input:

```
> S:=0; j:=1
```

Then use a while loop.

```
> while (1/j>=0.05) { S:=S+1/j; j:=j+1; }
```

or:

```
> tantque (1/j>=0.05) faire S:=S+1/j; j:=j+1; ftantque
```

then:

```
> S
```

$$\frac{55835135}{15519504}$$

Note that a **while** loop can also be written as a **for** loop. For example, as long as `S` is set to 0 and `j` is set to 1, the above loop can be written as

```
> for (;1/j>=0.05;) { S:=S+1/j; j:=j+1; }
```

or, with only `S` set to 0,

```
> for (j:=1; 1/j>=0.05; j++) { S:=S+1/j; }
```

which, of course, yields the same result.

25.3.6 Breaking out of loop

The **break** command exits a loop without finishing it.

- **break** takes no arguments or parentheses.
- **break** exits the current loop.

Example

Define a program:

```
testbreak(a,b):={
  local r;
  while (true) {
    if (b==0) { break; }
    r:=irem(a,b);
    a:=b;
    b:=r;
  }
  return a;
}
```

Then:

```
> testbreak(4,0)
```

$$4$$

since the **while** loop is interrupted when `b` is 0 and `a` is 4.

25.3.7 Skipping to the next iteration of a loop

The `continue` command will skip the rest of the current iteration of a loop and go to the next iteration.

- `continue` takes no arguments or parentheses.
- `continue` goes to the next iteration of the current loop without finishing the current iteration.

Example

If you enter:

```
S:=0;
for (j:=1,j<=10;j++) {
  if (j==5) { continue; }
  S:=S+j;
}
```

then `S` will be 50, which is the sum of the integers from 1 to 10 except for 5, since the loop never gets to `S:=S+j` when `j` is equal to 5.

25.3.8 Changing the order of execution

The `goto` command will tell a program to jump to a different spot in a program, where the spot needs to have been marked with `label`. They both must have the same argument, which is simply a sequence of characters.

- `label` takes *mark*, a sequence of characters.
- `label(`*mark*`)` labels the position in the program with *mark*.
- `goto` takes *mark*, a sequence of characters.
- `goto(mark)` goes to the part of the program labeled with *mark*.

Example

The following program will add the terms of the harmonic series until the term is less than some specified value `eps` and print the result.

```
harmsum(eps):={
  local S,j;
  S:=0;
  j:=0;
  label(spot);
  j:=j+1;
  S:=S+1/j;
  if (1/j>=eps) goto (spot);
  print(S);
  return 0;
}
```

25.4 Errors

25.4.1 Handling errors

Some commands produce errors, and if your program tries to run such a command it will halt with an error. The **try** and **catch** commands can help you to avoid halting the program; put potentially problematic statements in a block following **try**, and immediately after the block put **catch** with an argument of an unused symbol, and follow that with a block of statements that can handle the error.

try *tryblock* **catch** (*symbol*) *catchblock*

If *tryblock* does not produce an error, then the code

catch (*symbol*) *catchblock*

is not reached. Otherwise, if *tryblock* does produce an error, then a string describing the error is assigned to *symbol*, and *catchblock* is evaluated.

Examples

The command below produces an error:

```
> [[1,1]]*[[2,2]]
```

Error: Invalid dimension

However, the following command does not produce an error:

```
> try { [[1,1]]*[[2,2]] } catch (err) { print("The error is --- "+err) }
```

The error is — Error: Invalid dimension

1

With the following program:

```
test(x):={
  local y,str,err;
  try {
    y:=[[1,1]]*x;
    str:="This produced a product.";
  } catch (err) {
    y:=x;
    str:="This produced an error "+err+" The input is returned.";
  }
  print(str);
  return y;
}
```

input:

```
> test([[2],[2]])
```

"This produced a product."

[4]

with the text in the pane above the output line.

```
> test([[2,2]])
```

"This produced an error Error: Invalid dimension The input is returned."

[[2, 2]]

with the text in the pane above the output line.

You can catch this error in other programs. Consider the program:

```
g(x):={
  try {
    return f(x);
  } catch (err) {
    x:=0;
  }
  return x;
}
```

then:

```
> g(12)
```

12

since 12 is an integer. With a non-integer input, `f(x)` gives an error and so `g(x)` returns 0:

```
> g(1.2)
```

0

25.4.2 Throwing exceptions

You can produce your own string to describe an error message with the **throw** or `error` or `ERROR` command.

- **throw** takes *str*, a string describing an error.
- `throw(`*str*`)` generates an error with error string *str*, possibly to be caught by **catch**.

Example

With the following program:

```
f(x):={
  if (type(x)!=DOM\_INT)
    throw("Not an integer");
  else
    return x;
}
```

input:

```
> f(12)
```

12

since 12 is an integer.

```
> f(1.2)
```

will signal an error

```
Not an integer Error: Bad Argument Value
```

since 1.2 in not an integer.

25.5 Other useful instructions

25.5.1 Defining a function with a variable number of arguments

The `args` command returns the list of arguments of a function.

- `args` takes no arguments.
- `args` (or `args(NULL)`) returns a list of the arguments of the current function, starting with the name of the function at index 0.

Note that `args()` will not work, the command must be called as `args` or `args(NULL)`. You can also use `(args)[0]` to get the name of the function and `(args)[1]` to get the first argument, etc., but the parentheses about `args` is mandatory.

Examples

```
> testargs():={ local y; y:=args; return y[1]; }:;
  testargs(12,5)
```

12

As another example, enter the function:

```
total():={
  local s,a;
  a:=args;
  s:=0;
  for (k:=1;k<size(a);k++) {
    s:=s+a[k];
  }
  return s;
}
```

then:

```
> total(1,2,3,4)
```

10

25.5.2 Assignments in a program

Recall that the `=<` operator will change the value of a single entry in a list or matrix by reference (see Section 3.3.3, p. 37). This make it efficient when changing many values, one at a time, in a list, as might be done by a program.

You must be careful when doing this, since your intent might be changed when a program is compiled. For example, if a program contains

```
local a;
a:=[0,1,2,3,4];
...
a[3]=<33;
```

then in the compiled program, `a:=[0,1,2,3,4]` will be replaced by `a:=[0,1,2,33,4]`. To avoid this, you can assign a copy of the list to `a`; you could write:

```
local a;
a:=copy([0,1,2,3,4]);
...
a[3]=<33;
```

Alternatively, you could use a command which recreates a list every time the program is run, such as `makelist` or `$`, instead of copying a list; `a:=makelist(n,n,0,4)` or `a:=[n$(n=0..4)]` can also be used in place of `a:=[0,1,2,3,4]`.

25.5.3 Converting strings to `giac` expressions

Using strings as commands. The `expr` command lets you use a string as a command.

- `expr` takes `str`, a string which expresses a valid command.
- `expr(`*str*`)` converts *str* to the command and evaluates it.

Examples

```
> expr("c:=1"):;
  c
```

$$1$$

```
> a:="ifactor(54)":;
  expr(a)
```

$$2 \cdot 3^3$$

which is the same thing as entering `ifactor(54)` directly.

Converting strings to numbers. You can also use `expr` to convert a string to a number. If a string is simply a number enclosed by quotation marks, then `expr` will return the number.

Examples

The following strings will be converted to the appropriate number.

A string consisting of the digits 0-9 which does not start with 0 will be converted to an integer:

```
> expr("2133")
```

$$2133$$

A string consisting of the digits 0-9 that contains a single decimal point will be converted to a double:

```
> expr("123.4")
```

$$123.4$$

A string consisting of the digits 0-9 and a single decimal point, followed by `e` and then more digits, will be read as a floating point number:

```
> expr("1.23e4")
```

$$12300.0$$

A string consisting of the digits 0-7 which starts with 0 will be read as an integer base 8:

```
> expr("0176")
```

$$126$$

since 176 base 8 equals 126 base 10.

A string starting with `0x` followed by digits 0-9 and letters A-F (or a-f) will be read as an integer base 16:

```
> expr("0x2a3f")
```

10815

since 2A3F base 16 equals 10815 base 10.

A string starting with `0b` followed by digits 0 and 1 will be read as a binary integer:

```
> expr("0b1101")
```

13

since 1101 base 2 equals 13 base 10.

25.5.4 Creating symbols from strings

Variable and function names are symbols, namely sequences of characters, which are different from strings. For example, you can have a variable named `abc`, but not `"abc"`. The `make_symbol` command turns a string into a symbol; for example `make_symbol("abc")` is the symbol `abc`.

Examples

```
> make_symbol("abc"):=3
```

then:

```
> abc
```

3

The variable `abc` will have the value `3`. Similarly for functions:

```
> make_symbol("sin")(pi/4)
```

$$\frac{\sqrt{2}}{2}$$

which is equal to $\sin\frac{\pi}{4}$.

Creating arrays of symbols. The `symbol_array` command is used for creating multidimensional arrays of symbols.

- `symbol_array` takes two mandatory arguments and one optional argument:
  - *str*, a template string for symbol names.
  - $dim = n_1, n_2, \ldots, n_m$, a sequence of m positive integers.
  - Optionally, `purge`, the symbol.
- `symbol_array(`*str*, *dim*$\langle$`,purge`$\rangle$`)` returns an array with m dimensions $n_1, n_2, \ldots, n_m$ with indexed symbols as elements. Symbols are created from the template string *str* by appending indices. Template string may contain m instances of the character `'%'`, kth of which serves as the placeholder for the kth dimension index. If placeholders are omitted, indices are simply appended in case $m = 1$, while in case $m > 1$ they are separated from *str* and each other by the underscore (`'_'`). If `purge` is given, then the already assigned symbols are purged, otherwise a warning is printed for each such symbol.
- The indices are in accordance with the current syntax mode (they are either 0- or 1-based).

Examples

```
> symbol_array("x",5)
```

$$[x_0, x_1, x_2, x_3, x_4]$$

```
> symbol_array("a",2,3)
```

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \end{bmatrix}$$

```
> maple_mode(1):; symbol_array("a%%",2,3)
```

$$[[a11, a12, a13], [a21, a22, a23]]$$

```
> maple_mode(0):; s:=symbol_array("a%b%c%",2,3,4)
```

$$\left[\begin{bmatrix} \text{a0b0c0} & \text{a0b0c1} & \text{a0b0c2} & \text{a0b0c3} \\ \text{a0b1c0} & \text{a0b1c1} & \text{a0b1c2} & \text{a0b1c3} \\ \text{a0b2c0} & \text{a0b2c1} & \text{a0b2c2} & \text{a0b2c3} \end{bmatrix}, \begin{bmatrix} \text{a1b0c0} & \text{a1b0c1} & \text{a1b0c2} & \text{a1b0c3} \\ \text{a1b1c0} & \text{a1b1c1} & \text{a1b1c2} & \text{a1b1c3} \\ \text{a1b2c0} & \text{a1b2c1} & \text{a1b2c2} & \text{a1b2c3} \end{bmatrix}\right]$$

```
> s[1][2][3]
```

a1b2c3

25.5.5 Converting `giac` expressions to strings

The `string` command converts an expression to a string.

- `string` takes *expr*, an expression.
- `string(`*expr*`)` evaluates *expr* then converts it to a string.

Examples

```
> string(ifactor(6))
```

"2*3"

This is the same thing as entering `ifactor(6)+""`, i.e. adding the empty string to the expression.

If you want to convert an unevaluated expression to a string, you can quote the expression (see Section 9.1.4, p. 170):

```
> string(quote(ifactor(6)))
```

"ifactor(6)"

25.5.6 Converting real numbers to strings

The `format` command converts a real number to a string.

- `format` takes two arguments:
  - *r*, a real number.
  - *str*, a string used for formatting.

- `format(`*str*`)` returns *r* as a string with the requested formatting.

The formatting string can be one of the following:

- `f` (for *floating-point* format) followed by the number of digits to put after the decimal point.
- `s` (for *scientific* format) followed by the number of significant digits.
- `e` (for *engineering* format) followed by the number of digits to put after the decimal point, with one digit before the decimal points.

Examples

```
> format(sqrt(2)*10^10,"f13")
```

"14142135623.7308959960938"

```
> format(sqrt(2)*10^10,"s13")
```

"14142135623.73"

```
> format(sqrt(2)*10^10,"e13")
```

"1.4142135623731e+10"

25.5.7 Working with the graphics screen

Showing the graphic screen. Recall that the `DispG` screen contains the graphical output of Xcas. The `DispG` command opens the `DispG` screen.

- `Disp` takes no arguments and no parentheses.
- `Disp` brings up the `Disp` screen.

Clearing the graphic screen. The `ClrGraph` or `ClrDraw` command clears the screen.

- `ClrGraph` takes no arguments.
- `ClrGraph` or `ClrGraph()` clears the `Disp` screen.

Closing the graphic screen. The `DispHome` command closes the `DispG` screen.

- `DispHome` takes no arguments and no parentheses.
- `DispHome` closes the `DispG` screen.

25.5.8 Pausing a program

The `Pause` command pauses Xcas.

- `Pause` takes one optional argument (with no parentheses): *r*, a positive number.
- `Pause` *r* pauses Xcas for *r* seconds.
- `Pause` without an argument brings up a Pause informational window and pauses Xcas until you click `Close` in the Pause window.

The `WAIT` command also pauses Xcas. It acts just like `Pause`, but uses parentheses for its argument.

Example

```
> Pause 10
```

or:

```
> WAIT(10)
```

pauses Xcas for 10 seconds.

25.6 Debugging

25.6.1 Starting the debugger

The `debug` command starts the Xcas debugger.

- `debug` takes *fn(arg)*, a function and its argument.
- `debug(`*fn(arg)*`)` brings up a debug window which contains a pane with the program with the current line highlighted, an `eval` entry box, a pane with the program including the breakpoints, a row of buttons, and a pane keeping track of the values of variables.

By default, the value of all variables in the program are in this pane. The buttons are shortcuts for entering commands in the `eval` box, but you can enter other commands in the `eval` box to change the values of variables or to run a command in the context of the program.

Example

With the `sumprod` program:

```
sumprod(a,b):={
  local s,p;
  s:=a+b;
  p:=a*b;
  return [s,p];
}
```

input:

```
> debug(sumprod(2,3))
```

which opens the debug window shown in Figure 25.2. It has the following buttons:

- sst runs the `sst` command, which takes no arguments and runs the highlighted line in the program before moving to the next line.
- in runs the `sst_in` command, which takes no argument and runs one step in the program or a user defined function used in the program.
- cont runs the `cont` command, which takes no arguments and runs the commands from the highlighted line to a breakpoint.
- kill runs the `kill` command, which exits the debugger.
- break puts the command `breakpoint` in the `eval` box, with default arguments of the current program and the current line. It sets a breakpoint at the given line of the given program. Alternatively, if you click on a line in the program in the top pane, you will get the `breakpoint` command with that program and the line you clicked on.

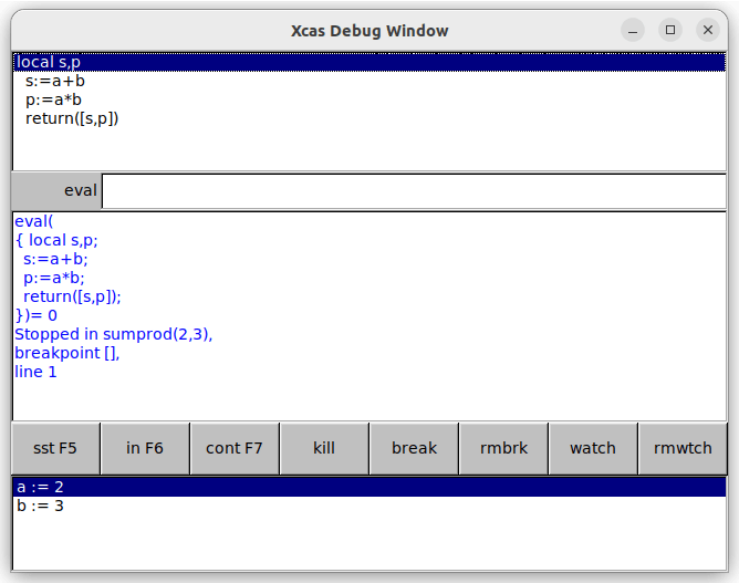


Figure 25.2: The debug window in Xcas

You can set a breakpoint when you write a program with the `halt` command. A `halt()` line in the program will bring up the debugger during runtime. If you want to debug the program, though, it is still better to use the debug command. Also, you should remove any `halt` commands when you are done debugging.

- rmbrk puts the command `rmbreakpoint` in the `eval` box, with default arguments of the current program and the current line. It removes a breakpoint at the given line of the given program. Alternatively, you can click on the line in the program in the top pane with the bookmark you want to remove.
- watch puts the command `watch` in the `eval` box, without the arguments filled in. It takes a list of variables as arguments, and will keep track of the values of these variables in the variable pane.
- rmwtch puts the command `rmwatch` in the `eval` box without the arguments filled in. The arguments are the variables you want to remove from the watch list.

25.7 Linking to and extending the `giac` library

25.7.1 Using `giac` inside a C++ program

To use `giac` inside of a C++ program, put

```
#include <giac/giac.h>
```

at the beginning of the file. To compile the file, use

```
c++ -g progname.cc -lgiac -lgmp
```

After compiling, there will be a file `a.out` which can be run with the command

```
./a.out
```

As an example, put the following program in a file named `pgcd.cc`.

```
#include <giac/config.h>
#include <giac/giac.h>
```

```
using namespace std;
using namespace giac;

gen pgcd(gen a,gen b) {
  gen q,r;
  for (;b!=0;) {
    r=irem(a,b,q);
    a=b; b=r;
  }
  return a;
}

int main() {
  cout << "Enter 2 integers ";
  gen a,b;
  cin >> a >> b;
  cout << pgcd(a,b) << endl;
  return 0;
}
```

After compiling this with

```
c++ -g pgcd.cc -lgiac -lgmp
```

and running it with

```
./a.out
```

a prompt will appear:

```
Enter 2 integers
```

After entering two integers, such as

```
Enter 2 integers  30 36
```

the result will appear:

```
6
```

25.7.2 Defining new `giac` functions

New `giac` functions can be defined with a C++ program. All data in the program used in formal calculations needs to be `gen` type. A variable `g` can be declared to be `gen` type with

```
gen g;
```

In this case, `g.type` can have different values. The value of `g` is fetched differently for different types:

- If `g.type` is `_INT_`, then `g.val` is an integer type `int`.
- If `g.type` is `_DOUBLE_`, then `g._DOUBLE_val` is a real `double`.
- If `g.type` is `_SYMB`, then `g._SYMBptr` points to a value of type `symbolic`. The function is contained in the `sommet` member and its argument(s) in the `feuille` member.
- If `g.type` is `_VECT`, then `g._VECTptr` points to a value of type `vecteur`, which behaves like `std::vector`.
- If `g.type` is `_ZINT`, then `g._ZINTptr` points to a value of integer type `zint`.
- If `g.type` is `_IDNT`, then `g._IDNTptr` points to an identifier of type `identificateur`.

- If g.type will be _CPLX, then g._CPLXptr and g._CPLXptr+1 are pointers to the real and imaginary parts of g.

As an example, put the following program in a file called pgcd.cpp.

```
#include <stdexcept>
#include <cmath>
#include <cstdlib>
#include <giac/config.h>
#include <giac/giac.h>

using namespace std;

#ifndef NO_NAMESPACE_GIAC
namespace giac {
#endif

  gen monpgcd(const gen & a0,const gen & b0) {
    gen q,r,a=a0,b=b0;
    for (;b!=0;) {
      r=irem(a,b,q);
      a=b; b=r;
    }
    return a;
  }

  gen _monpgcd(const gen & args,GIAC_CONTEXT) {
    if ((args.type!=_VECT) || (args._VECTptr->size()!=2))
    return gensizeerr(contextptr);  // return an error
    vecteur &v=*args._VECTptr;
    return monpgcd(v[0],v[1]);
  }

  const string _monpgcd_s[]="monpgcd";
  unary_function_eval __monpgcd(0,&_monpgcd,_monpgcd_s);
  unary_function_ptr at_monpgcd(&__monpgcd,0,true);

#ifndef NO_NAMESPACE_GIAC
}
#endif
```

After compiling this with the commands with

```
g++ -I.. -fPIC -DPIC -g -c pgcd.cpp -o pgcd.lo && ln -sf pgcd.lo pgcd.o && \
  gcc -shared pgcd.lo -lc -lgiac -Wl,-soname -Wl,libpgcd.so.0 -o \
  libpgcd.so.0.0.0 && ln -sf libpgcd.so.0.0.0 libpgcd.so.0 && \
  ln -sf libpgcd.so.0.0.0 libpgcd.so
```

the new command can be inserted with the insmod command in giac, where insmod takes the full absolute path of the libpgcd.so file as argument.

```
> insmod("/path/to/file/libpgcd.so")
```

Afterwards, the monpgcd command will be another giac command. For example, input:

```
> monpgcd(30,36)
```

6

26 2D graphics

26.1 Introduction

26.1.1 Points, vectors and complex numbers

A point in the Cartesian plane is described with an ordered pair (a, b). It has x-coordinate (abscissa) a and y-coordinate (ordinate) b.

A vector from one point (a_1, b_1) to another (a_2, b_2) has associated ordered pair $(a_2 - a_1, b_2 - b_1)$; so the abscissa is $a_2 - a_1$ and the ordinate is $b_2 - b_1$.

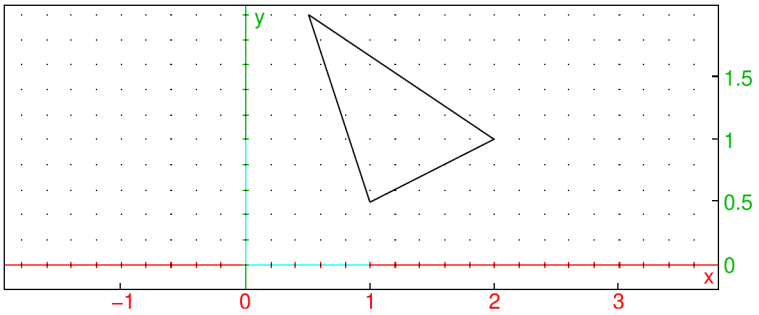
A complex number $a + \mathrm{i}b$ can be associated with the point (a, b) in the Cartesian plane. The complex number is called the *affix* of the point.

A point in Xcas is specified with the `point` command (see Section 26.5.2, p. 685), which takes as argument either two real numbers a, b or a complex number $a + \mathrm{i}b$. In this chapter, when a command take a point as an argument, the point can either be the result of the `point` command or simply a complex number.

An interactive graphic screen opens whenever a geometric object is drawn, or with the command Alt+G. The objects on the screen can also be created and manipulated with the mouse.

As an example (to be explained in more detail later), the `triangle` command draws a triangle; the result will be a graphics screen containing axes, the triangle and a control panel on the right.

```
> triangle(1+i/2,2+i,1/2+2i)
```



26.1.2 Clearing the `DispG` screen

The `DispG` screen records all graphic commands since the beginning of the session or the screen was last erased. The Alt+D command (or the menu command Cfg ▶ Show ▶ DispG) brings up this screen.

The `erase` command clears the `DispG` screen without restarting the session. The commandline `erase` or `erase()` clears the `DispG` screen. This can be useful for commands such as `graph2tex`, which only takes into account the objects on the `DispG` screen.

26.1.3 Toggling the axes

The `switch_axes` command shows, hides or toggles the coordinate axes on the graphics screen. This can also be controlled by a Show axes checkbox in the configuration panel brought up with the cfg button on the graphic screen control panel.

- `switch_axes` takes n, either 0 or 1.

- `switch_axes()` toggles whether or not the coordinate axes are show in subsequent graphics screens.
- `switch_axes(0)` causes all later graphic screens to omit the axes.
- `switch_axes(1)` causes all later graphic screens to have the axes.

When the axes are visible, they have tick marks whose separation is determined by the X-tick and Y-tick values on the graphic configuration screen. Setting these values to 0 also removes the axes.

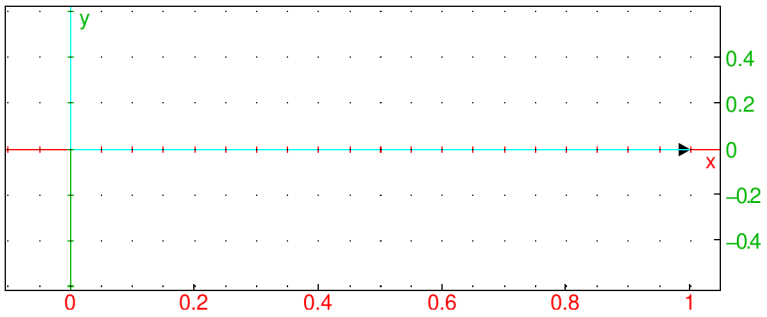
26.2 Basic commands

26.2.1 Drawing unit vectors in the plane

The `Ox_2d_unit_vector` command takes no arguments and draws the unit vector in the x-direction on a plane.

Example

```
> Ox_2d_unit_vector()
```



Similarly, the `Oy_2d_unit_vector` command draws the unit vector in the y direction. The `frame_2d` command simultaneously draws both unit vectors.

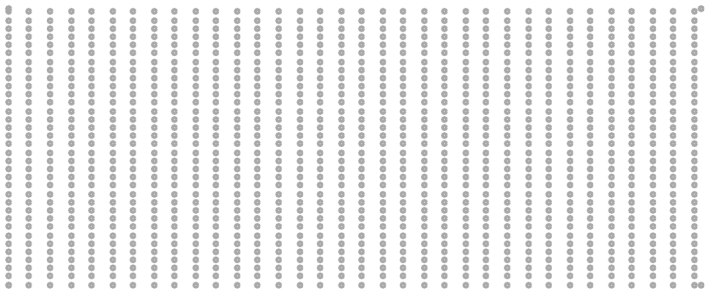
26.2.2 Drawing dotted paper

The `dot_paper` command draws dotted paper.

- `dot_paper` takes three mandatory arguments and two optional arguments.
  - *xspacing*, the spacing in the x direction.
  - θ, the angle from the horizontal to draw the dots.
  - *yspacing*, the spacing in the y direction.
  - Optionally, `x=`$x_{min}..x_{max}$, to determine how far the dots extend in the x direction (by default, the distances given in the graphic configuration page accessible from the main menu).
  - Optionally, `y=`$y_{min}..y_{max}$, to determine how far the dots extend in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `dot_paper(`*xspacing*$,\theta,$ *yspacing* $\langle$`, x=`$x_{min}..x_{max}$`, y=`$y_{min}..y_{max}\rangle$`)` draws dotted paper.

Example

```
> axes=0; dot_paper(0.6,pi/2,0.6,display=grey)
```



26.2.3 Drawing lined paper

The `line_paper` command draws lined paper.

- `line_paper` takes two mandatory arguments and two optional arguments.
  - *xspacing*, the spacing in the x direction.
  - θ, the angle from the horizontal to draw the lines.
  - Optionally, `x=`$x_{min}..x_{max}$, to determine how far the lines extend in the x direction (by default, the distances given in the graphic configuration page accessible from the main menu).
  - Optionally, `y=`$y_{min}..y_{max}$, to determine how far the lines extend in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `line_paper(`*xspacing*, θ $\langle$, `x=`$x_{min}..x_{max}$, `y=`$y_{min}..y_{max}$ $\rangle$`)` draws lined paper.

Example

```
> axes=0; line_paper(0.6,pi/3,display=grey)
```

26.2.4 Drawing grid paper

The `grid_paper` command draws grid paper.

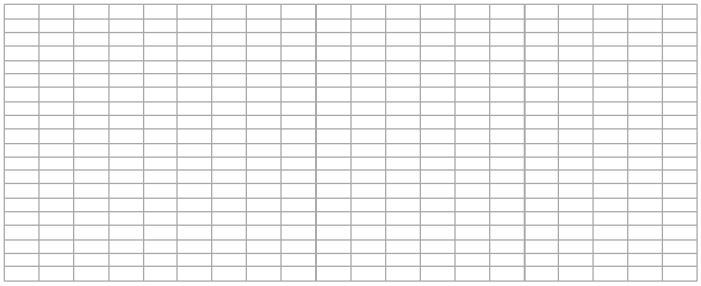
- `grid_paper` takes three mandatory arguments and two optional arguments.
  - *xspacing*, the spacing in the x direction.
  - θ, the angle from the horizontal to draw the grid.
  - *yspacing*, the spacing in the y direction.

  - Optionally, `x=`$x_{min}..x_{max}$, to determine how far the grid extends in the x direction (by default, the distances given in the graphic configuration page accessible from the main menu).
  - Optionally, `y=`$y_{min}..y_{max}$, to determine how far the grid extends in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).

- `grid_paper(`*xspacing*, θ, *yspacing* $\langle$, `x=`$x_{min}..x_{max}$, `y=`$y_{min}..y_{max}$ $\rangle$`)` draws grid paper.

Example

```
> axes=0; grid_paper(1,pi/2,1,display=grey)
```



26.2.5 Drawing triangular paper

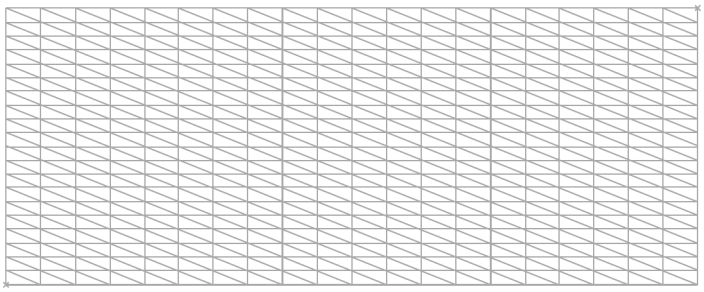
The `triangle_paper` command draws triangular paper.

- `triangle_paper` takes three mandatory arguments and two optional arguments.
  - *xspacing*, the spacing in the x direction.
  - θ, the angle from the horizontal.
  - *yspacing*, the spacing in the y direction.
  - Optionally, `x=`$x_{min}..x_{max}$, to determine how far the grid extends in the x direction (by default, the distances given in the graphic configuration page accessible from the main menu).
  - Optionally, `y=`$y_{min}..y_{max}$, to determine how far the grid extends in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).

- `triangle_paper(`*xspacing*, θ, *yspacing* $\langle$, `x=`$x_{min}..x_{max}$, `y=`$y_{min}..y_{max}$ $\rangle$`)` draws triangle paper.

Example

```
> axes=0; triangle_paper(1,pi/2,1,display=grey)
```



26.3 Display features of graphics

26.3.1 Graphic features

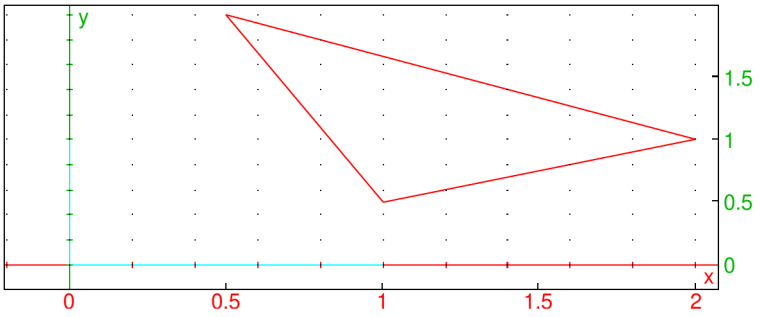
Graphic objects and graphic screens can have features, such as labels and colors, that are only included when requested, and other features, such as line width, which are configurable. Some features will be global, meaning that they will apply to the entire graphic screen, and some will be local, meaning that they will only apply to individual objects.

26.3.2 Parameters for changing features

Graphical features are changed by giving appropriate values to certain parameters. Several values can be given at once with an expression of the form *feature*`=value1+value2+...+valueN`. Some values can be set using optional arguments to graphic commands, which will set the feature locally; namely, it will only apply to that particular graphic object. Some values can be specified at the beginning of a line, which will set the feature globally; it will apply to all the graphic objects created on that line. For some features, both options are available.

Parameters for local features. Commands which create graphic objects, such as `triangle`, can have optional arguments to change a features of the object. For example, the argument `color=red` will make an object red.

```
> triangle(1+i/2,2+i,1/2+2i,color=red)
```



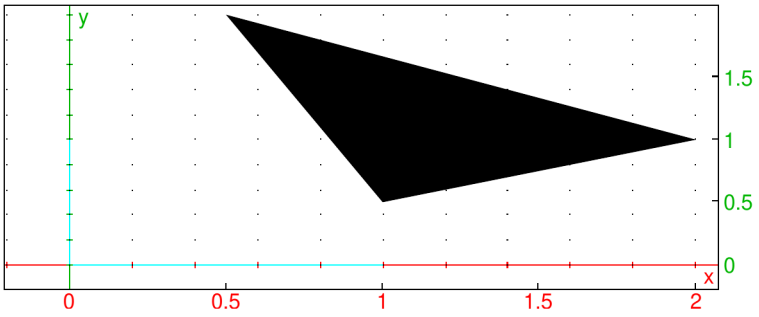
The features and their possible values are:

- `display` or `color` — These two parameter names have the same effect. See Section 19.1.2, p. 455 for information on supported values.
- `thickness` — This controls line thickness, it can be an integer from 1 to 7.
- `nstep` — The number of sampling points for 3D objects.
- `tstep` — The step size of the parameter when drawing a one parameter parametric plot.
- `ustep` — The step size of the first parameter when drawing a two-parameter parametric plot.
- `vstep` — The step size of the second parameter when drawing a two-parameter parametric plot.
- `xstep` — The step size of the x variable.
- `ystep` — The step size of the y variable.
- `zstep` — The step size of the z variable.
- `frames` — The number of graphs computed when an animated graph is created with the `animate` or `animate3d` command.

- `legend` — This adds a legend to a graphic object and should be a string. It is probably most useful when that object is a point or a polygon. If the object is a polygon, the legend will be placed in the middle of the last side. Other parameters for the graphic object will specify the color or position of the legend.
- `gl_texture` — This sets an image file to be put on the graphic object; it should be the name of the file.

Examples (of the `filled` option)

```
> triangle(1+i/2,2+i,1/2+2i,display=filled)
```

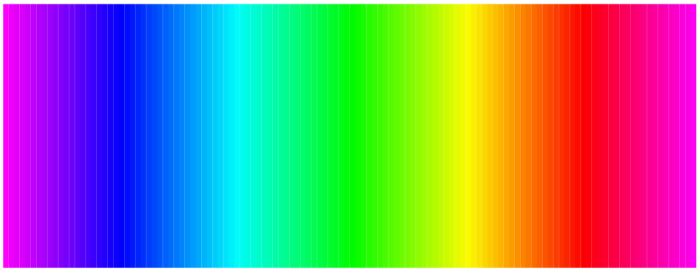


To see the rainbow colors, you can enter and compile the program:

```
rainbow():={
  local j,C;
  C:=[];
  for (j:=256;j<382;j++) {
    C:=append(C,square(j,j+1,color=j+filled));
  }
}
```

followed by

```
> axes=false; rainbow();
```

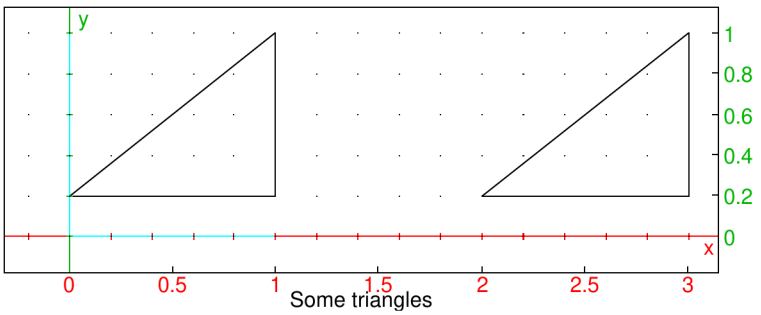


The number of a color is its x-coordinate. To see just one color, say the color corresponding to n for $256 \leq n \leq 381$, enter: `rainbow()[`n`-256]`.

Parameters for global features. Parameters set at the beginning of a line change features on the entire graphic screen. It only takes effect when the line ends with a graphic command. For example, starting the line with `title=`*titlestring* will give the graphic screen a title.

Example

```
> title="Some triangles"; triangle(0+i/5,1+i/5,1+i); triangle(2+i/5,3+i/5,3+i);
```



The parameters for global features and their possible values are:

- `axes` — This determines whether axes are shown or hidden; a value of `0` or `false` hides the axes, a value of `1` or `true` shows the axes.
- `labels` — This sets labels for the axes, it should be a list of two strings [“x axis label”, “y axis label”].
- `label` — This puts labels on the graphic screen in the following ways.
  - To set the units on the axes, it can be a list of two or three strings, [“x units”, “y units”] or [“x units”, “y units”, “z units”].
  - To place a string at a particular point, it can be a list of two integers followed by a string. The integers determine the point, starting from $(0, 0)$ in the top left of the screen.
- `title` — This sets the title for the graphic window, it should be a string.
- `gl_texture` — This sets the wallpaper of the graphic window to be an image file, it should be the name of the file.
- `gl_x_axis_name`, `gl_y_axis_name`, `gl_z_axis_name` — These set the names of the axes.
- `gl_x_axis_unit`, `gl_y_axis_unit`, `gl_z_axis_unit` — These set the units of the axes.
- `gl_x_axis_color`, `gl_y_axis_color`, `gl_z_axis_color` — These set the colors of the axes labels; they take the same color options as the local parameter `color`.
- `gl_ortho` — This ensures that the graph is orthonormal when it is set to `1`.
- `gl_x`, `gl_y`, `gl_z` — These define the framing of the graph; they should be ranges $min..max$. (They are not compatible with interactive graphs.)
- `gl_xtick`, `gl_ytick`, `gl_ztick` — These determine the spacing of the ticks on the axes.
- `gl_shownames` — This shows or hides object names, it can be `true` or `false`.
- `gl_rotation` — This sets the axis of rotation for 3D scene animations; it should be a direction vector $[x, y, z]$.
- `gl_quaternion` — This sets the quaternion for viewing 3D scenes; it should be a fourtuple $[x, y, z, t]$. (This is not compatible with interactive graphs.)

26.3.3 Commands for global display features

Adding a legend. The `legend` command creates a legend on the screen.

- `legend` takes two mandatory arguments and one optional argument:

  - *pos*, either be a point or a list of two integers giving the number of pixels from the upper left hand corner, specifying the position to put the legend.
  - *legend*, a string or a variable.
  - Optionally, *quad*, which can be one of `quadrant1`, `quadrant2`, `quadrant3` or `quadrant4`. This indicates where to put the legend relative to the point (by default, it is `quadrant1`).

- `legend(`*pos*, *legend*⟨, *quad*⟩`)` draws the legend at the requested position.

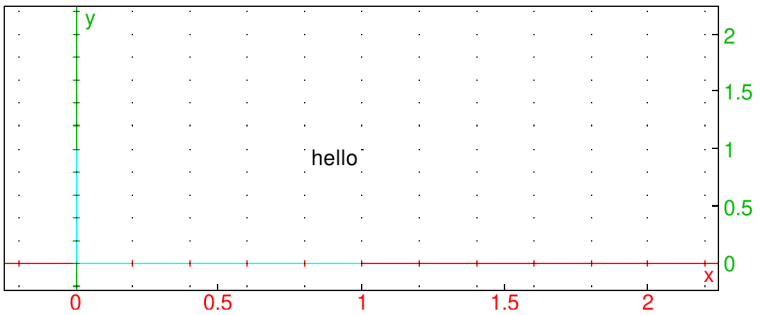
Example

To put "hello" to the upper left of the point $(1, 1)$:

```
> legend(1+i,"hello",quadrant3)
```

or:

```
> legend(1+i,quadrant3,"hello")
```



Changing various features. The `display` or `color` command changes the properties of graphics; the same properties that can also be changed with the `display` and `color` parameters (see Section 26.3.2, p. 678).

The `display` or `color` command draws objects with specified properties.

- `display` takes one mandatory arguments and one optional argument:
  - Optionally, *command*, a command to draw an object.
  - *arg*, which can be a possible value of the `display` parameter (see Section 26.3.2, p. 678) or `hidden_name`.

- `display(`*command*, *arg*`)` draws the object given by *command* with the property given by *arg*, or draws the object without a label if *arg*=`hidden_name`.

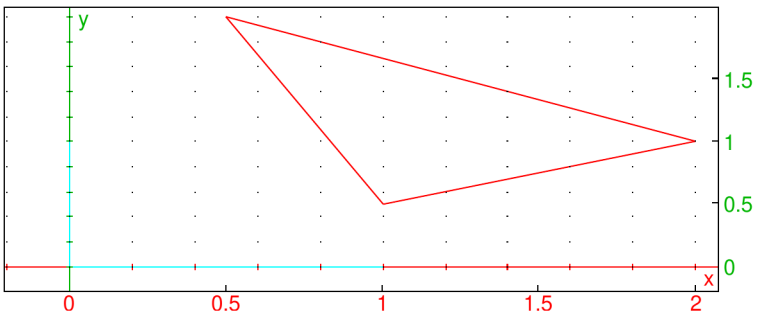
- `display(`*arg*`)` applies the property given by *arg* to all subsequent objects; `display(0)` resets the display parameters.

Examples

```
> display(1+i/2,2+i,1/2+2i,red)
```

or:

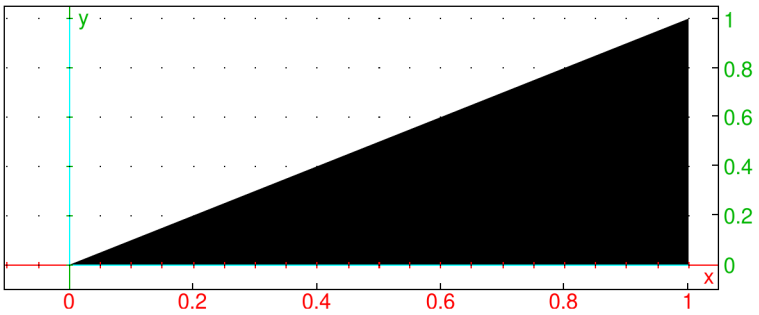
```
> triangle(1+i/2,2+i,1/2+2i,display=red)
```




```
> triangle(0,1,1+i,display=filled)
```

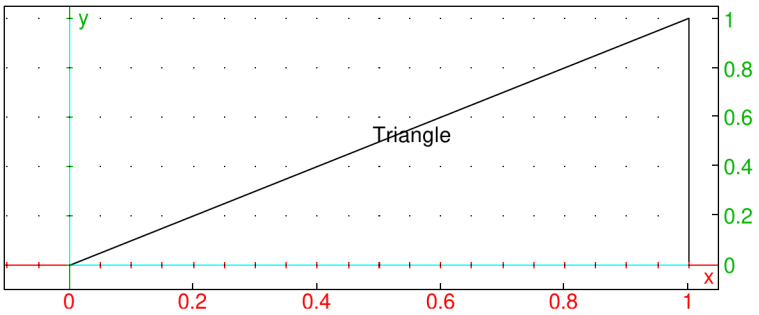
or:

```
> display(triangle(0,1,1+i),filled)
```



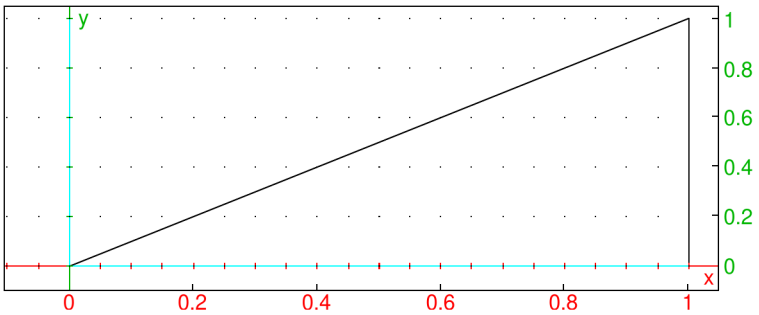
By default, if a geometric object is named, the drawing is labeled.

```
> Triangle:=triangle(0,1,1+i)
```



Creating the object with the `display` command and the `hidden_name` argument will draw it without the label.

```
> display(Triangle,hidden_name)
```



26.3.4 Defining geometric objects without drawing them

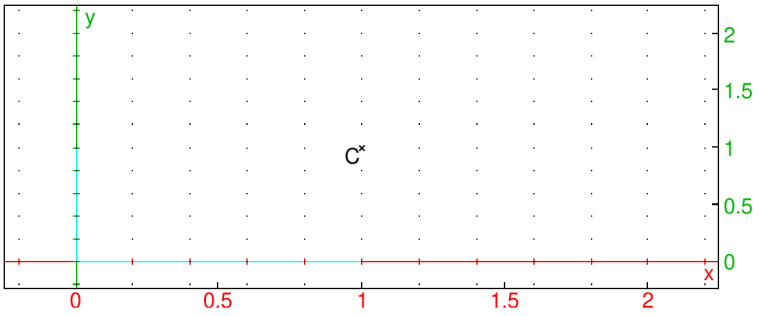
The `nodisp` command defines an object without displaying it.

- `nodisp` takes *command*, a command to create an object.
- `nodisp(`*command*`)` creates the object withouth drawing it.

Setting a variable to a graphic object draws the object.

Examples

```
> C:=point(1+i)
```



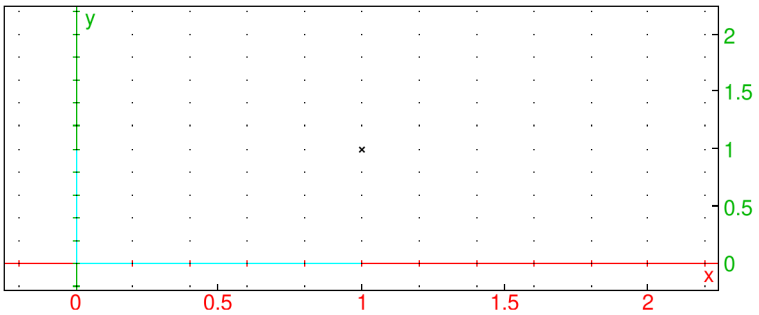

```
> nodisp(C:=point(1+i))
```

Here, the point C is defined but not displayed. It is equivalent to following the command with `:;`,

```
> C:=point(1+i):;
```

To define a point as above and display it without the label, enter the point's name;

```
> C
```



Alternatively, you can get the same effect by defining the point within an `eval` statement:

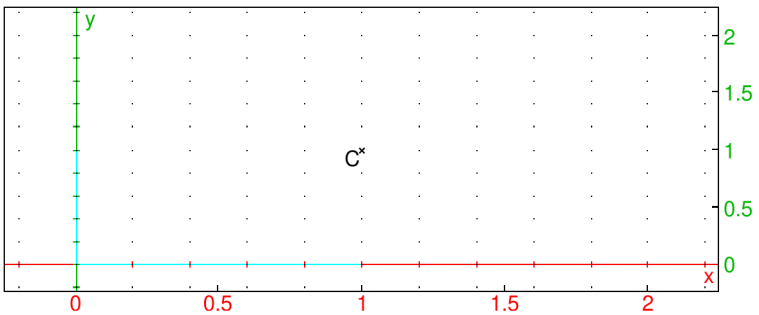
```
> eval(C:=point(1+i))
```

To later display the point with a label, use the `legend` command:

```
> legend(C,"C")
```

or:

```
> point(affix(C),legend="C")
```



In this case, the string "C" can be replaced with any other string as a label. Alternatively, redefine the variable as itself:

```
> C:=C
```

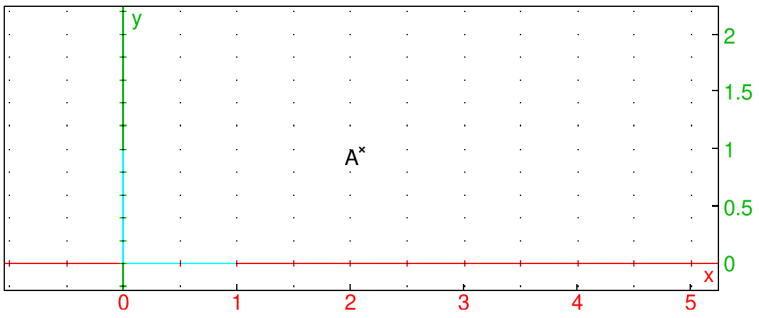
prints C with its label.

26.4 Geometric demonstrations with sliding parameters

Variables should be unspecified to demonstrate a general geometric result, but need to have specific values when drawing. There are a couple of different approaches to deal with this.

One approach is to use the `assume` command (see Section 3.3.8, p. 39). If a variable is *assumed* to have a value, then that value will be used in graphics but the variable will still be unspecified for calculations. For example:

```
> assume(a=2.1):;
  A:=point(a+i)
```



but the variable a will still be treated as a variable in calculations:

```
> distance(0,A)
```

$$\sqrt{(-a)^2+1}$$

Another approach would be to use the `point` or `pointer` mode in a geometry screen. If no geometry screen is shown, the command Alt+G or the Geo ▶ New figure 2d menu will open a screen. Clicking on the Mode button right above the graphic screen and choosing `pointer` or `point` will put the screen in `pointer` or `point` mode. If a point is defined and displayed, such as with `A:=point(2.1+i)`, then clicking on the name of the point (`A` in this case) with the right mouse button will bring up a configuration screen. As long as there is a point defined with non-symbolic values, there will be a `symb` box on the configuration screen. Selecting the `symb` box and choosing `OK` will be equivalent to the commands:

```
assume(Ax=[2.1,-8.16901408451,8.16901408451]);
assume(Ay=[1,-5.0,5.0]
```

This brings up two lines beneath the arrows to the right of the screen which can be used to change the assumed values of `Ax` and `Ay`. Also, the point `A` will be redefined as `point(Ax,Ay)` (see Figure 26.1).

26.5 Points in the plane

26.5.1 Points and complex numbers

The *affix* of a point (a, b) in the plane is the complex number $a + bi$. In this section, when a command take points as arguments, the points can be specified by a pair or by a complex number.

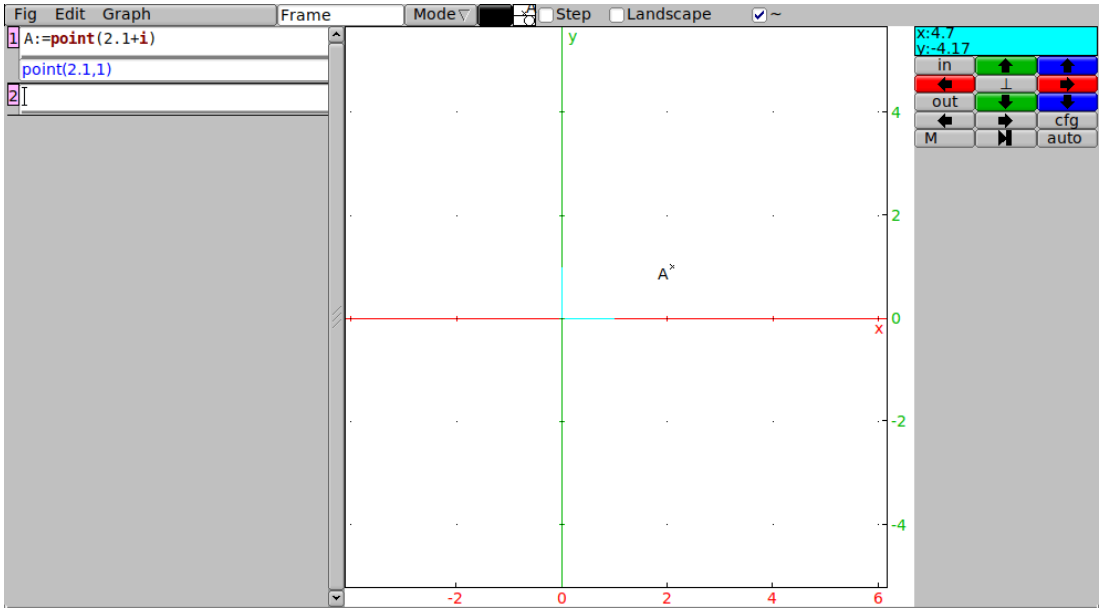


Figure 26.1: A 2D-geometry screen in Xcas

26.5.2 Point in the plane

See Section 27.3.1, p. 756 for points in space.

In the 2D geometry screen in point mode, clicking on a point with the left mouse button will choose that point. Points chosen this way are automatically named, first with `A`, then `B`, etc.

Alternatively, the `point` command chooses a point.

- `point` takes one or two arguments: *coords*, where *coords* can be one of:
  - a, b, a sequence of two coordinates.
  - $[a, b]$, a list of two coordinates.
  - $a + b\mathrm{i}$, the affix of the point.
- `point(`*coords*`)` returns and draws the point with the given coordinates.

Example

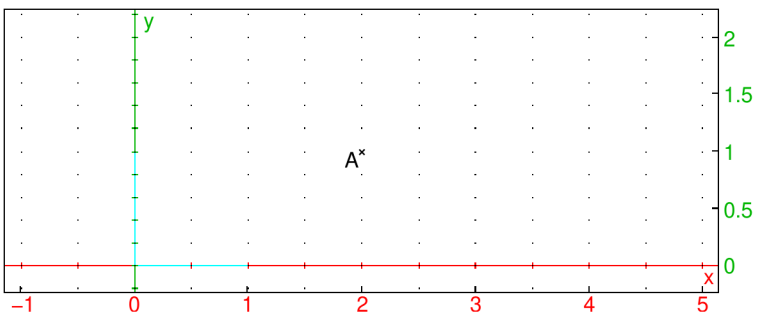
```
> A:=point(2,1)
```

or:

```
> A:=point([2,1])
```

or:

```
> A:=point(2+i)
```



The marker used to indicate the point can be changed; see Section 26.3.2, p. 678.

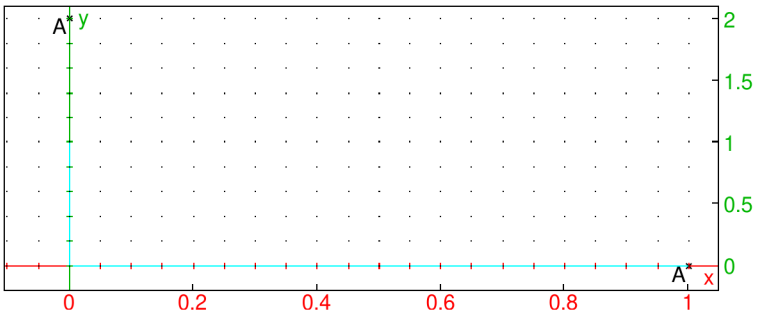
If the `point` command has two numbers for arguments, at least one of which is complex but not real, then it will choose two points.

Example

```
> A:=point(1,2*i)
```

or:

```
> A:=point([1,2*i])
```



There are two points named `A`; one with affix 1 and one with affix `2i`.

26.5.3 Difference and sum of two points in the plane

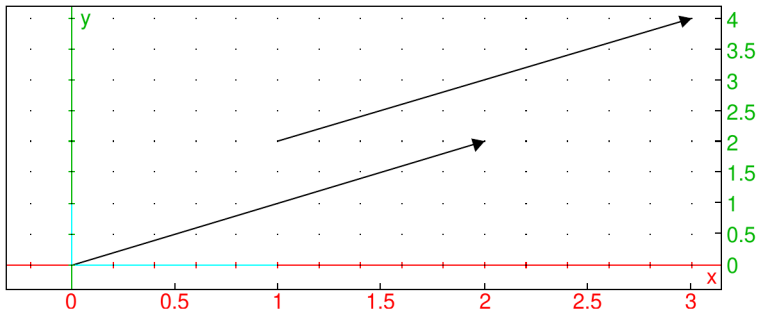
Let A and B be two points in the plane, with affixes $a_1 + \mathrm{i}a_2$ and $b_1 + \mathrm{i}b_2$ respectively.

```
> A:=point(1+2*i):; B:=point(3+4*i):;
```

Then:

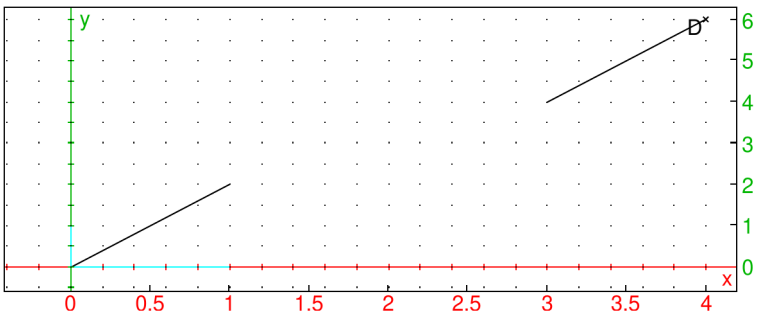
- The difference $B - A$ returns the affix $(b_1 - a_1) + \mathrm{i}(b_2 - a_2)$, which represents the vector AB.

  ```
  > vector(A,B); vector(point(0),point(B-A))
  ```



- The sum $B + A$ returns the affix $(b_1 + a_1) + \mathrm{i}(b_2 + a_2)$. If `D:=point(B+A)`, then `BD=OA`.

  ```
  > D:=point(B+A);
    segment(B,D); segment(point(0),A)
  ```



Note that $-A$ is the point symmetrical to A with respect to the origin.

The sum of three points $A+B+C$ can be viewed as the translate of C by the vector $A+B$. So if A or B contains parameters, you should write this as $C+(A+B)$ or `evalc`$(A+B)+C$.

26.5.4 Defining random points in the plane

The `point2d` command defines a random point whose coordinates are integers between -5 and 5.

- `point2d` takes *names*, a sequence of names for the points.
- `point2d(`*names*`)` assigns a random point whose coordinates are integers between -5 and 5 to each name.

Examples

Assign `A` to a random point (once assigned, the point is fixed):

```
> point2d(A)
```

Generate three random points and uses them to create a triangle:

```
> point2d(A,B,C);
  triangle(A,B,C)
```

In other words, this generates a random triangle.

26.5.5 Points in polar coordinates

The `point` command can be used for specifying a point in polar coordinates by using the polar representation of complex numbers, e.g.

```
> point(2*exp(i*pi/4))
```

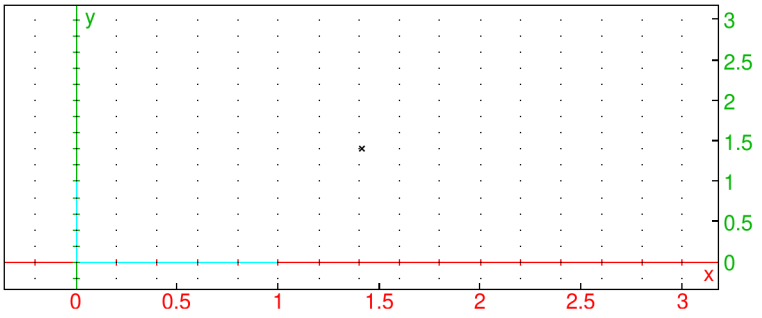
which is the point with polar coordinates $r=2$, $\theta=\pi/4$.

The `polar_point` or `point_polar` command is an easier way to specify a point in polar coordinates.

- `polar_point` takes two arguments:
  - r, a number.
  - θ, a number.
- `polar_plot`(r,θ) returns and draws the point with polar coordinates r,θ.

Example

```
> polar_point(2,pi/4)
```



which is the same point as before.

26.5.6 Finding a point of intersection of two objects in the plane

See Section 27.3.3, p. 757 for single points of intersection of objects in space.

The `single_inter` or `line_inter` command finds an intersection point of two geometric objects.

- `single_inter` takes two mandatory arguments and one optional argument.
  - obj_1, obj_2, two geometric objects.
  - Optionally, pt, a point or list of points.

  `line_inter(`obj_1, obj_2 ⟨, pt⟩`)` returns one of the points of intersection of obj_1 and obj_2.

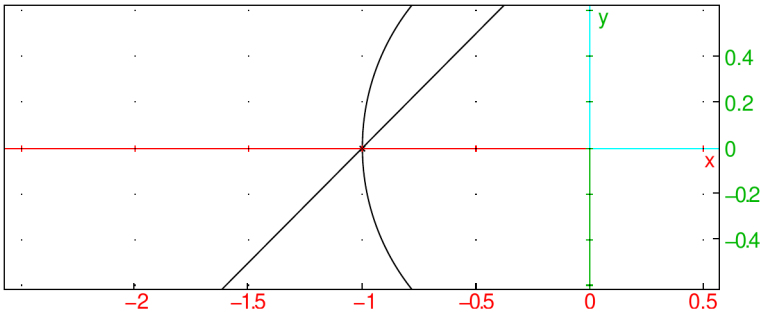
  If pt is a single point, then the command returns the point of intersection closest to pt.

  If pt is a list of points, then the command tries to return a point not in pt.

Example

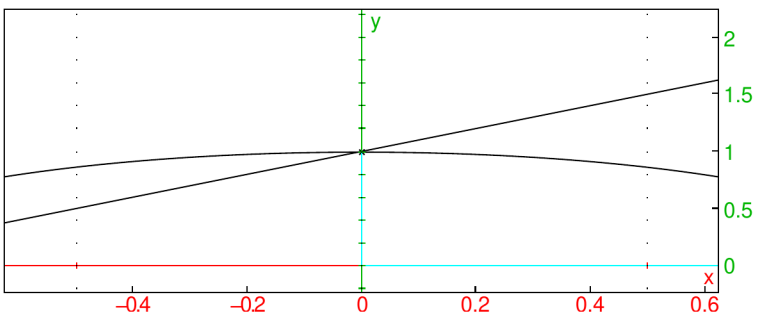
The command `circle(0,1)` creates the unit circle and `line(-1,i)` creates a line, these two objects intersect at the points $(-1, 0)$ and $(0, 1)$.

```
> circle(0,1),line(-1,i),single_inter(circle(0,1),line(-1,i))
```



which is the point $(-1, 0)$.

```
> circle(0,1),line(-1,i),single_inter(circle(0,1),line(-1,i),[-1])
```



which is the point $(0, 1)$. Similarly, since this second point of intersection is closest to $(0, 1/2)$, entering:

```
> single_inter(circle(0,1),line(-1,i),i/2)
```

also draws the second point.

26.5.7 Finding the points of intersection of two geometric objects in the plane

See Section 27.3.4, p. 758 for points of intersection of objects in space.

The `inter` command finds the intersection of two geometric objects in the plane.

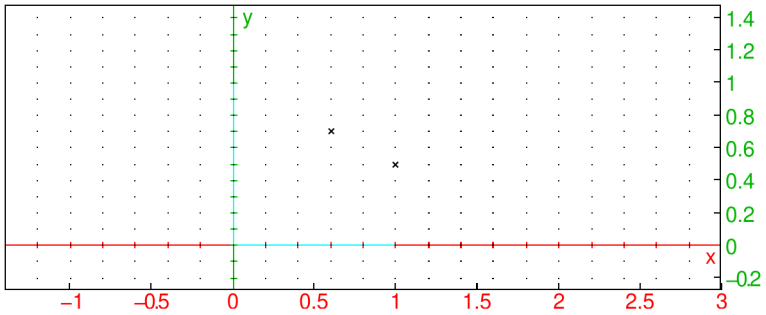
- `inter` takes two mandatory arguments and one optional argument.
  - obj_1, obj_2, two geometric objects.
  - Optionally, P, a point.

  `inter(`obj_1`,`obj_2 $\langle$`,`$P\rangle$`)` returns a list of points of intersection of obj_1 and obj_2.

  With the argument P, the command returns the point of intersection closest to P.

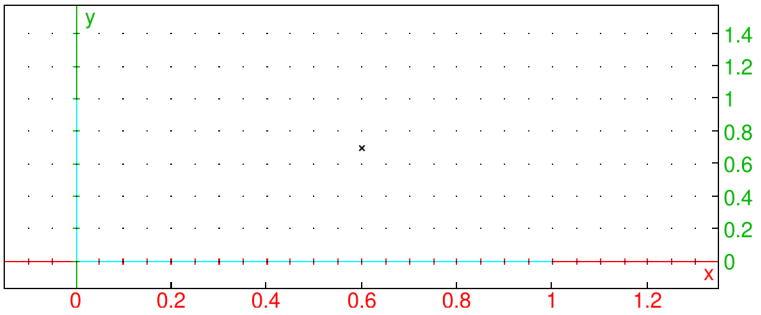
Examples

```
> inter(circle(1+i,1/2),line(2,i))
```



which are the points at $(1, 0)$ and $(0, 1)$. To get just one of the points, use the usual list indices.To get the point closest to e.g. $(0, 1/2)$:

```
> inter(circle(1+i,1/2),line(2,i),i/2)
```



26.5.8 Finding the orthocenter of a triangle in the plane

The `orthocenter` command finds the orthocenter of a triangle.

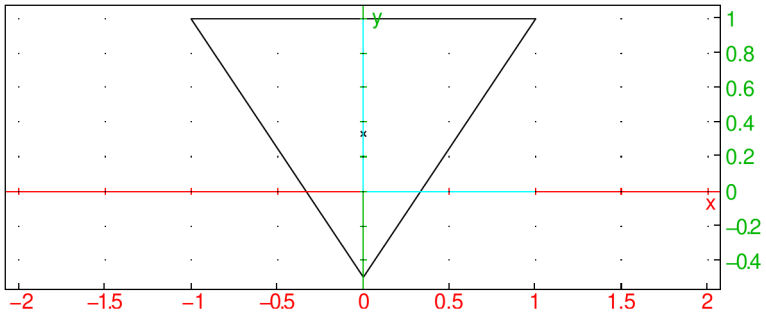
- `orthocenter` takes T, a triangle. The triange can also be specified with three points.
- `orthocenter(`T`)` returns the orthocenter of T.

Example

```
> triangle(-i/2,1+i,-1+i), orthocenter(triangle(-i/2,1+i,-1+i))
```

or:

```
> triangle(-i/2,1+i,-1+i), orthocenter(-i/2,1+i,-1+i)
```



which is the point $(0, 0)$, the orthocenter of the triangle.

26.5.9 Finding the midpoint of a segment in the plane

See Section 27.3.5, p. 759 for midpoints in space.

The `midpoint` command finds the midpoint of two points.

- `midpoint` takes two arguments: P, Q, two points (which can also be given as a list).
- `midpoint`(P, Q) draws and returns the midpoint of the segment determined by these points.

26.5.10 Barycenter in the plane

See Section 27.3.6, p. 759 for barycenters of objects in space.

The `barycenter` command returns and draws the barycenter of a set of weighted points.

- `barycenter` takes $L_1, L_2, \ldots, L_n$, a sequence of lists of length two, where each list consists of a point and a weight. This information can also be given as a matrix with two columns (the first column the points and the second column the weights) or a matrix with two rows and more than two columns.
- `barycenter`$(L_1, L_2, \ldots, L_n)$ draws and returns the barycenter of the weighted points.

Example

The following commands will draw the barycenter of the points $(1, 1)$ with weight 1, $(1, -1)$ with weight 1 and $(1, 4)$ with weight 2.

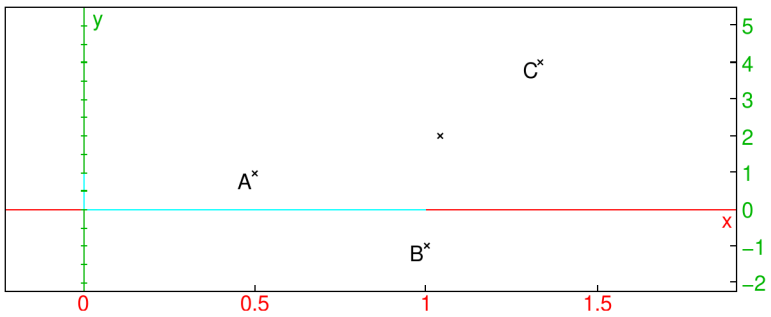
```
> A:=point(1/2+i); B:=point(1-i); C:=point(4/3+4*i);
  barycenter([A,1],[B,1],[C,2])
```

or with the alternative second line:

```
> barycenter([[A,1],[B,1],[C,2]])
```

or:

```
> barycenter([[A,B,C],[1,1,2]])
```



26.5.11 Isobarycenter of n points in the plane

See Section 27.3.7, p. 759 for isobarycenters of objects in space.

The `isobarycenter` command finds the isobarycenter of a list of points; the isobarycenter is the barycenter when all points are equally weighted.

- `isobarycenter` takes L, a list of points. (The points can also be given by a sequence).
- `isobarycenter(`L`)` draws and returns the isobarycenter of the points.

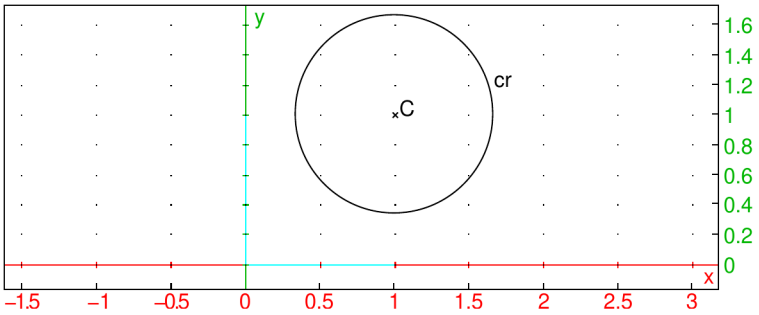
26.5.12 Center of a circle in the plane

The `center` command finds the center of a circle.

- `center` takes C, a circle.
- `center(`C`)` draws and returns the center of C.

Example

```
> cr:=circle(point(1+i),2/3); C:=center(cr)
```



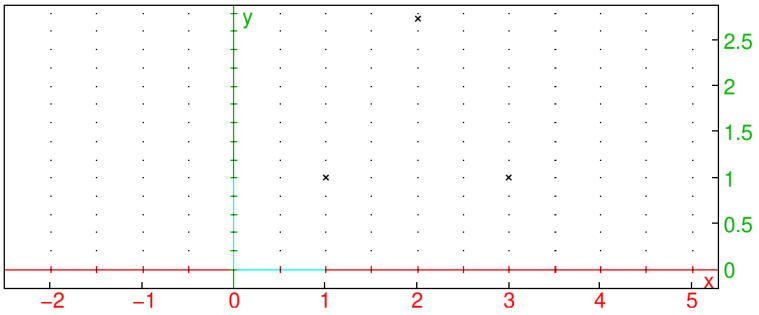
26.5.13 Vertices of a polygon in the plane

The `vertices` or `vertices_abc` command finds the vertices of a polygon.

- `vertices` takes P, a polygon.
- `vertices(`P`)` returns a list of the vertices of P and draws them.

Examples

```
> vertices(equilateral_triangle(1+i,3+i))
```



Individual vertices can be fetched by using the `[]` operator, like e.g.

```
> C:=vertices(equilateral_triangle(1+i,3+i))[1]
```

which returns the second vertex.

26.5.14 Vertices of a polygon in the plane, closed

The `vertices_abca` command finds the "closed" list of vertices (it repeats the beginning vertex).

- `vertices_abca` takes P, a polygon.
- `vertices_abca(`P`)` returns a closed list of the vertices of P and draws them.

26.5.15 A point on a geometric object in the plane

The `element` command is most useful in a 2D geometry screen; it creates objects that are restricted to a geometric figure.

`element` takes different types of arguments.

- `element` can take one mandatory argument and one optional argument:
  - $a..b$, a range of values.
  - Optionally, *init* and *step*, an initial value (by default $(a+b)/2$) and step size (by default $(b-a)/100$).
- `element(`$a..b\,\langle$, *init*, *step*$\rangle$`)` creates a parameter restricted to the interval from a to b, with the given initial value and whose value can be changed in the given step sizes.

For example, the command `t:=element(0..pi)` creates a parameter `t` which can take on values between 0 and π and has initial value $\pi/2$. It also creates a slider labeled `t` which can be used to change the values. The values of any later formulas involving `t` will change with `t`.

- `element` can take one mandatory argument and one optional argument:
  - C, a curve.
  - Optionally, *init*, an initial value (by default 1/2).
- `element(`$C\,\langle$,*init*$\rangle$`)` creates a point which will be restricted to the curve, the initial position of the point is determined by setting the parameter (in the default parameterization of the object) to the initial value. If the point can be moved by the mouse (as it can when the geometry screen is in `Pointer` mode), then the motion will be restricted to the geometric object.

For example, the command `A:=element(circle(0,2))` creates a point labeled `A` whose position is restricted to the circle of radius 2 centered at the origin. Since the circle has default parameterization $2e^{it}$, `A` starts out at $2e^{i/2}$.

- `element` can take two mandatory arguments:
  - P, a polygon or polygonal line with n sides.
  - `[floor(`t`),frac(`t`)]`, where t is a variable previously defined by `t=element(0..`n`)`.
- `element(`P`,[floor(`t`),frac(`t`)])` creates a point restricted to the polygonal line. With the sides numbered starting at 0, the value of `floor(`t`)` determines which side the point is on, and the value of `frac(`t`)` determines how far along the side the point is.

26.6 Lines in plane geometry

26.6.1 Lines and directed lines in the plane

See Section 27.4.1, p. 760 for lines in space.

The `line` command returns and draws a directed line. It can take its arguments in four different ways.

1. **Two points.**
   - `line` takes two arguments: P, Q, two points (which can also be given as a list).
   - `line(`P, Q`)` returns and draws the line whose direction is from the P to Q.
2. **A point and a slope.**
   - `line` takes two arguments:
     - p, a point.
     - `slope=`m
   - `line(`p`,slope=`m`)` returns and draws the line through the given point with the given slope, where direction of the line is determined by the slope.
3. **A point and a direction vector.**
   - `line` takes two arguments:
     - P, a point.
     - $[u_1, u_2]$, a direction vector.
   - `line(`P`,[`u_1, u_2`])` returns and draws the line through the given point with the direction given by the direction vector.
4. **An equation.**
   - `line` takes $ax + by + c = 0$, an equation.
   - `line(`$ax + by + c = 0$`)` returns and draws the line given by the equation. The direction of the line is given by $[b, -a]$.

Example

```
> line(0,1+i)
```

or:

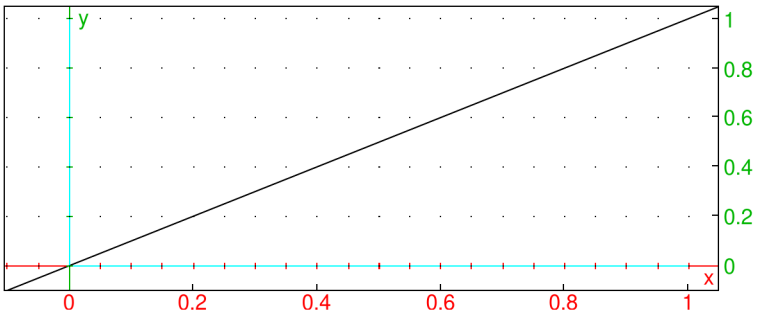
```
> line(1+i,slope=1)
```

or:

```
> line(1+i,[3,3])
```

or:

```
> line(y-x=0)
```



Remark. To draw a line with an additional argument for color (such as `color=blue`), this argument must be the third argument. In particular, for a list of two points to specify a line in this command, the list must be turned into a sequence, such as with `op`. For example, given a list `L` of two points (possibly the result of a different command) which determines a line, to draw the line `blue` enter `line(op(L),color=blue)`; entering `line(L,color=blue)` will result in an error.

26.6.2 Half-lines in the plane

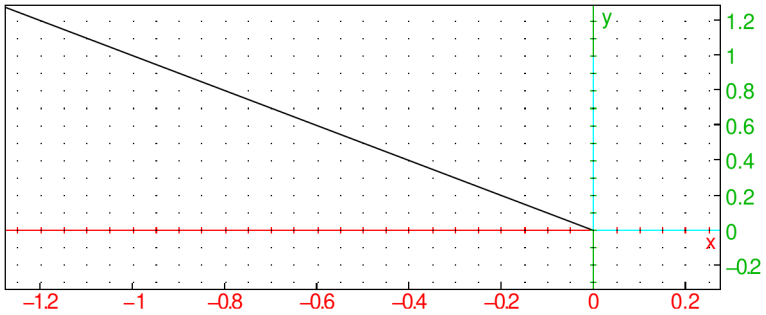
See Section 27.4.2, p. 761 for half-lines in space.

The `half_line` command creates rays.

- `half_line` take two arguments: P, Q, two points (which can also be given as a list).
- `half_line`(P, Q) returns and draws the ray from P through Q

Example

```
> half_line(0,-1+i)
```



26.6.3 Line segments in the plane

See Section 27.4.3, p. 761 for segments in space.

The `segment` command draws line segments. (The `segment` command can also draw vectors (see Section 26.6.4, p. 695.)

- `segment` takes two arguments: P, Q, two points (which can also be given as a list).
- `segment`(P, Q) returns the corresponding line segment and draws it.

The `Line` command also draws line segments, but with a slightly different syntax.

- `Line` takes four arguments: a, b, c, d, four real numbers.
- `Line`(a, b, c, d) returns and draws the line segment from (a, b) to (c, d).

Example

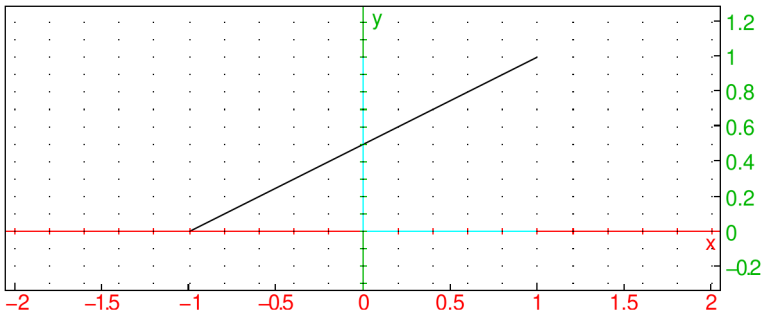
```
> segment(-1,1+i)
```

or:

```
> segment(point(-1),point(1,1))
```

or:

```
> Line(-1,0,1,1)
```



26.6.4 Vectors in the plane

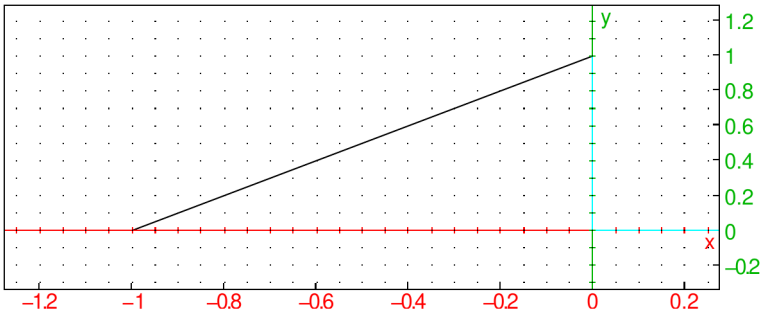
See Section 27.4.4, p. 762 for vectors in space.

The `segment` commands returns and draws vectors. (The `segment` command can also draw line segments, see section 26.6.3.)

- `segment` takes two arguments:
  - p, a point.
  - v, a vector.
- `segment`(p, v) returns the corresponding vector and draws it as a line segment from p to $p + v$.

Example

```
> segment([-1,0],[1,1])
```

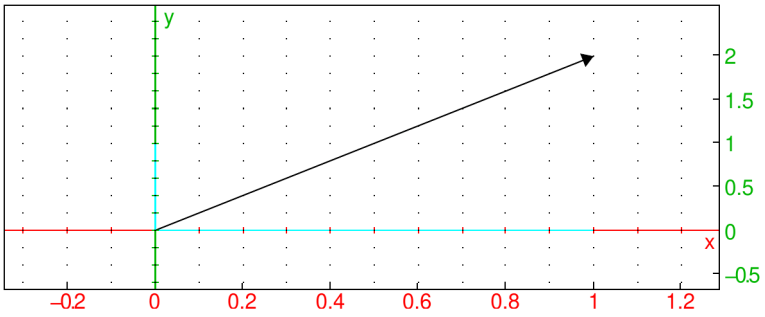


The `vector` command also makes vectors, with a different syntax. It can take its arguments in two different ways.

1. **The coordinates of the vector.**
   - `vector` takes L, a list of the coordinates of the vector.
   - `vector(`L`)` returns and draws the vector with the given coordinates, starting from the origin.
2. **Two points or a point and a vector.**
   - `vector` takes two arguments:
     - P, a point.
     - Q, a point or a vector. If Q is a point, it can be combined with P in a list.
   - `vector(`P, Q`)` returns and draws the corresponding vector. If the arguments are two points, the vector goes from P to Q. If the arguments are a point and a vector, then the vector starts at P.

Examples

```
> vector([1,2])
```



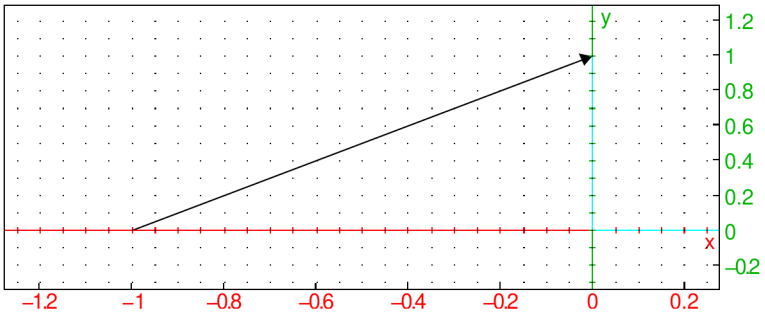

```
> vector([-1,0],[1,i])
```

or:

```
> vector(-1,i)
```

or:

```
> V:=vector(1,2+i):; vector(-1,V)
```



26.6.5 Parallel lines in the plane

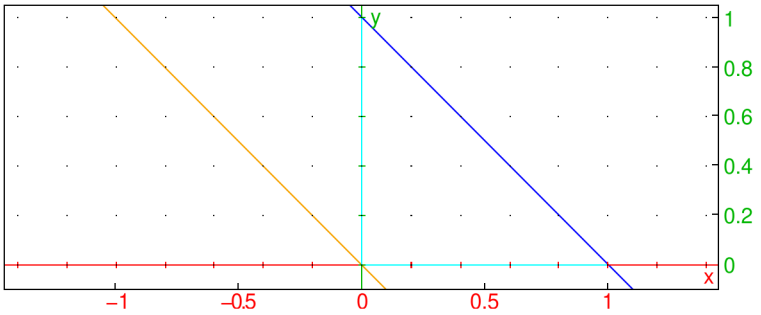
See Section 27.4.5, p. 763 for parallel lines in space.

The `parallel` command finds a line parallel to a given line.

- `parallel` takes two arguments:
  - p, a point.
  - ℓ, a line.
- `parallel`(p, ℓ) returns and draws the line parallel to ℓ passing through p.

Example

```
> L:=line(1,i,color=blue); parallel(0,L,color=orange)
```



26.6.6 Perpendicular lines in the plane

See Section 27.4.6, p. 765 for perpendicular lines in space.

The `perpendicular` command finds a line perpendicular to a given line.

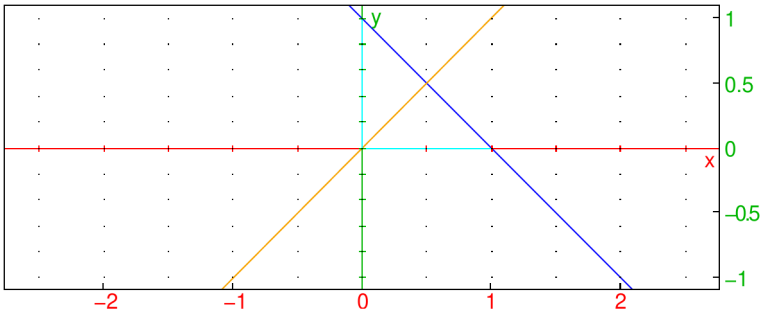
- `perpendicular` takes two arguments:
  - p, a point.
  - ℓ, a line. The line can also be specified by giving a sequence of two points on it.
- `perpendicular`(p, ℓ) returns and draws the line perpendicular to ℓ passing through p.

Example

```
> L:=line(1,i,color=blue); perpendicular(0,L,color=orange)
```

or:

```
> line(1,i,color=blue); perpendicular(0,1,i,color=orange)
```



26.6.7 Tangents to curves in the plane

See Section 27.5.3, p. 768 for tangents in space.

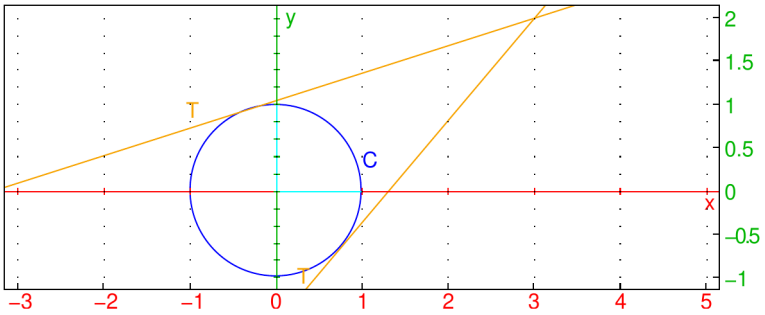
The `tangent` command finds tangents to curves.

- `tangent` takes one or two arguments:
  - C, a curve.
  - p, a point.

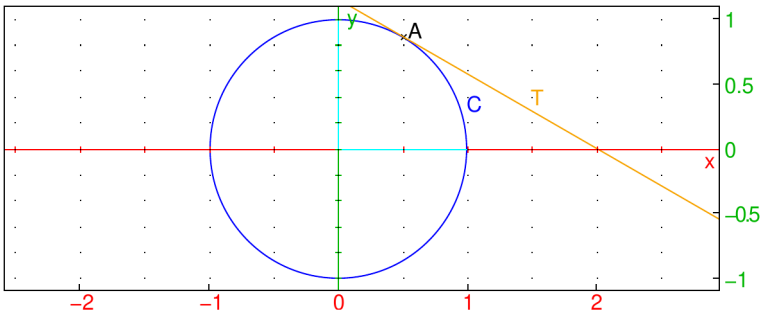
  or
  - e, a point defined with `element` (see Section 26.5.15, p. 692) using a curve and parameter value.
- `tangent(`C, p`)` (or `tangent(`e`)`) returns and draws the list of lines tangent to the curve passing through the given point.

Examples

```
> C:=circle(0,1,color=blue); T:=tangent(C,3+2i,color=orange)
```




```
> purge(t):; t:=element(0..pi,pi/3):;
  C:=circle(0,1,color=blue); A:=element(C,t); T:=tangent(A,color=orange)
```



When `tangent` is called with an element, the tangent will change along with the point on the element.

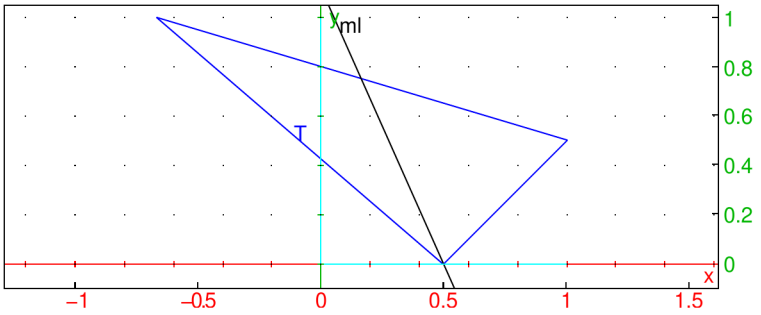
26.6.8 Median of a triangle in the plane

The `median_line` command finds a median line to a triangle.

- `median_line` takes three arguments: a, b, c, points.
- `median_line`(a, b, c) returns and draws the median line to the triangle with vertices a, b, c; through a and bisecting the segment from b to c.

Example

```
> T:=triangle(1/2,1+i/2,-2/3+i,color=blue); ml:=median_line(1/2,1+i/2,-2/3+i)
```



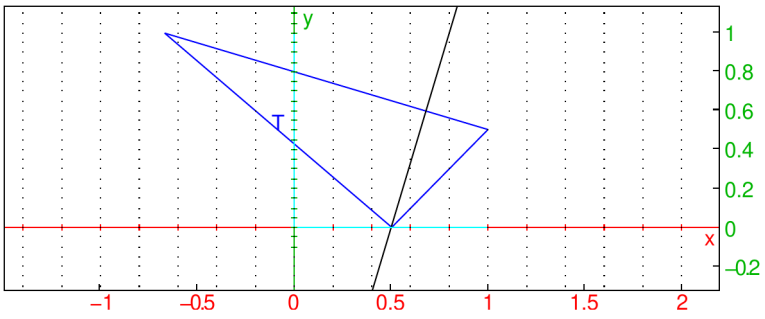
26.6.9 Altitude of a triangle

The `altitude` command finds an altitude line of a triangle.

- `altitude` takes three arguments: a, b, c, three points.
- `altitude`(a, b, c) returns and draws the altitude line to the triangle with vertices a, b, c, through a and perpedicular to the segment from b to c.

Example

```
> T:=triangle(1/2,1+i/2,-2/3+i,color=blue); altitude(1/2,1+i/2,-2/3+i)
```



26.6.10 Perpendicular bisector of a segment in the plane

See Section 27.5.2, p. 768 for perpendicular bisectors in space.

The `perpen_bisector` command finds the perpendicular bisector of a line segment.

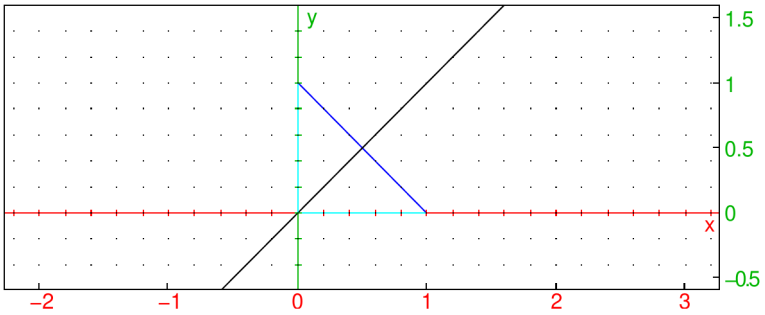
- `perpen_bisector` takes *seg*, a line segment (or the end points of the segment).
- `perpen_bisector`(*seg*) returns and draws the perpendicular bisector of *seg*.

Example

```
> segment(1,i,color=blue); perpen_bisector(1,i)
```

or:

```
> S:=segment(1,i,color=blue); perpen_bisector(S)
```



The `perpen_bisector` command can also take two lines as segments, in which case it returns and draws the perpendicular bisector of the segment from the first point defining the first line and the second point defining the second line.

26.6.11 Angle bisector

The `bisector` command finds angle bisectors.

- `bisector` takes three arguments: a, b, c, three points (which can also be given as a list).
- `bisector`(a, b, c) returns and draws the bisector of $\angle bac$.

26.6.12 Exterior angle bisector

The `exbisector` command finds exterior angle bisectors.

- `exbisector` takes three arguments: a, b, c, three points (which can also be given as a list).
- `exbisector`(a, b, c) returns and draws the bisector of the exterior angle of the triangle determined by a, b and c; a is the vertex of the angle, the opposite of the ray through a and b determine one side of the angle and a and c determine the second side.

26.7 Triangles in the plane

See Section 27.6, p. 769 for triangles in space.

26.7.1 Arbitrary triangles in the plane

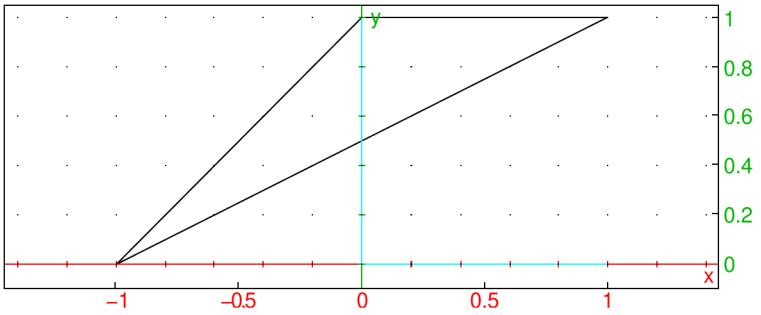
See Section 27.6.1, p. 769 for the `triangle` command in space.

The `triangle` command creates triangles.

- `triangle` takes three arguments: a, b, c, three points (which can be given as a list).
- `triangle`(a, b, c) returns and draws the triangle with vertices a, b and c.

Example

```
> triangle(-1,i,1+i)
```



26.7.2 Isosceles triangles in the plane

See Section 27.6.2, p. 770 for isosceles triangles in space.

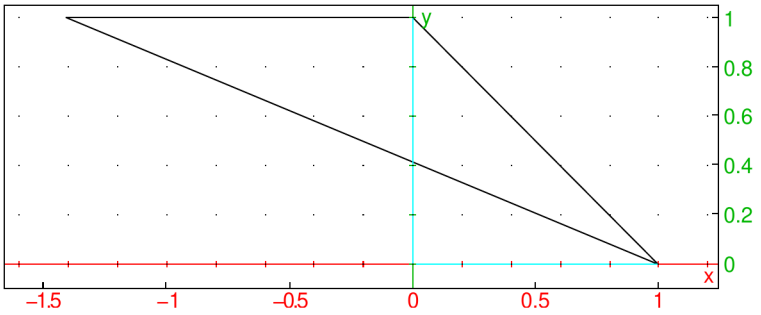
The `isosceles_triangle` command creates isosceles triangles.

- `isosceles_triangle` takes three mandatory arguments and one optional argument:
  - a, b, two points.
  - θ, an angle.
  - Optionally, `var`, a variable name.
- `isosceles_triangle`$(a, b, \theta\,\langle, var\rangle)$ returns and draws the isosceles triangle abc, where ab and ac are equal sides and θ is the angle between ab and ac.

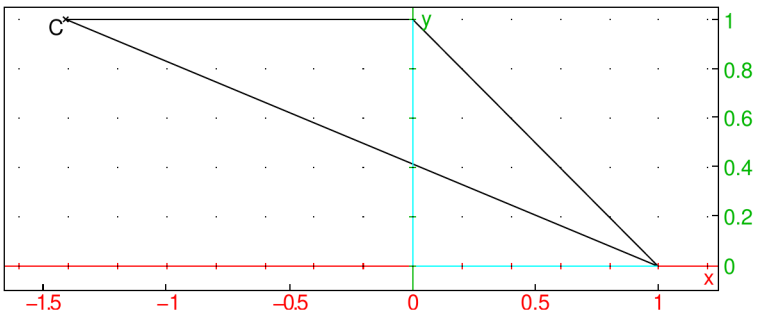
  With the argument var, c will be assigned to var.

Examples

```
> isosceles_triangle(i,1,-3*pi/4)
```




```
> isosceles_triangle(i,1,-3*pi/4,C)
```



```
> normal(affix(C))
```

$$-\sqrt{2}+\mathrm{i}$$

26.7.3 Right triangles in the plane

See Section 27.6.3, p. 771 for right triangles in space.

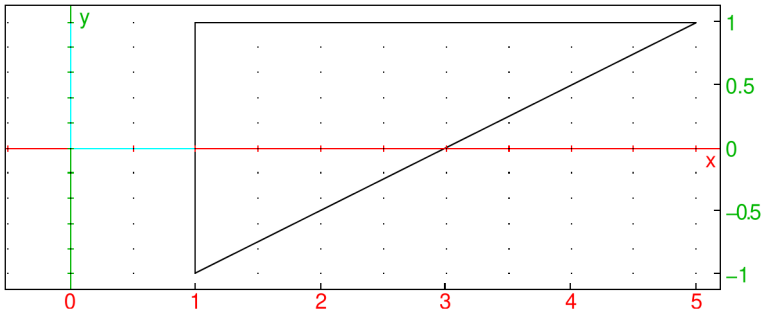
The `right_triangle` command creates right triangles.

- `right_triangle` takes three mandatory arguments and one optional argument:
  - A, B, two points.
  - k, a nonzero real number.
  - Optionally *var*, a variable name.
- `right_triangle`$(A, B, k\,\langle, var\rangle)$ returns and draws the right triangle ABC, with the right angle at A and with length$(AC) = |k| \cdot$ length(AB). If $k > 0$, then AB to AC is counterclockwise; if $k < 0$ then AB to AC is clockwise.

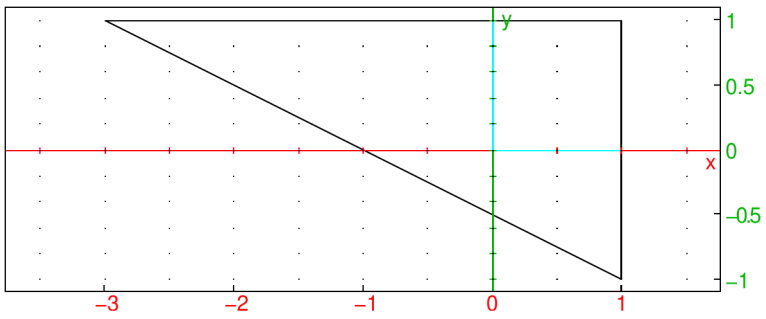
  With the argument *var*, c will be assigned to *var*.

Examples

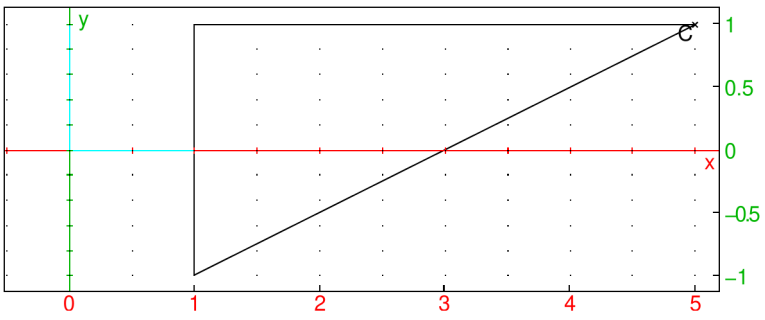
```
> right_triangle(1+i,1-i,2)
```




```
> right_triangle(1+i,1-i,-2)
```




```
> right_triangle(1+i,1-i,2,C)
```




```
> affix(C)
```

$$5+\mathrm{i}$$

26.7.4 Equilateral triangles in the plane

See Section 27.6.4, p. 773 for equilateral triangles in space.

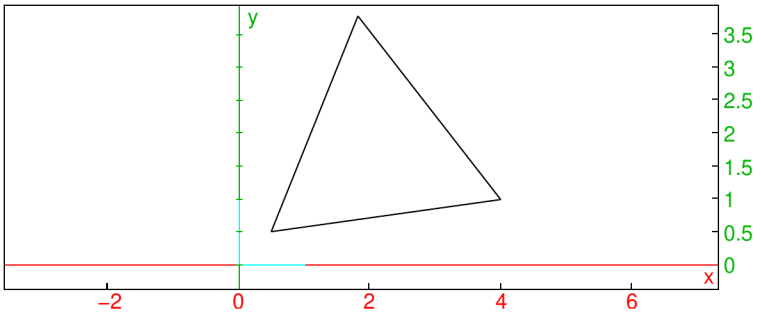
The `equilateral_triangle` command creates equilateral triangles.

- `equilateral_triangle` takes two mandatory arguments and one optional argument:
  - A, B, two points.
  - Optionally, *var*, a variable name.
- `equilateral_triangle(`$A, B\,\langle, var\rangle$`)` returns and draws the equilateral ABC, where AB to AC is counterclockwise.

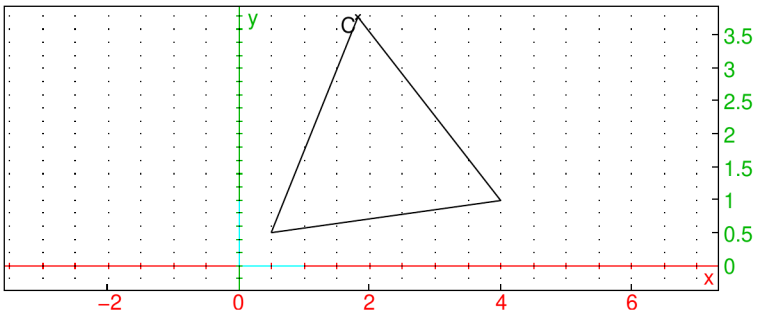
  With the argument *var*, C will be assigned to *var*.

Examples

```
> equilateral_triangle(1/2+i/2,4+i)
```




```
> equilateral_triangle(1/2+i/2,4+i,C)
```



```
> evalc(affix(C))
```

$$\frac{-2\sqrt{3}+18}{8}+\frac{\mathrm{i}\left(14\sqrt{3}+6\right)}{8}$$

26.8 Quadrilaterals in the plane

See Section 27.7, p. 773 for quadrilaterals in space.

26.8.1 Squares in the plane

See Section 27.7.1, p. 773 for squares in space.

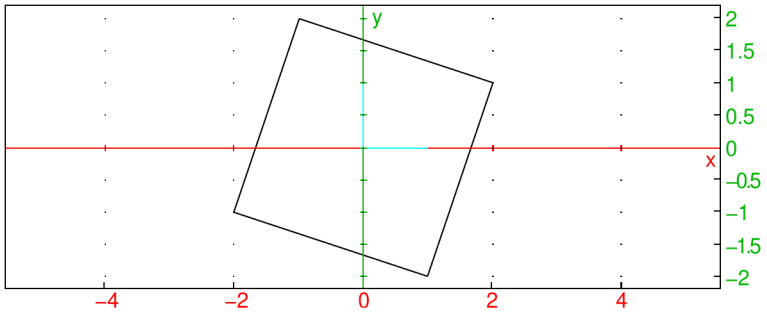
The `square` command creates squares.

- `square` takes two mandatory arguments and two optional arguments.
  - A, B, two points.
  - Optionally, *varc*, *vard*, two variable names.
- `square(`A, B $\langle$, *varc*, *vard*$\rangle$`)` returns and draws the square $ABCD$, where the square is traversed counterclockwise.

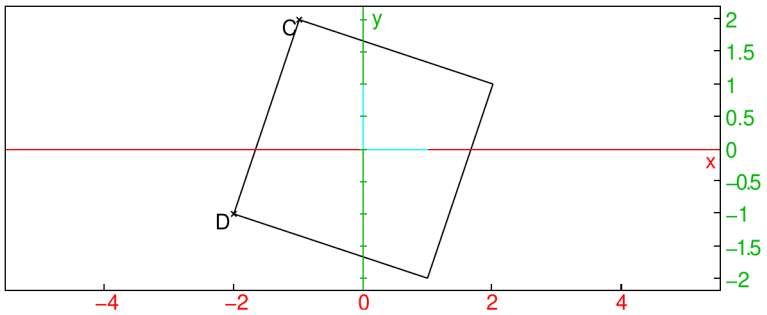
  If the arguments *varc* and *vard* are given, then C and D will be assigned to them.

Examples

```
> square(1-2i,2+i)
```




```
> square(1-2i,2+i,C,D)
```




```
> affix(C), affix(D)
```

$$-1+2\mathrm{i}, -2-\mathrm{i}$$

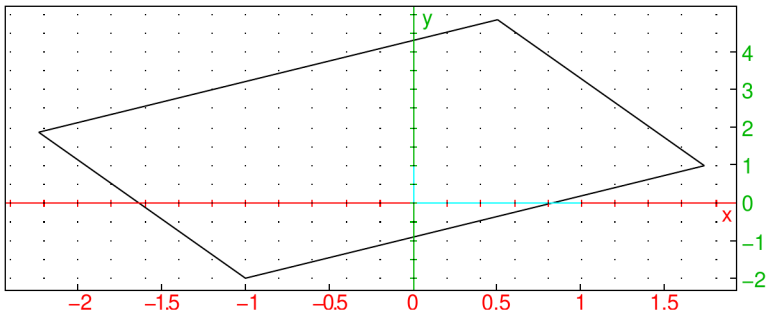
26.8.2 Rhombuses in the plane

See Section 27.7.2, p. 774 for rhombuses in space.
The `rhombus` command creates rhombuses.

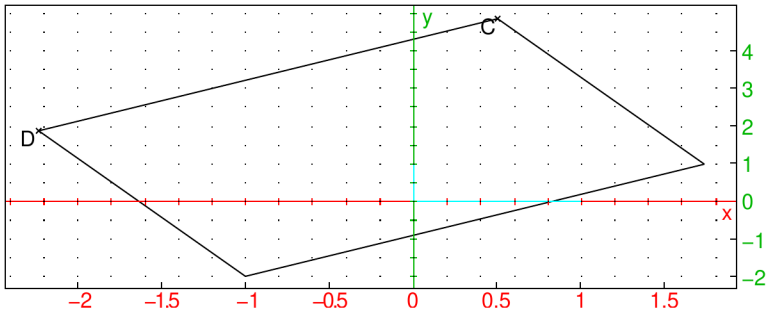
- `rhombus` takes three mandatory arguments and two optional arguments.
  - A, B, two points.
  - a, a real number.
  - Optionally, *varc*, *vard*, two variable names.
- `rhombus`$(A, B, a\,\langle, varc, vard\rangle)$ returns and draws the rhombus $ABCD$, where a is the counterclockwise angle from AB to AC. If the arguments *varc* and *vard* are given, then C and D will be assigned to them.

Examples

```
> rhombus(-1-2*i,sqrt(3)+i,pi/3)
```




```
> rhombus(-1-2*i,sqrt(3)+i,pi/3,C,D)
```




```
> evalc(affix(C),affix(D))
```

$$\frac{1}{2}+\frac{\mathrm{i}\left(\sqrt{3}+8\right)}{2},\frac{-2\sqrt{3}-1}{2}+\frac{\mathrm{i}\left(\sqrt{3}+2\right)}{2}$$

26.8.3 Rectangles in the plane

See Section 27.7.3, p. 775 for rectangles in space.
The `rectangle` creates rectangles.

- `rectangle` takes three mandatory arguments and two optional arguments:
  - A, B, two points.

– k, a nonzero real number.

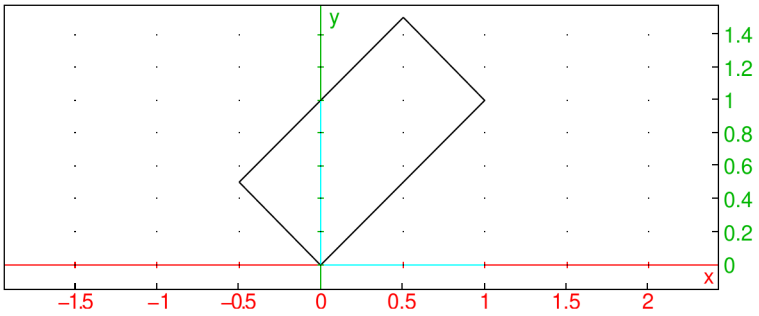
– Optionally, *varc*, *vard*, two variable names.

- `rectangle`$(A, B, k\,\langle, varc, vard\rangle)$ returns and draws the rectangle $ABCD$, where $AD = |k| \cdot AB$ and the angle from AB to AD is counterclockwise if $k > 0$, clockwise if $k < 0$.

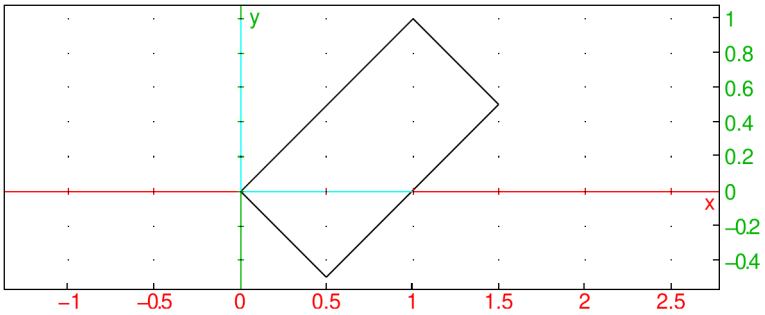
  If the arguments *varc* and *vard* are given, then C and D will be assigned to them.

Examples

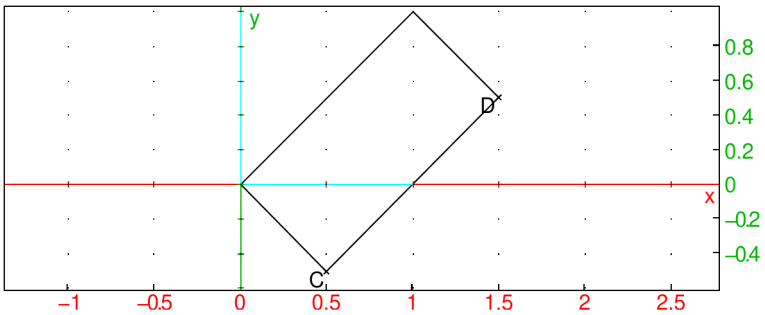
```
> rectangle(0,1+i,1/2)
```




```
> rectangle(0,1+i,-1/2)
```




```
> rectangle(0,1+i,-1/2,C,D)
```



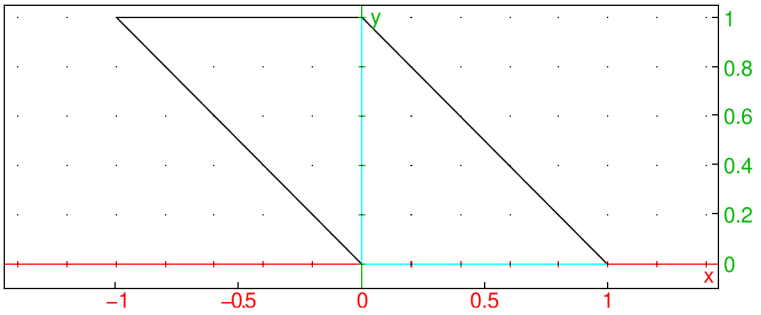

```
> affix(C),affix(D)
```

$$\frac{1}{2} - \frac{\mathrm{i}}{2}, \frac{3}{2} + \frac{\mathrm{i}}{2}$$

Given `rectangle`(A, B, k), Xcas computes D by `affix`(D) = `affix`(A) + $k\mathrm{e}^{\mathrm{i}\pi/2}$(`affix`$(B)$ − `affix`(A)). If k is complex, then `rectangle` draws a parallelogram.

Example

```
> rectangle(0,1,1+i)
```



26.8.4 Parallelograms in the plane

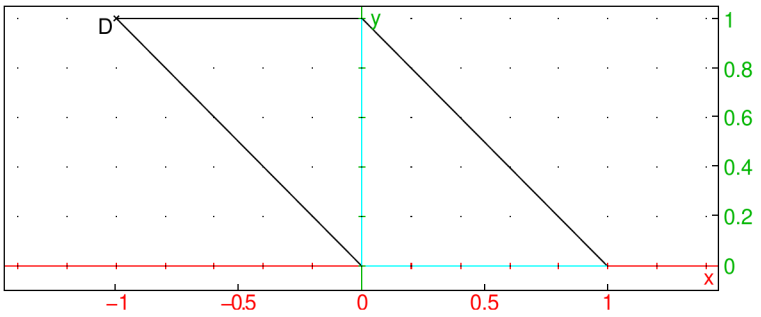
See Section 27.7.4, p. 777 for parallelograms in space.
The `parallelogram` command creates parallelograms.

- `parallelogram` takes three mandatory arguments and one optional argument:
  - A, B, C, three points.
  - Optionally, *var*, a variable name.
- `parallelogram(`$A, B, C \langle, var \rangle$`)` returns and draws the parallelogram $ABCD$ for the appropriate `D`.

  If the argument *var* is given, then D will be assigned to it.

Examples

```
> parallelogram(0,1,i,D)
```




```
> affix(D)
```

$$-1+\mathrm{i}$$

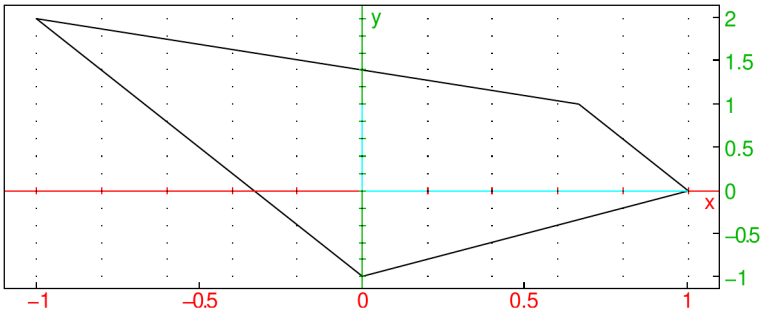
26.8.5 Arbitrary quadrilaterals in the plane

See Section 27.7.5, p. 777 for quadrilaterals in space.
The `quadrilateral` creates arbitrary quadrilaterals.

- `quadrilateral` takes four arguments: A, B, C, D, four points.
- `quadrilateral(`A, B, C, D`)` returns and draws the quadrilateral $ABCD$.

Example

```
> quadrilateral(-i,1,2/3+i,-1+2*i)
```



26.9 Other polygons in the plane

See Section 27.8, p. 778 for polygons in space.

26.9.1 Regular hexagons in the plane

See Section 27.8.1, p. 778 for hexagons in space.

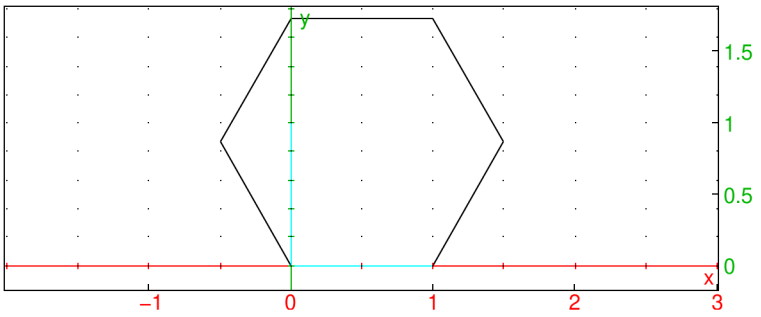
The `hexagon` command creates hexagons.

- `hexagon` takes two mandatory arguments and four optional arguments:
  - A, B, two points.
  - Optionally, `varc`, `vard`, `vare`, *varf*, variable names.
- `hexagon(`A, B, C ⟨, *varc*, *vard*, *vare*⟩`)` returns and draws the regular hexagon $ABCDEF$, where the vertices are counterclockwise.

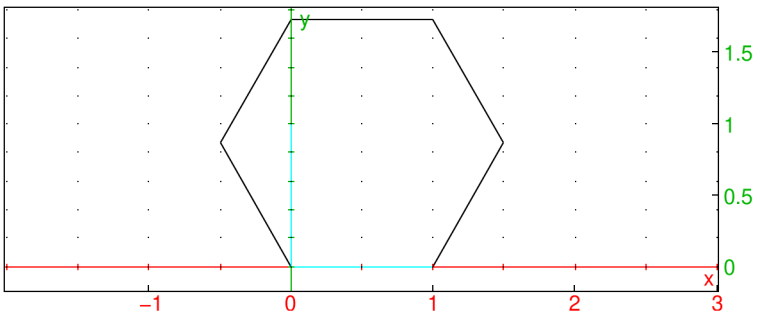
  If the arguments *varc*, *vard*, *vare*, *varf* are given, then the points C, D, E and F will be assigned to them.

Examples

```
> hexagon(0,1)
```




```
> hexagon(0,1,C,D,E,F)
```




```
> affix(C), affix(D), affix(E), affix(F)
```

$$\frac{\sqrt{3}\mathrm{i}+1}{2}+1, \frac{2}{2}\left(\sqrt{3}\mathrm{i}+1\right), \frac{2}{2}\left(\sqrt{3}\mathrm{i}+1\right)-1, \frac{\sqrt{3}\mathrm{i}+1}{2}-1$$

26.9.2 Regular polygons in the plane

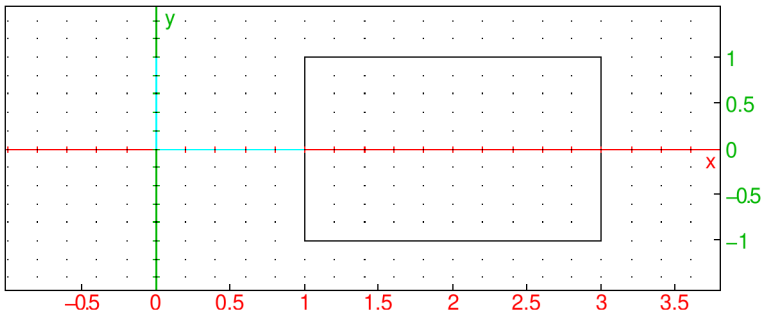
See Section 27.8.2, p. 779 for regular polygons in space.

The `isopolygon` command creates regular polygons.

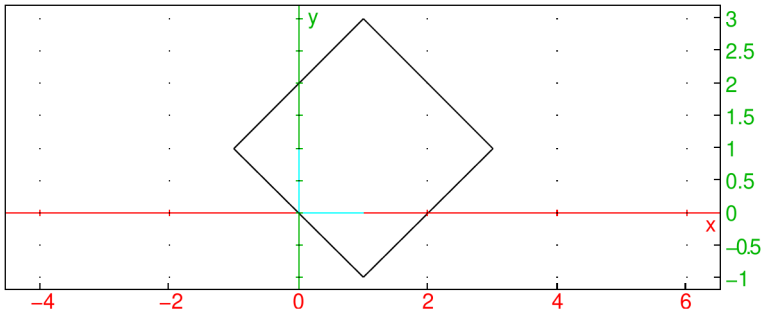
- `isopolygon` takes three arguments:
  - A, B, two points.
  - k, a nonzero integer.
- `isopolygon`$(A, B, k$ returns and draws the regular $|k|$-sided polygon with one side AB. If $k > 0$, then the polygon will continue counterclockwise; if $k < 0$, then it will be clockwise.

Examples

```
> isopolygon(1+i,1-i,4)
```




```
> isopolygon(1+i,1-i,-4)
```



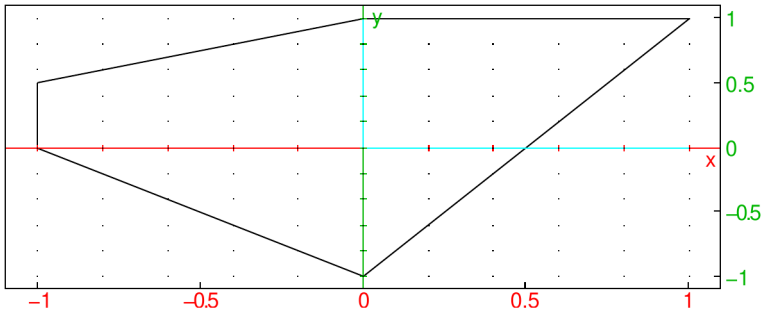
26.9.3 General polygons in the plane

See Section 27.8.3, p. 779 for general polygons in space.
The `polygon` command draws general polygons.

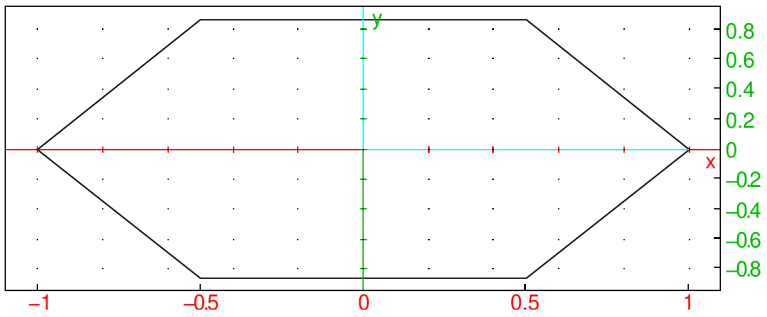
- `polygon` takes *points*, a sequence or list of points.
- `polygon(`*points*`)` returns and draws the polygon with vertices given by the points.

Examples

```
> polygon(-1,-1+i/2,i,1+i,-i)
```




```
> polygon(makelist(x->exp(i*pi*x/3),0,5,1))
```



26.9.4 Polygonal lines in the plane

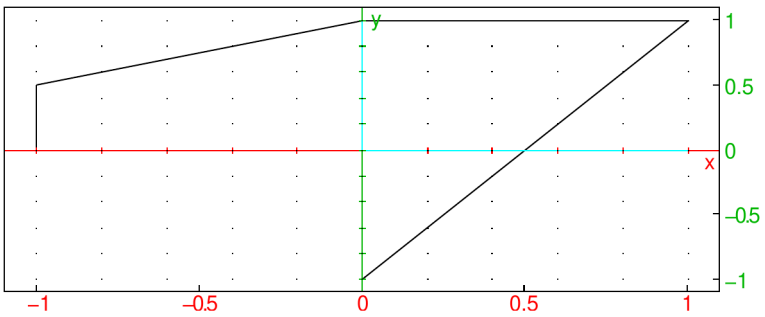
See Section 27.8.4, p. 780 for polygonal lines in space.
The `open_polygon` command draws a polygonal path.

- `open_polygon` takes an arbitrary number of points:

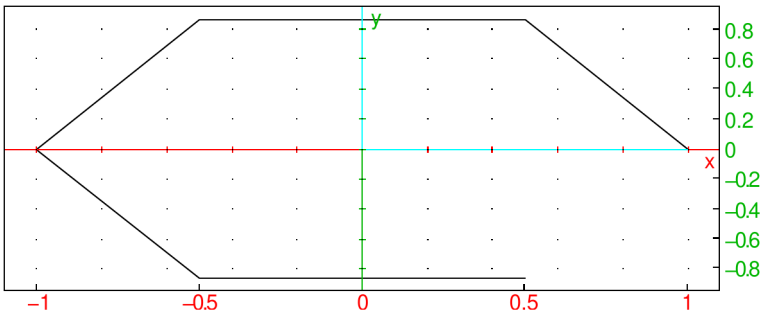
  points, a sequence or list of points.
- `open_polygon(`*points*`)` returns and draws the polygon line with the vertices given by the points.

Examples

```
> open_polygon(-1,-1+i/2,i,1+i,-i)
```




```
> open_polygon(makelist(x->exp(i*pi*x/3),0,5,1))
```



26.9.5 Convex hulls

The `convexhull` command uses the Graham scanning algorithm to find the convex hull of a set of points.

- `convexhull` takes *points*, a sequence or list of points.
- `convexhull(`*points*`)` returns the vertices of the convex hull of the points.

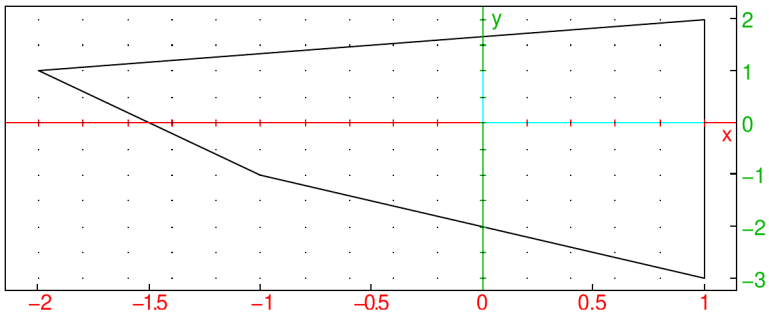
Example

```
> convexhull(0,1,1+i,1+2i,-1-i,1-3i,-2+i)
```

$$1-3\mathrm{i},\ 1+2\mathrm{i},\ -2+\mathrm{i},\ -1-\mathrm{i}$$

To draw the hull, use the `polygon` command with the output of `convexhull` (see Section 26.9.3, p. 710).

```
> polygon(convexhull(0,1,1+i,1+2i,-1-i,1-3i,-2+i))
```



26.10 Circles

26.10.1 Circles and arcs in the plane

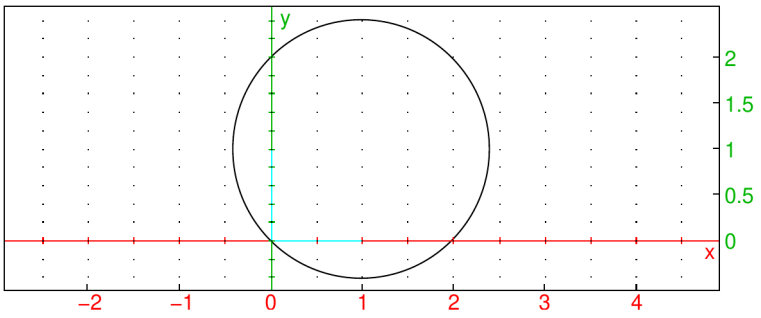
See also Section 26.10.2, p. 713.

See Section 27.9.1, p. 780 for circles in space.
The `circle` command creates circles and arcs. You can specify the circle in different ways.

- `circle` can take one argument: *eqn*, the equation of a circle with variables x and y (or an expression assumed to be set to 0).

  `circle(`*eqn*`)` returns and draws the circle. For example:

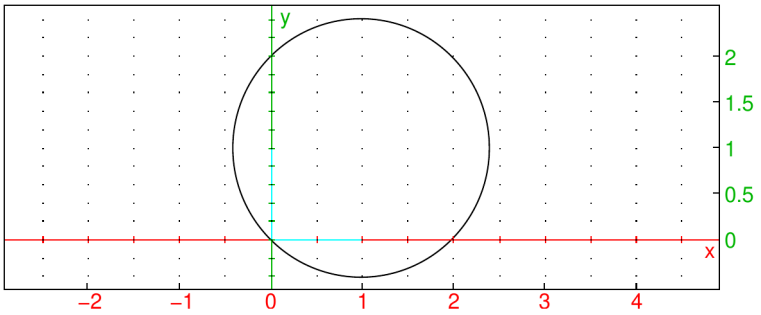
  ```
  > circle(x^2+y^2-2*x-2*y)
  ```



- `circle` can take two arguments:
  - P, a point.
  - α, a complex number.

  `circle(`P, α`)` returns and draws the circle centered at P and whose radius is $|\alpha|$. For example:

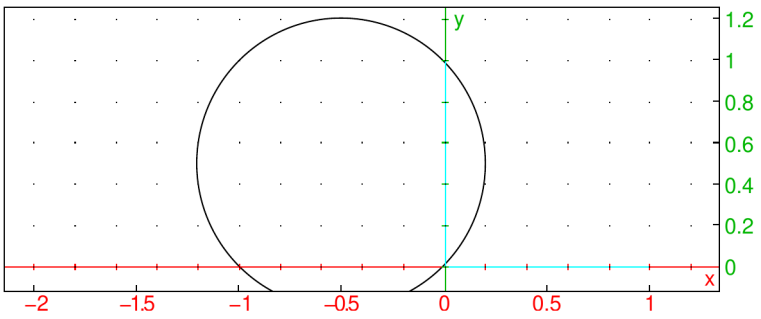
  ```
  > circle(-1,i)
  ```



- `circle` can take two arguments: A, B, two points (where B must be the value of `point` and not simply the affix).

  `circle(`A, B`)` returns and draws the circle whose diameter is AB. For example:

  ```
  > circle(-1,point(i))
  ```



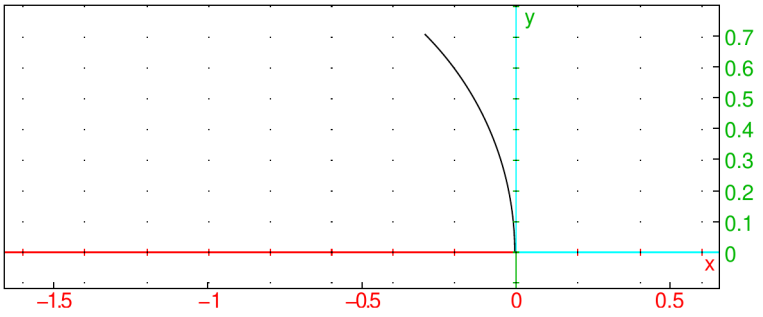
- `circle` can take four mandatory arguments and two optional arguments:
  - C, a point.
  - r, a complex number.
  - a, b, two real numbers.
  - Optionally, var_1, var_2, variable names.

  `circle`(C, r, a, b) returns and draws an arc of the circle with center C and radius $|r|$, with central angles a and b. The angles start on the axis defined by C and $C + r$.

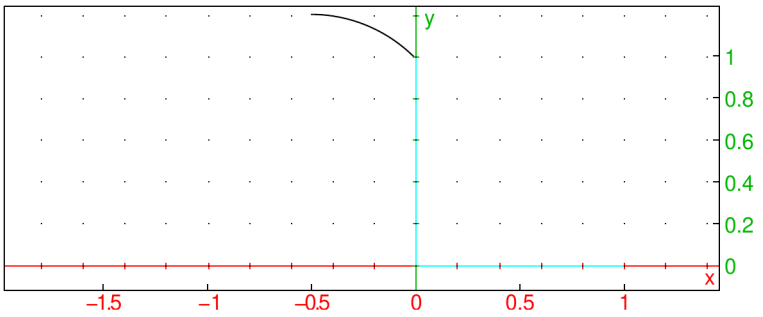
  If the arguments var_1 and var_2 are given, they will be assigned to the ends of the arc.

  For example:

```
> circle(-1,1,0,pi/4)
```




```
> circle(-1,point(i),0,pi/4)
```



26.10.2 Circular arcs

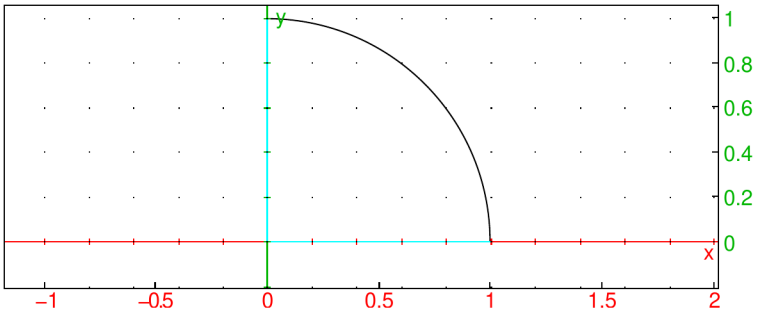
See also Section 26.10.1, p. 711

The `arc` command creates circular arcs.

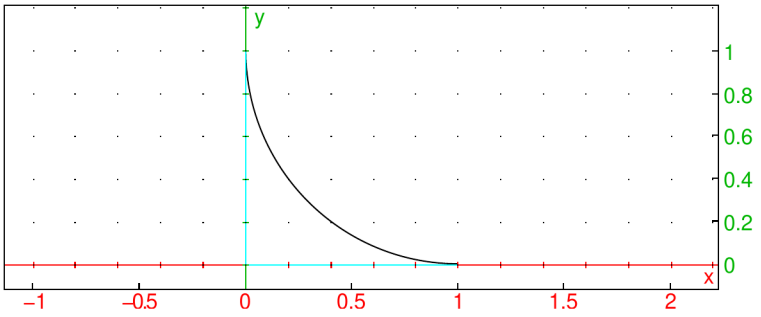
- `arc` takes three mandatory arguments and two optional arguments:
  - A, B, two points.
  - a, a real number between -2π and 2π.
  - Optionally, *varc*, *varr*, two variable names.
- `arc`$(A, B, a\langle\rangle)$ returns and draws the circular arc from A to B that represents and angle of a. (Note that the center of the circle will be $(A + B)/2 + \mathrm{i}(B - A)/(2\tan(a/2))$.)
  If the arguments `varc,varr` are given, they will be assigned the center and radius of the circle.

Examples

```
> arc(1,i,pi/2)
```




```
> arc(1,i,-pi/2)
```



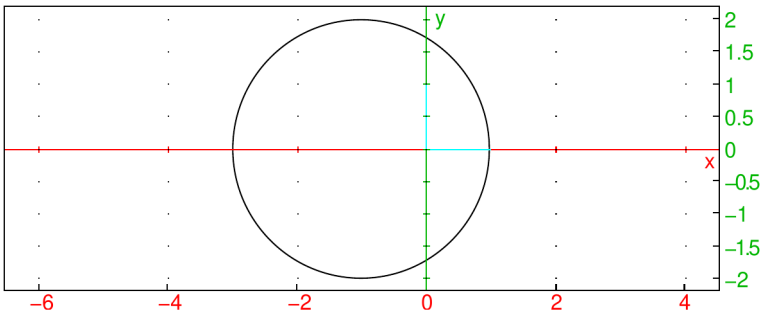
26.10.3 Circles (TI compatibility)

The `Circle` command creates a circle.

- `Circle` takes three mandatory arguments and one optional argument:
  - x, y, r, three real numbers.
  - Optionally, n, either 0 or 1 (by default, 1).
- `Circle`(x, y, r, n) returns the circle centered at (x, y) with radius r. If $n = 1$, it also draws the circle; if $n = 0$, it erases it.

Example

```
> Circle(-1,0,2)
```



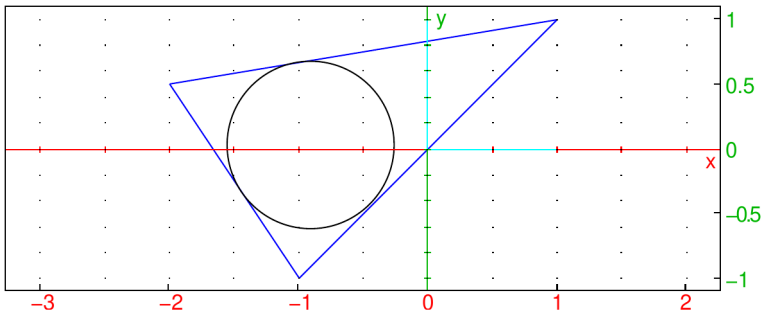
26.10.4 Inscribed circles

The `incircle` command creates the inscribed circle of a triangle.

- `incircle` takes three arguments: A, B, C, three points.
- `incircle`(A, B, C) returns and draws the circle inscribed in triangle ABC.

Example

```
> triangle(-1-i,-2+i/2,1+i,color=blue); incircle(-1-i,-2+i/2,1+i)
```



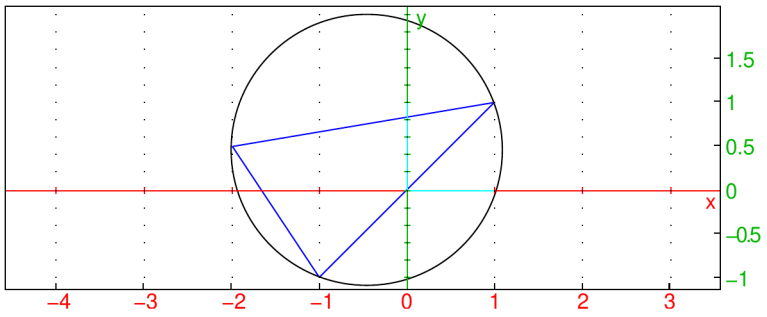
26.10.5 Circumscribed circles

The `circumcircle` command creates the circumscribed circle of a triangle.

- `circumcircle` takes three arguments: A, B, C, three points.
- `circumcircle`(A, B, C) returns and draws the circle circumscribed about triangle ABC.

Example

```
> triangle(-1-i,-2+i/2,1+i,color=blue); circumcircle(-1-i,-2+i/2,1+i)
```



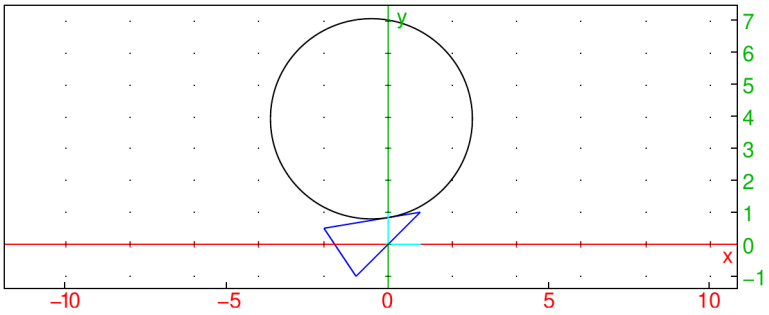
26.10.6 Excircles

The `excircle` draws an excircle of a triangle.

- `excircle` takes three arguments: A, B, C, three points.
- `excircle`(A, B, C) returns and draws the excircle of the triangle ABC in the interior angle of A.

Example

```
> triangle(-1-i,-2+i/2,1+i,color=blue); excircle(-1-i,-2+i/2,1+i)
```



26.10.7 Power of a point relative to a circle

Given a circle C of radius r and a point A at a distance of d from the center of C, the power of A relative to C is $d^2 - r^2$.

The `powerpc` command finds the power of a point relative to a circle.

- `powerpc` takes two arguments:
  - C, a circle.
  - P, a point.
- `powerpc`(C, P) returns the power of P relative to C.

Example

```
> powerpc(circle(0,1+i),3+i)
```

8

26.10.8 Radical axis of two circles

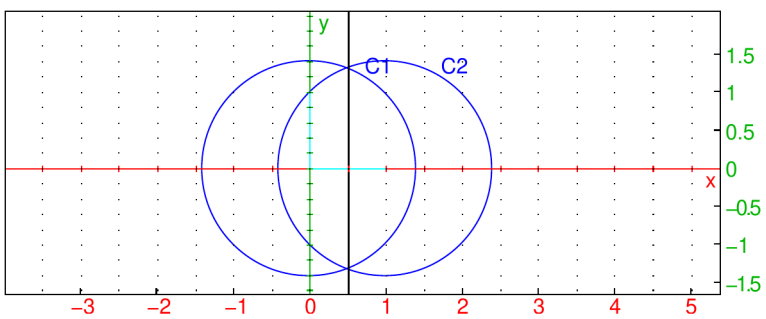
The radical axis of two circles is the set of points which have the same power with respect to each circle.

The `radical_axis` command finds the radical axis of two circles.

- `radical_axis` takes two arguments: C_1, C_2, two circles.
- `radical_axis`(C_1, C_2) returns and draws the radical axis of C_1 and C_2.

Example

```
> C1:=circle(0,1+i,color=blue); C2:=circle(1,1+i,color=blue); radical_axis(C1,C2)
```



26.11 Other conic sections

26.11.1 Ellipse in the plane

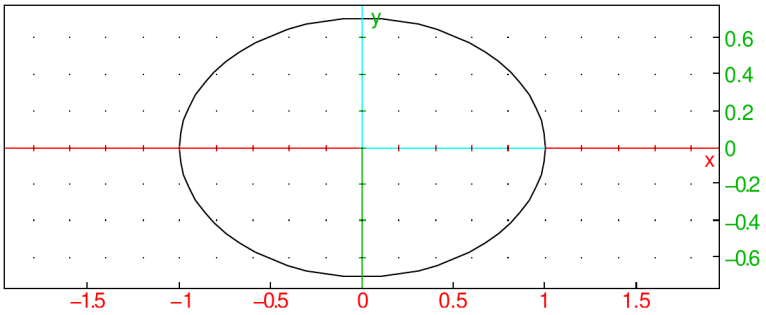
For ellipses in space, see Section 27.9.2, p. 781.

The `ellipse` command draws ellipses and other conic sections. `ellipse` can take parameters in two different ways.

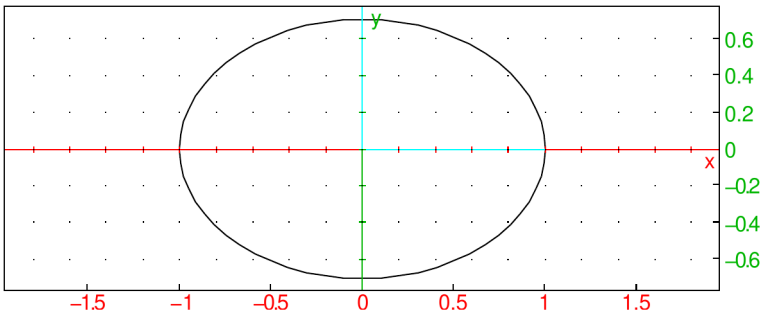
- `ellipse` can take one argument: *eqn*, a second degree equation in the variables `x` and `y` (or an expression which will be set to zero).
- `ellipse(`*eqn*`)` returns and draws the conic section given by *eqn*.
- Alternatively, `ellipse` can take three arguments:
  - A, B, two points.
  - C, a point or a real number.
- `ellipse(`A, B, C`)` returns and draws the ellipse with foci A and B and passing through C (if C is a point) or whose semi-major axis has length C (if C is a real number).
- Note that if the third argument is a point on the real axis, the real affix of the point won't work, it needs to be specified with the `point` command.

Examples

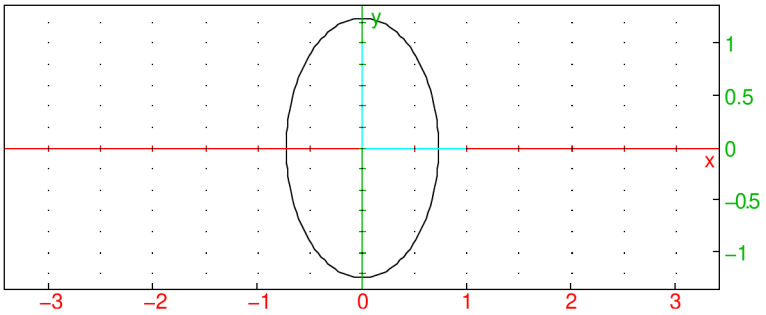
```
> ellipse(x^2+2*y^2-1)
```




```
> ellipse(-i,i,i+1)
```




```
> ellipse(-i,i,sqrt(5)-1)
```



26.11.2 Hyperbola in the plane

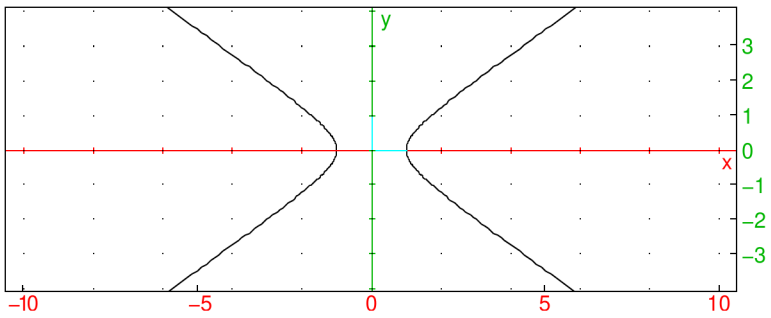
For hyperbolas in space, see Section 27.9.3, p. 782.

The `hyperbola` command draws hyperbolas and other conic sections. `hyperbola` can take parameters in two different ways.

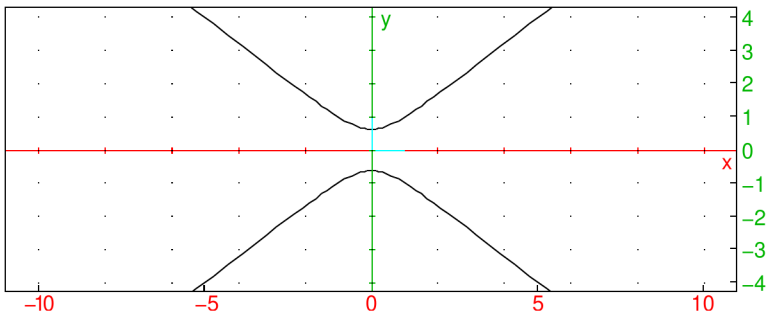
- `hyperbola` can take one argument: *eqn*, a second degree equation in the variables `x` and `y` (or an expression which will be set to zero).
- `hyperbola(`*eqn*`)` returns and draws the conic section given by the equation *eqn*.
- Alternatively, `hyperbola` can take three arguments:
  - A, B, two points.
  - C, a point or a real number.
- `hyperbola(`A, B, C`)` returns and draws the hyperbola with foci A and B and passing through C (if C is a point) or whose semi-major axis has length C (if C is a real number).
- Note that if the third argument is a point on the real axis, the real affix of the point won't work, it needs to be specified with the `point` command.

Examples

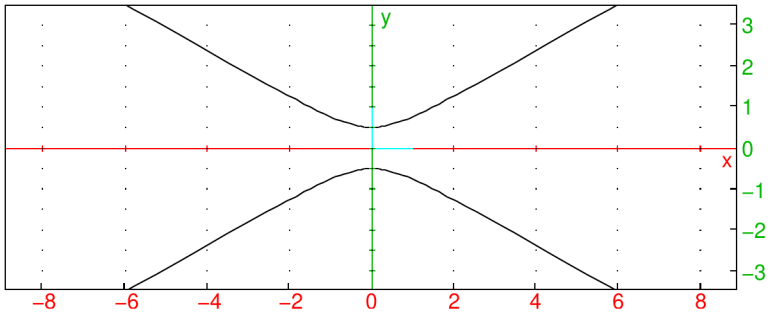
```
> hyperbola(x^2-2*y^2-1)
```




```
> hyperbola(-i,i,i+1)
```




```
> hyperbola(-i,i,1/2)
```



26.11.3 Parabola in the plane

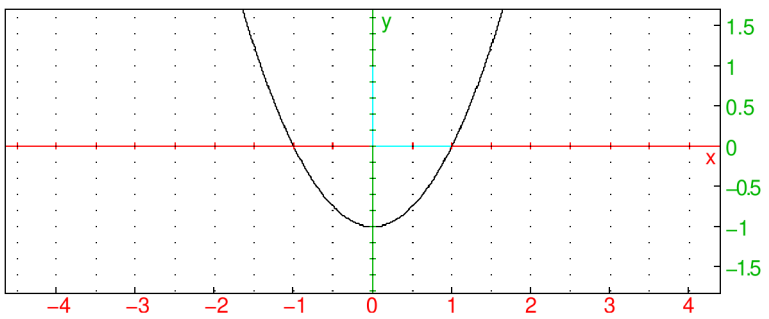
For parabolas in space, see Section 27.9.4, p. 782.

The `parabola` command draws parabolas and other conic sections. `parabola` can take parameters in two different ways.

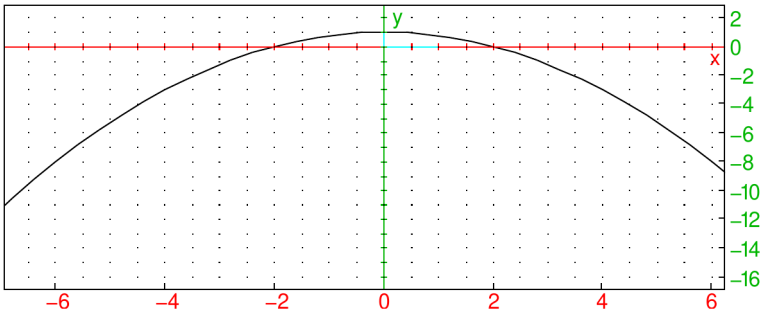
- `parabola` can take one argument: *eqn*, a second degree equation in the variables `x` and `y` (or an expression which will be set to zero).
- `parabola(`*eqn*`)` returns and draws the conic section given by the equation *eqn*.
- Alternatively, `parabola` can take two arguments: F, V, two points, or
  - $A = (a, b)$, a point.
  - c, a real number.
- `parabola(`F, V`)` returns and draws the parabola with focus F and vertex V.
- `parabola(`A, c`)` returns and draws the parabola $y = b + c(x - a)^2$.

Examples

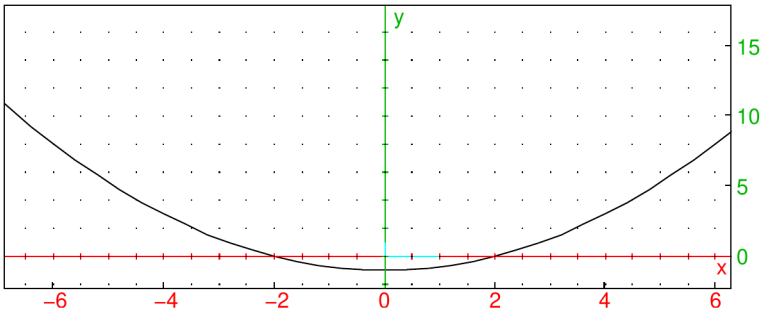
```
> parabola(x^2-y-1)
```



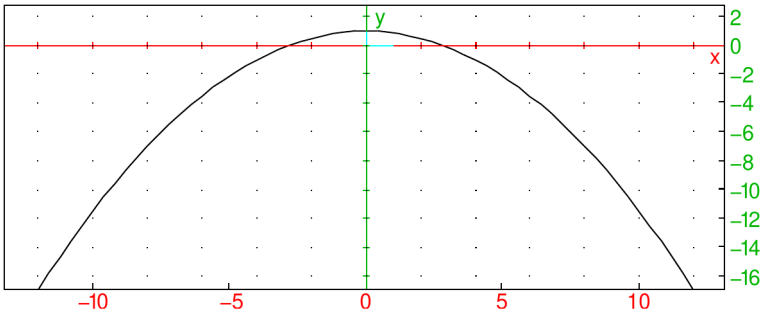
```
> parabola(0,i)
```




```
> parabola(-i,1)
```




```
> parabola(-i,i,1/2)
```



26.12 Coordinates in the plane

26.12.1 Affix of a point or vector

The `affix` command finds the affix of a point or vector (the corresponding complex number).

- `affix` takes P, a point or vector.
- `affix(`P`)` returns the affix of P.

Examples

```
> affix(point(2,3))
```

$$2+3\mathrm{i}$$

```
> affix(vector(-1,i))
```

$$1+\mathrm{i}$$

26.12.2 Abscissa of a point or vector in the plane

See Section 27.10.1, p. 782 for abscissas in 3D geometry.

The `abscissa` command finds the abscissa (x-coordinate) of a point.

- `abscissa` takes P, a point.
- `abscissa(`P`)` returns the abscissa of P.

Examples

```
> abscissa(point(1+2*i))
```

$$1$$

```
> abscissa(point(i)-point(1+2*i))
```

$$-1$$

```
> abscissa(1+2*i)
```

$$1$$

```
> abscissa([1,2])
```

$$1$$

26.12.3 Ordinate of a point or vector in the plane

See Section 27.10.2, p. 783 for ordinates in 3D geometry.

The `ordinate` command finds the ordinate (y coordinate) of a point.

- `ordinate` takes P, a point.
- `ordinate(`P`)` returns the ordinate of P.

Examples

```
> ordinate(point(1+2*i))
```

$$2$$

```
> ordinate(point(i)-point(1+2*i))
```

$$-1$$

```
> ordinate(1+2*i)
```

$$2$$

```
> ordinate([1,2])
```

$$2$$

26.12.4 Coordinates of a point, vector or line in the plane

See Section 27.10.4, p. 783 for coordinates in 3D geometry.

The `coordinates` finds the coordinates of a point or two points that determine a line.

- `coordinates` takes X, either a point, a sequence or list of points, or a line.
- `coordinates(`X`)` returns:
  - a list consisting of the abscissa and ordinate of X, if X is a point or a vector, or a sequence or list of such lists, if X is a sequence or list of points.
  - a list of two points on the line X, in the order determined by the direction of the line, if X is a line.

Examples

```
> coordinates(1+2*i)
```

or:

```
> coordinates(point(1+2*i))
```

or:

```
> coordinates(vector(1+2*i))
```

$$[1, 2]$$

```
> coordinates(point(1+2*i)-point(i))
```

or:

```
> coordinates(vector(i,1+2*i))
```

or:

```
> coordinates(vector(point(i),point(1+2*i)))
```

or:

```
> coordinates(vector([0,1],[1,2]))
```

$$[1, 1]$$

```
> d:=line(-1+i,1+2*i)
```

or:

```
> d:=line(point(-1,1),point(1,2))
```

then:

```
> coordinates(d)
```

$$[-1 + \mathrm{i}, 1 + 2\mathrm{i}]$$

```
> coordinates(line(y=x/2+3/2))
```

$$\left[\frac{3\mathrm{i}}{2}, 1 + 2\mathrm{i}\right]$$

```
> coordinates(line(x-2*y+3=0))
```

$$\left[\frac{3\mathrm{i}}{2}, \frac{-4 + \mathrm{i}}{2}\right]$$

```
> coordinates(i,1+2*i)
```

or:

```
> coordinates(point(i),point(1+2*i))
```

$$[0,1],[1,2]$$

Note that if the argument is a list of real numbers, it is interpreted as a list of points on the real axis.

```
> coordinates([1,2])
```

$$\begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix}$$

26.12.5 Rectangular coordinates of a point

The `rectangular_coordinates` command finds the rectangular coordinates of a point given its polar coordinates.

- `rectangular_coordinates` takes two arguments: r and θ, two real numbers.
- `rectangular_coordinates`(r,θ) returns a list of the rectangular coordinates of the point with polar coordinates r and θ.

Example

```
> rectangular_coordinates(2,pi/4)
```

or:

```
> rectangular_coordinates(polar_point(2,pi/4))
```

$$\left[\frac{2}{2}\sqrt{2},\frac{2}{2}\sqrt{2}\right]$$

26.12.6 Polar coordinates of a point

The `polar_coordinates` command finds the polar coordinates of a point.

- `polar_coordinates` takes P, a point.
- `polar_coordinates`(P) returns a list of the polar coordinates of P.

Example

```
> polar_coordinates(1+i)
```

or:

```
> polar_coordinates(point(1+i))
```

or:

```
> polar_coordinates([1,1])
```

$$\left[\sqrt{2},\frac{\pi}{4}\right]$$

26.12.7 Cartesian equation of a geometric object in the plane

See Section 27.10.5, p. 784 for Cartesian equations of 3D objects.

The `equation` command finds the Cartesian equation for a geometric object.

- `equation` takes G, a geometric object.
- `equation(`G`)` returns the Cartesian equation in the variables `x` and `y` for G.

Note that `x` and `y` must be formal variables, you might need to purge them with with `purge(x,y)`.

Example

```
> equation(line(-1,i))
```

$$y = x + 1$$

26.12.8 Parametric equation of a geometric object in the plane

See Section 27.10.6, p. 785 for parametric equations in 3D geometry.

The `parameq` command finds a parametric equation for a curve.

- `parameq` takes C, a curve.
- `parameq(`C`)` returns a parametric equation for C, in the form x`(t)+i`y`(t)`.

Note that `t` must be a formal variable, it may be necessary to purge it with `purge(t)`.

Examples

```
> parameq(line(-1,i))
```

$$(1 + \mathrm{i})\, t - 1$$

```
> parameq(circle(-1,i))
```

$$-1 + \mathrm{i}e^{\mathrm{i}t}$$

```
> normal(parameq(ellipse(-1,1,i)))
```

$$\frac{-2\mathrm{i}t^2 - 4t + \mathrm{i}}{2t^2 + 1}$$

26.13 Measurements

26.13.1 Measurement and display

Many commands to find measures have a version ending in `at` (or `atraw`) which are used to interactively find and display the measure in a 2D geometry screen. To use them, open a geometry screen with Alt+G and select a measure from the Mode ▶ Measure menu. Now, clicking on the names of objects (or, if a point is being selected, a name will be automatically generated if clicking on an open point) with the mouse and then clicking on another point will put the measurement at that point; if the mode is the version ending in `at` resp. `atraw`, then the measurement will have a label resp. appear without a label.

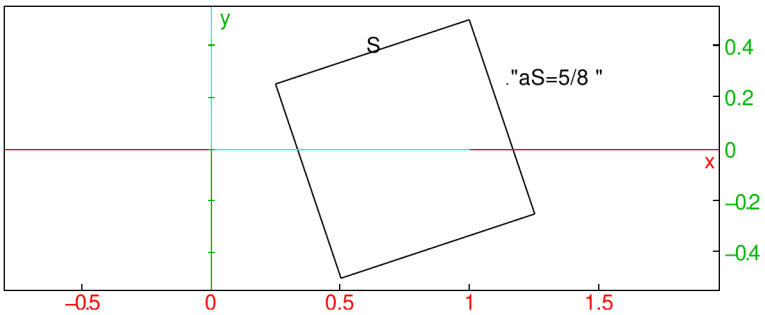
The commands with `at` and `atraw` versions are the following:

- `distance`, `distanceat`, `distanceatraw` — This finds the distance between two points or other geometric objects (see Section 26.13.2, p. 726).
- `angle`, `angleat`, `angleatraw` — This finds the measure of an angle BAC given points A, B and C (see Section 26.13.4, p. 727).
- `area`, `areaat`, `areaatraw` — This finds the area of a circle or a polygon which is star-shaped with respect to its first vertex (see Section 26.13.6, p. 729).
- `perimeter`, `perimeterat`, `perimeteratraw` — This finds the perimeter of a circle, circular arc or a polygon (see Section 26.13.7, p. 729).
- `slope`, `slopeat`, `slopeatraw` — This finds the slope of a line, segment, or two points which determine a line (see Section 26.13.8, p. 730).

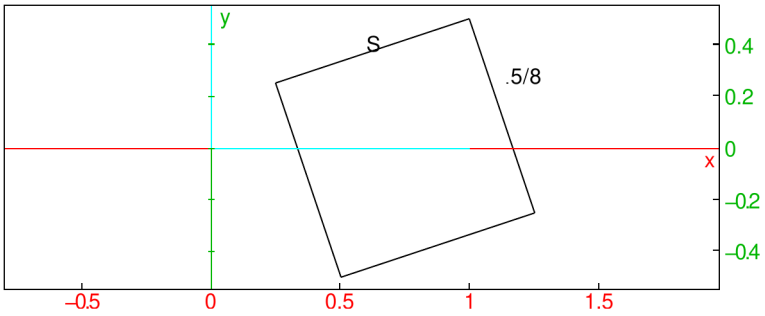
These commands can also be used from the command line. They are like the measurement command but take an extra argument, the point to display the measurement. When using the version ending in `at`, use names for the objects rather to create the objects within the measurement command.

Examples

```
> S:=square(1/4+i/4,1/2-i/2); areaat(S,8/7+i/4)
```

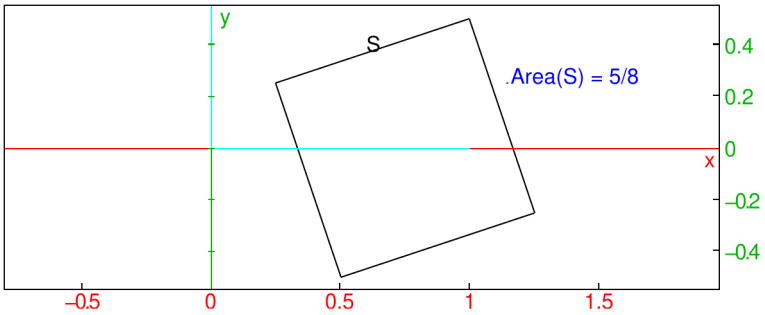



```
> S:=square(1/4+i/4,1/2-i/2); areaatraw(S,8/7+i/4)
```



More sophisticated legends are created with the `legend` command (see Section 26.3.3, p. 680).

```
> S:=square(1/4+i/4,1/2-i/2); a:=area(S); legend(8/7+i/4,"Area(S)="+string(a),blue)
```



The `extract_measure` command displays a measurement.

- `extract_measure` takes *atcommand*, one of the `at` or `atraw` commands (which displays a measurement).
- `extract_measure(`*atcommand*`)` returns the measurement.

Example

```
> A:=point(-1); B:=point(1+i); C:=point(i);
  extract_measure(angleat(A,B,C,0.2i))
```

$$\arctan\left(\frac{1}{3}\right)$$

26.13.2 Distance between objects in the plane

See Section 27.10.7, p. 785 for distances in 3D geometry.

The `distance` command finds the distance between two geometric objects (a point is considered a geometric object).

- `distance` two arguments: G_1, G_2, two geometric objects.
- `distance(`G_1, G_2`)` returns the distance between G_1 and G_2.

Examples

```
> distance(-1,1+i)
```

$$\sqrt{5}$$

```
> distance(0,line(-1,1+i))
```

$$\frac{\sqrt{5}}{5}$$

```
> distance(circle(0,1),line(-2,1+3*i))
```

$$\sqrt{2}-1$$

Note that when the distance calculation uses parameters, XCAS must be in real mode.

Example

In real mode:

```
> assumes(a=[4,0,5,0.1]); A:=point(0); B:=point(a);
  simplify(distance(A,B)); simplify(distance(B,A))
```

$$|a|\,,|a|$$

In complex mode:

```
> assumes(a=[4,0,5,0.1]); A:=point(0); B:=point(a);
  simplify(distance(A,B)); simplify(distance(B,A))
```

$$-a, a$$

The `distance` command has `distanceat` and `distanceatraw` versions (see Section 26.13.1, p. 724).

26.13.3 Squared length of a segment in the plane

See Section 27.10.8, p. 785 for squares of lengths in 3D geometry.

The `distance2` command finds the square of the distance between two points.

- `distance2` takes two arguments: P, Q, two points.
- `distance2`(P, Q) returns the square of the distance between P and Q.

Example

```
> distance2(-1, 1+i)
```

$$5$$

26.13.4 Measure of an angle in the plane

See Section 27.10.9, p. 786 for angle measures in 3D geometry.

The `angle` command finds the measure of an angle.

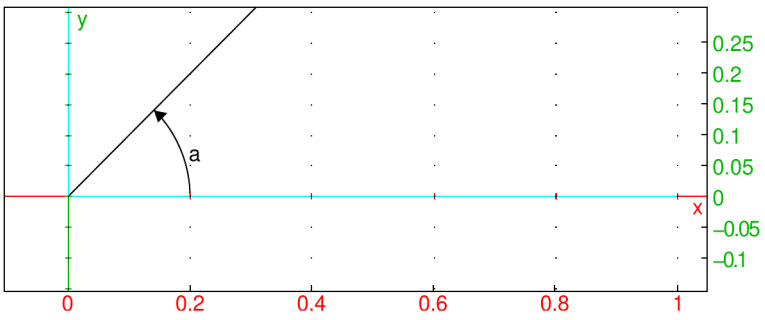
- `angle` takes three mandatory arguments and one optional argument:
  - A, B, C, three points.
  - *str*, a string.
- `angle`$(A, B, C\,\langle, str\rangle)$ returns the measure of angle ABC (in the units that Xcas is configured for). With the argument *str*, the angle will be drawn indicated by a small arc and labeled with the string. If the angle is a right angle, the indicator will be a corner rather than an arc.

Examples

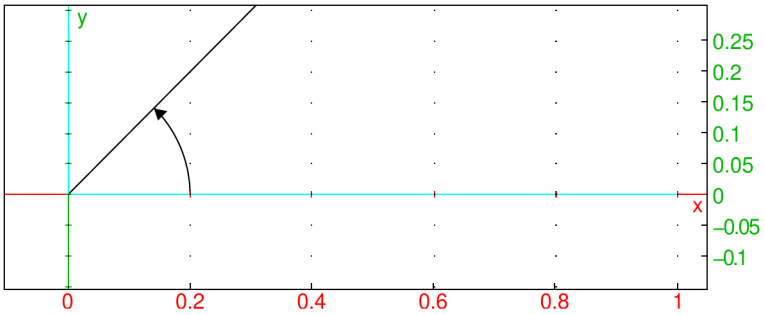
```
> angle(0,1,1+i)
```

$$\frac{1}{4}\pi$$

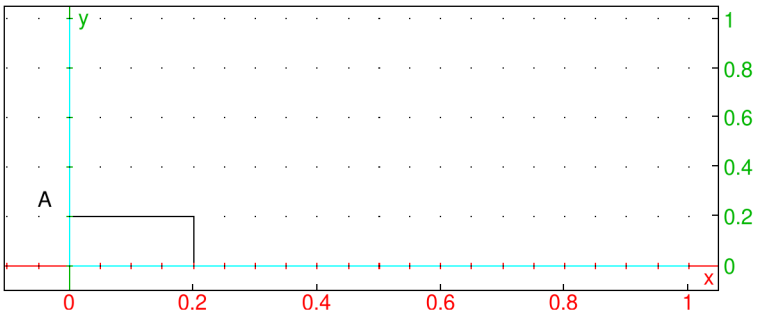
```
> angle(0,1,1+i,"a")
```




```
> angle(0,1,1+i,"")
```




```
> angle(0,1,i,"A")
```




```
> angle(0,1,i,"A")[0]
```

$$\frac{1}{2}\pi$$

The `angle` command has `angleat` and `angleatraw` versions (see Section 26.13.1, p. 724). For the command line versions of these commands, the optional fourth argument for `angle` is replaced by a mandatory fourth argument for the point to put the measurement.

Converting between radians and degrees. You can use `radians` and `degrees` commands to convert from degrees to radians and vice versa, respectively. These commands return inexact values.

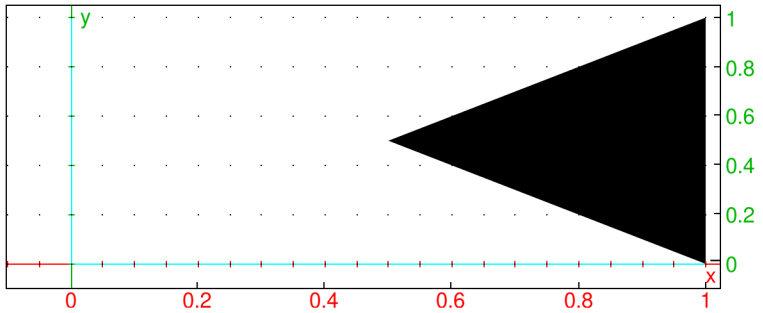
26.13.5 Graphical representation of the area of a polygon

The `plotarea` or `areaplot` command finds the (signed) area of a polygon.

- `plotarea` takes P, a polygon.
- `plotarea(`P`)` draws the filled polygon, with the signed area. (The area is positive if the polygon is counterclockwise, negative if it is clockwise.)

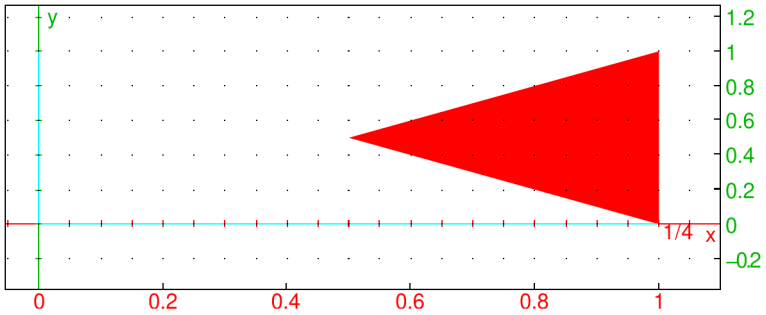
Examples

```
> plotarea(polygon(1,(1+i)/2,1+i))
```



The fill color can be changed as a local feature (see 26.3.2) and the position of the legend can be changed (see Section 26.3.3, p. 680).

```
> plotarea(polygon(1,1+i,(1+i)/2),display=red+quadrant2)
```



26.13.6 Area of a polygon

The `area` command finds the area of a circle or a star-shaped polygon.

- `area` takes P, a circle or polygon which is star-shaped with respect to its first vertex (i.e., the line segment from the first vertex to any point in the polygon lies within the polygon).
- `area(`P`)` returns the area of P.

Examples

```
> area(triangle(0,1,i))
```

$$\frac{1}{2}$$

```
> area(square(0,2))
```

$$4$$

The `area` command has `areaat` and `areaatraw` versions (see Section 26.13.1, p. 724).

26.13.7 Perimeter of a polygon

See also `arcLen`, Section 13.3.8, p. 290.

The `perimeter` command finds the length of a circle, circular arc or polygon.

- `perimeter` takes C, a circle, circular arc or a polygon.
- `perimeter(`C`)` returns the perimeter of the object.

Examples

```
> perimeter(circle(0,1))
```

$$2\pi$$

```
> perimeter(circle(0,1,pi/4,pi))
```

$$\frac{3}{4}\pi$$

```
> perimeter(arc(0,pi/4,pi))
```

$$\frac{1}{8}\pi^2$$

```
> perimeter(triangle(0,1,i))
```

$$\sqrt{2}+2$$

```
> perimeter(square(0,2))
```

$$8$$

The `perimeter` command has `perimeterat` and `perimeteratraw` versions (see Section 26.13.1, p. 724).

26.13.8 Slope of a line

The `slope` command finds the slope of a line.

- `slope` takes one or two arguments: L, a line, a line segment, or two points determining a line.
- `slope(`L`)` returns the slope of the line.

Examples

```
> slope(line(1,2i))
```

or:

```
> slope(segment(1,2i))
```

or:

```
> slope(1,2i)
```

$$-2$$

```
> slope(line(x-2y=3))
```

$$\frac{1}{2}$$

```
> slope(tangent(plotfunc(sin(x)),pi/4))
```

or:

```
> slope(LineTan(sin(x),pi/4))
```

$$\frac{\sqrt{2}}{2}$$

The `slope` command has `slopeat` and `slopeatraw` versions (see Section 26.13.1, p. 724).

26.13.9 Radius of a circle

The `radius` command finds the radius of a circle.

- `radius` takes C, a circle.
- `radius(`C`)` returns the radius of C.

Example

```
> radius(circle(-1,point(i)))
```

$$\frac{1}{\sqrt{2}}$$

26.13.10 Length of a vector

The `abs` command finds the absolute value of a number or the length of a vector (see also Section 8.3.2, p. 155 and Section 7.4.4, p. 141).

- `abs` takes v, a number or a vector defined by two points.
- `abs(`v`)` returns the absolute value of v if v is a complex number or the length of v if v is a vector.

Example

```
> abs(1+i)
```

or:

```
> abs(point(1+2*i)-point(i))
```

$$\sqrt{2}$$

26.13.11 Angle of a vector

The `arg` command finds the angle of a complex number (the argument) or the angle of a vector defined by two points.

- `arg` takes v, a number or a vector defined by two points.
- `arg(`v`)` returns the argument of v if v is a complex number or the angle between the positive x direction and v if v is a vector.

Example

```
> arg(1+i)
```

$$\frac{\pi}{4}$$

26.13.12 Normalize a complex number

The `normalize` command normalizes a non-zero complex number; i.e., it finds a complex number with the same argument and absolute value 1.

- `normalize` takes c, a non-zero complex number.
- `normalize(`c`)` returns the normalized version of c; namely, $c/|c|$.

Example

```
> normalize(3+4*i)
```

$$\frac{3+4i}{5}$$

26.14 Transformations

26.14.1 General remarks

The transformations in this section operate on any geometric object. As arguments, they can take the parameters which specify the transformation. With those arguments, they will return a new command which performs the transformation. If they are given a geometric object as the final argument, they will return the transformed object.

26.14.2 Translations in the plane

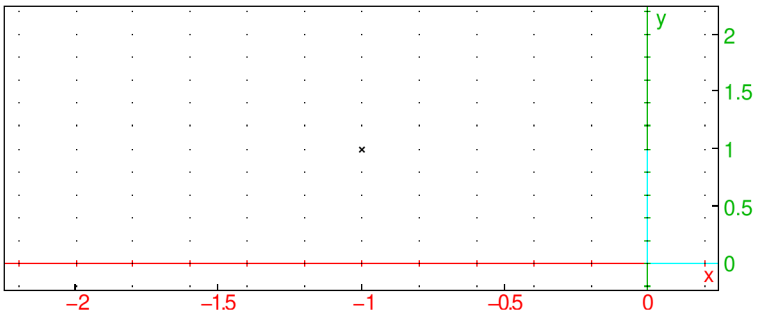
See Section 27.12.2, p. 793 for translations in space.

The `translation` command creates a translation.

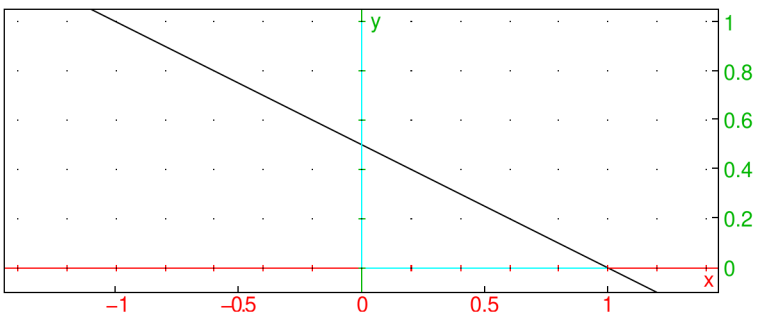
- `translation` takes one mandatory argument and one optional argument:
  - v, the translation vector, which can be given as a vector, a list of coordinates, a difference of points or a complex number.
  - Optionally, G, a geometric object.
- `translation(`v`)` returns a new command which translates by v.
- `translation(`v, G`)` returns and draws the translation G by the vector v.

Examples

```
> t:=translation(1+i); t(-2)
```




```
> translation([1,1],line(-2,-i))
```



26.14.3 Reflections in the plane

See Section 27.12.3, p. 794 for reflections in space.
The `reflection` command creates a reflection.

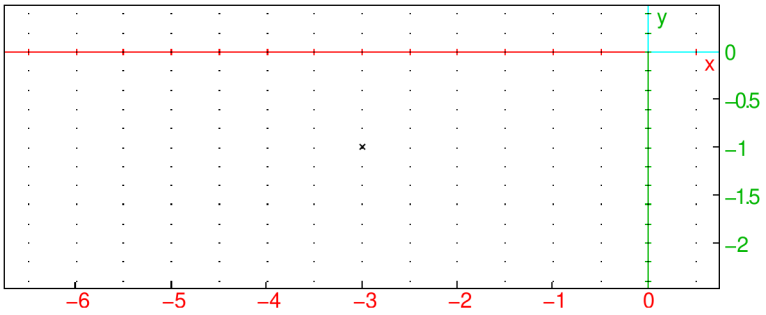
- `reflection` takes one mandatory argument and one optional argument:
  - P, a point or a line.
  - Optionally, G, a geometric object.
- `reflection(`P`)` returns a new command which reflects about P.
- `reflection(`P, G`)` returns and draws the reflection of G about P.

Examples

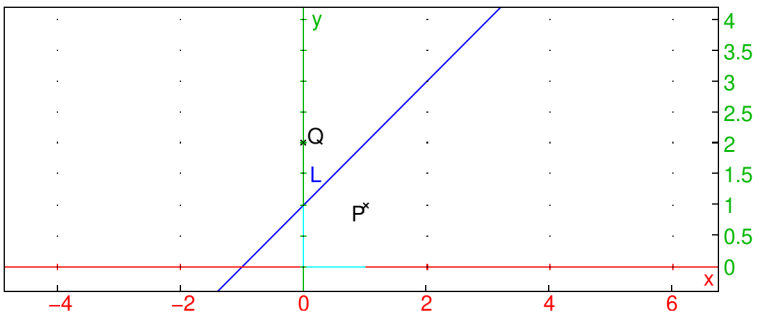
```
> rf:=reflection(-1); rf(1+i)
```

or:

```
> reflection(-1,1+i)
```




```
> L:=line(-1,i,color=blue); P:=point(1+i); Q:=reflection(L,P)
```



26.14.4 Rotation in the plane

See Section 27.12.4, p. 795 for rotations in space.
The `rotation` command creates a rotation.

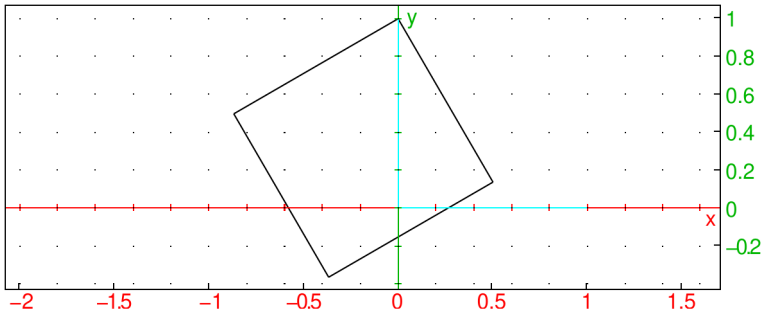
- `rotation` takes two mandatory arguments and one optional argument:
  - P, a point (the center of rotation).
  - θ, the angle of rotation.
  - Optionally, G, a geometric object.
- `rotation(`P, θ`)` returns a new command which rotations about P through an angle of θ.
- `reflection(`P, θ, G`)` returns and draws the rotation of G about P through an angle of θ.

Examples

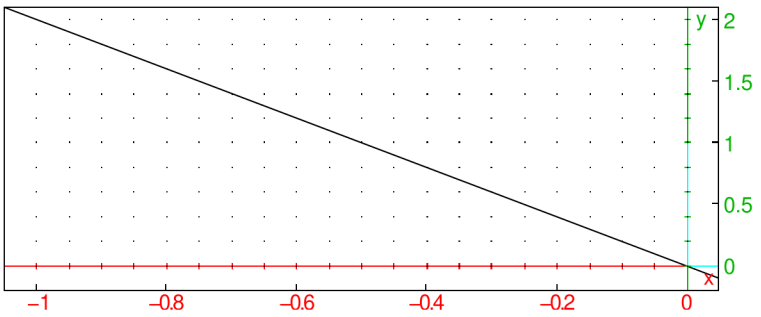
```
> r:=rotation(i,-pi/3); r(square(0,1)
```

or:

```
> rotation(i,-pi/3,square(0,1))
```




```
> rotation(i,-pi/2,line(1+i,-1))
```



26.14.5 Homothety in the plane

See Section 27.12.5, p. 796 for homotheties in space.

A homothety is a dilation about a given point. The `homothety` command creates a homothety.

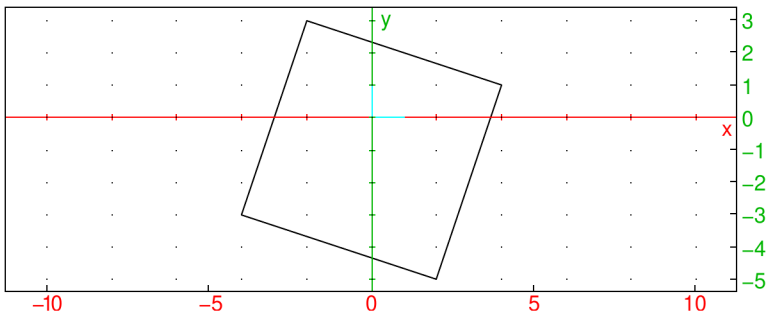
- `homothety` takes two mandatory arguments and one optional argument:
  - P, a point (the center of the homothety).
  - r, a number (the scaling ratio).
  - Optionally, G, a geometric object.
- `homothety(`P, r`)` returns a new command which dilates about P by a factor of r. If r is complex, this will rotate as well as scale.
- `homothety(`P, r, G`)` returns and draws the dilation of G about P by a factor or r.

Example

```
> h:=homothety(i,2); h(square(1-2i,2+i))
```

or:

```
> homothety(i,2,square(1-2i,2+i))
```



26.14.6 Similarity in the plane

See Section 27.12.6, p. 797 for similarities in space.

The `similarity` command creates a command to rotate and scale about a given point.

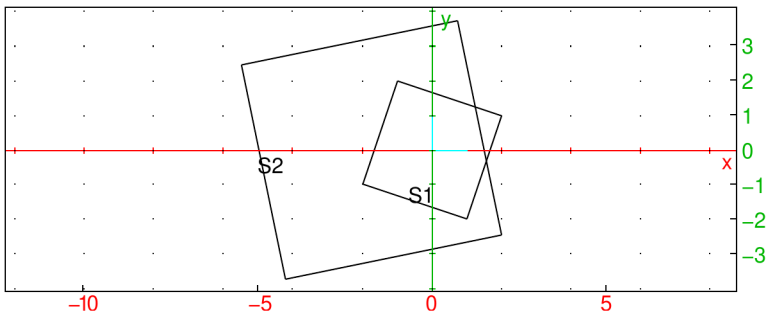
- `similarity` takes three mandatory arguments and one optional argument:
  - P, a point (the center of the rotation).
  - r, a real number (the scaling ratio).
  - θ, a real number (the angle of rotation).
  - Optionally, G, a geometric object.
- `similarity(`P, r, θ`)` returns a new command which rotates about P through an angle of θ and scales about P by a factor of r.
- `similarity(`P, r, θ, G`)` returns and draws the transformation of G.

Example

```
> s:=similarity(i,2,-pi/3); S1:=square(1-2i,2+i); S2:=s(S1)
```

or:

```
> S1:=square(1-2i,2+i); S2:=similarity(i,2,-pi/3,S1)
```



Note that for a point P and real numbers r and θ, the command `similarity(`P, r, θ`)` is the same as `homothety(`P`,`k`*exp(i*`a`))`.

26.14.7 Inversion in the plane

See Section 27.12.7, p. 798 for inversions in space.

Given a circle C with center P and radius r, the *inversion* of a point A with respect to C is the point A' on the ray $\overrightarrow{PA}$ satisfying $\overline{PA} \cdot \overline{PA'} = r^2$.

The `inversion` command creates an inversion.

- `inversion` takes two mandatory arguments and one optional argument:

– P, a point (the center of the inversion).
– r, a real number (the radius).
– Optionally, G, a geometric object.

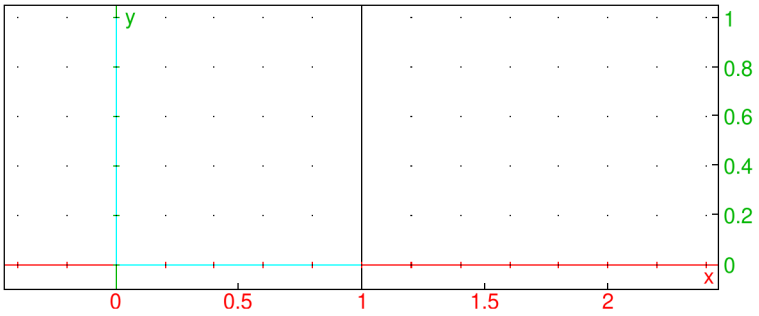
- `inversion(`P, r`)` returns a new command which performs the inversion.
- `inversion(`P, r, G`)` returns and draws the inversion of G.

Example

```
> inver:=inversion(i,2); inver(circle(1+i,1))
```

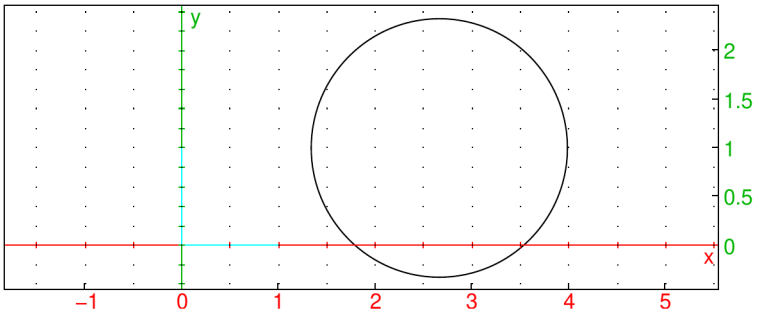
or:

```
> inversion(i,2,circle(1+i,1))
```



then:

```
> inver(circle(1+i,1/2))
```



26.14.8 Orthogonal projection in the plane

See Section 27.12.8, p. 799 for projections in space.

The `projection` command creates a projection.

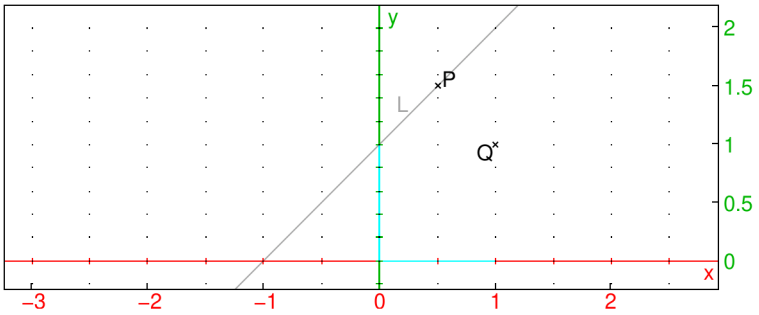
- `projection` takes one mandatory argument and one optional argument:
  – O, a geometric object.
  – Optionally, G, a geometric object.
- `projection(`O`)` returns a new command which projects points onto O.
- `projection(`O, G`)` returns and draws the projection of G onto O.

Examples

```
> L:=line(-1,i,color=grey); p1:=projection(L);
  Q:=point(i+1); P:=p1(Q)
```

or:

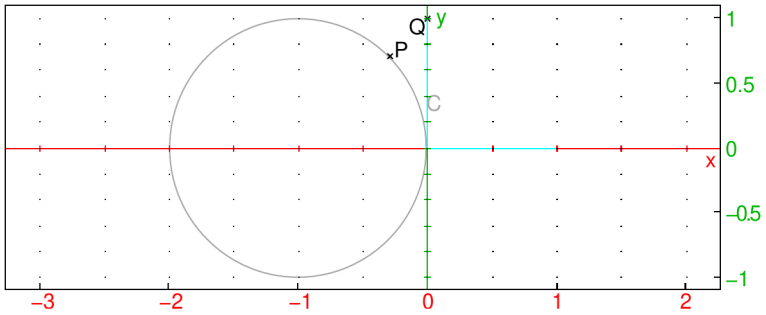
```
> L:=line(-1,i,color=grey); Q:=point(i+1); P:=projection(L,Q)
```




```
> C:=circle(-1,1,color=grey); p2:=projection(C);
  Q:=point(i); P:=p2(i)
```

or:

```
> C:=circle(-1,1,color=grey); Q:=point(i); P:=projection(C,i)
```



26.15 Properties

26.15.1 Checking whether a point is on an object in the plane

See Section 27.11.1, p. 787 for checking elements in 3D geometry.

The `is_element` command determines whether or not a point lies on a geometric object.

- `is_element` takes two arguments:
  - P, a point.
  - G, a geometric object.
- `is_element`(P, G) returns 1 if P is an element of G and 0 otherwise.

Examples

```
> is_element(-1-i,line(0,1+i))
```

1

```
> is_element(i,line(0,1+i))
```

0

26.15.2 Checking whether three points are collinear in the plane

See Section 27.11.6, p. 789 for checking for collinearity in 3D geometry.
The `is_collinear` command determines whether or not three points are collinear.

- `is_collinear` takes three arguments: A, B, C, three points.
- `is_collinear`(A, B, C) returns 1 if A, B and C are collinear and 0 otherwise.

Examples

```
> is_collinear(0,1+i,-1-i)
```

1

```
> is_collinear(i/100,1+i,-1-i)
```

0

26.15.3 Checking whether four points are concyclic in the plane

See Section 27.11.7, p. 789 for checking for concyclicity in 3D geometry.
The `is_concyclic` command determines whether or not points are cyclic.

- `is_concyclic` takes L, a list or sequence of points.
- `is_concyclic`(L) returns 1 if the points in L all lie on the same circle, and 0 otherwise.

Examples

```
> is_concyclic(1+i,-1+i,-1-i,1-i)
```

1

```
> is_concyclic(i,-1+i,-1-i,1-i)
```

0

26.15.4 Checking whether a point is in a polygon or circle

The `is_inside` command determines whether or not a point is in a polygon or a circle.

- `is_inside` takes two arguments:
  - P, a point.
  - C, a polygon or a circle.
- `is_inside`(P, C) returns 1 if P is inside C (including the boundary) and 0 otherwise.

Examples

```
> is_inside(0,circle(-1,1))
```

$$1$$

```
> is_inside(2,polygon([1,2-i,3+i]))
```

$$1$$

```
> is_inside(1-i,triangle([1,2-i,3+i]))
```

$$0$$

26.15.5 Checking whether an object is an equilateral triangle in the plane

See Section 27.11.9, p. 790 for checking for equilateral triangles in 3D geometry.

The `is_equilateral` command determines whether or not a geometric object is an equilateral triangle.

- `is_equilateral` takes G, a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_equilateral`(G) returns 1 if the object is an equilateral triangle and 0 otherwise.

Examples

```
> is_equilateral(0,2,1+i*sqrt(3))
```

$$1$$

```
> T:=equilateral_triangle(0,2,C); is_equilateral(T[0])
```

$$1$$

Note that `T[0]` is a triangle since `T` is a list made of a triangle and the vertex `C`.

```
> affix(C)
```

$$\sqrt{3}\mathrm{i}+1$$

```
> is_equilateral(1+i,-1+i,-1-i)
```

$$0$$

26.15.6 Checking whether an object in the plane is an isosceles triangle

See Section 27.11.10, p. 790 for checking for isosceles triangles in 3D geometry.

The `is_isosceles` command determines whether or not a geometric object is an isoceles triangle.

- `is_isosceles` takes G, a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_isosceles`(G) returns $1, 2$ or 3 if the object is an isoceles triangle (the number indicates which vertex is on two equal sides), returns 4 if the object is an equilateral triangle, and returns 0 otherwise.

Examples

```
> is_isosceles(0,1+i,i)
```

$$2$$

```
> T:=isosceles_triangle(0,1,pi/4); is_isosceles(T)
```

$$1$$

```
> T:=isosceles_triangle(0,1,pi/4,C); is_isosceles(T[0])
```

$$1$$

Note that `T[0]` is a triangle since `T` is a list made of a triangle and the vertex `C`.

```
> affix(C)
```

$$\frac{\sqrt{2}}{2}+\frac{\sqrt{2}}{2}\mathrm{i}$$

```
> is_isosceles(1+i,-1+i,-i)
```

$$3$$

```
> is_isosceles(0,2,1+i*sqrt(3))
```

$$4$$

26.15.7 Checking whether an object in the plane is a right triangle or a rectangle

See Section 27.11.11, p. 791 for checking for right triangles and rectangles in 3D geometry.

The `is_rectangle` command determines whether or not a geometric object is a rectangle or right triangle.

- `is_rectangle` takes G, a geometric object or a sequence of three or four points assumed to be the vertices of a triangle or a quadrilateral.
- `is_rectangle`(G) returns:
  - (for triangle G) 1, 2 or 3 if G is a right triangle (the number indicates which vertex has the right angle).
  - (for quadrilateral G) 1 if G is a rectangle but not a square.
  - (for quadrilateral G) 2 if G is square.
  - 0 otherwise.

Examples

```
> is_rectangle(1,1+i,i)
```

$$2$$

```
> is_rectangle(1+i,-2+i,-2-i,1-i)
```

$$1$$

```
> R:=rectangle(-2-i,1-i,3,C,D); is_rectangle(R[0])
```

$$1$$

Note that `R[0]` is a rectangle since `R` is a list made of a rectangle and vertices `C` and `D`.

```
> affix(C,D)
```

$$-2+8\mathrm{i}, 1+8\mathrm{i}$$

26.15.8 Checking whether an object in the plane is a square

See Section 27.11.12, p. 791 for checking for squares in 3D geometry.

The `is_square` command determines whether or not a geometric object is a square.

- `is_square` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square`(G) returns 1 if the object is a square and returns 0 otherwise.

Examples

```
> is_square(1+i,-1+i,-1-i,1-i)
```

$$1$$

```
> K:=square(1+i,-1+i); is_square(K)
```

$$1$$

```
> K:=square(1+i,-1+i,C,D); is_square(K[0])
```

$$1$$

Note that `K[0]` is a square since `K` is a list made of a square and vertices `C` and `D`.

```
> affix(C,D)
```

$$-1-\mathrm{i}, 1-\mathrm{i}$$

```
> is_square(i,-1+i,-1-i,1-i)
```

$$0$$

26.15.9 Checking whether an object in the plane is a rhombus

See Section 27.11.13, p. 792 for checking for rhombuses in 3D geometry.

The `is_rhombus` command determines whether or not a geometric object is a rhombus.

- `is_rhombus` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square`(G) returns 1 if G is a rhombus but not a square, returns 2 if G is a square and returns 0 otherwise.

Examples

```
> is_rhombus(1+i,-1+i,-1-i,1-i)
```

$$1$$

```
> K:=rhombus(1+i,-1+i,pi/4); is_rhombus(K)
```

$$1$$

```
> K:=rhombus(1+i,-1+i,pi/4,C,D); is_rhombus(K[0])
```

$$1$$

Note that `K[0]` is a rhombus since `K` is a list made of a rhombus and vertices `C` and `D`.

```
> affix(C,D)
```

$$-\sqrt{2}-\mathrm{i}, -\sqrt{2}+\mathrm{i}$$

```
> is_rhombus(i,-1+i,-1-i,1-i)
```

$$0$$

26.15.10 Checking whether an object in the plane is a parallelogram

See Section 27.11.14, p. 792 for checking for parallelograms in 3D geometry.

The `is_parallelogram` command determines whether or not an object is a parallelogram.

- `is_parallelogram` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_parallelogram(`G`)` returns 1 if G is a parallelogram, but not a rhombus or a rectangle, returns 2 if G is a rhombus but not a rectangle, returns 3 if G is a rectangle but not a square, returns 4 is G is a square, and returns 0 otherwise.

Examples

```
> is_parallelogram(i,-1+i,-1-i,1-i)
```

$$0$$

```
> is_parallelogram(1+i,-1+i,-1-i,1-i)
```

$$1$$

```
> Q:=quadrilateral(1+i,-1+i,-1-i,1-i); is_parallelogram(Q)
```

$$4$$

```
> P:=parallelogram(-1-i,1-i,i,D); is_parallelogram(P[0])
```

$$1$$

Note that `P[0]` is a parallelogram since `P` is a list made of a parallelogram and vertex `D`.

```
> affix(D)
```

$$-2+\mathrm{i}$$

26.15.11 Checking whether two lines in the plane are parallel

See Section 27.11.3, p. 787 for checking for parallels in 3D geometry.
The `is_parallel` command determines whether or not two lines are parallel.

- `is_parallel` takes two arguments: L_1, L_2, two lines.
- `is_parallel`(L_1, L_2) returns 1 if L_1 and L_2 are parallel and otherwise returns 0.

Examples

```
> is_parallel(line(0,1+i),line(i,-1))
```

1

```
> is_parallel(line(0,1+i),line(i,-1-i))
```

0

26.15.12 Checking whether two lines in the plane are perpendicular

See Section 27.11.4, p. 788 for checking for perpendicularity in 3D geometry.
The `is_perpendicular` command determines whether or not two lines are perpendicular.

- `is_perpendicular` takes two arguments: L_1, L_2, two lines.
- `is_perpendicular`(L_1, L_2) returns 1 if L_1 and L_2 are perpendicular and otherwise returns 0.

Examples

```
> is_perpendicular(line(0,1+i),line(i,1))
```

1

```
> is_parallel(line(0,1+i),line(1+i,1))
```

0

26.15.13 Checking whether two circles in the plane are orthogonal

See Section 27.11.5, p. 788 for checking for orthogonality in 3D geometry.
The `is_orthogonal` command determines whether or not two lines or circles are orthogonal.

- `is_orthogonal` takes two arguments: C_1, C_2, two objects, both lines or both circles.
- `is_orthogonal`(C_1, C_2) returns 1 if C_1 and C_2 are orthogonal and returns 0 otherwise.

Examples

```
> is_orthogonal(line(0,1+i),line(i,1))
```

1

```
> is_orthogonal(line(2,i),line(0,1+i))
```

0

```
> is_orthogonal(circle(0,1+i),circle(2,1+i))
```

1

```
> is_orthogonal(circle(0,1),circle(2,1))
```

0

26.15.14 Checking whether elements are conjugates

The `is_conjugate` command determines whether or not two objects are conjugates of each other.

- To check for conjugates with respect to a circle, `is_conjugate` takes three arguments:
  - C, a circle.
  - P, Q, each of which is a point or a line.
- `is_conjugate`(C, P, Q) returns 1 if P and Q are conjugate with respect to C and 0 otherwise.
- To check for conjugates with respect to two points or two lines, `is_conjugate` takes three arguments:
  - L_1, L_2, two lines or two points.
  - P, Q, each of which is a point or a line.
- `is_conjugate`(L_1, L_2, P, Q) returns 1 if P and Q are conjugate with respect to L_1 and L_2, otherwise it returns 0.

Examples

```
> is_conjugate(circle(0,1+i),point(1-i),point(3+i))
```

1

```
> is_conjugate(circle(0,1),point((1+i)/2),line(1+i,2))
```

1

```
> is_conjugate(circle(0,1),line(1+i,2),line((1+i)/2,0))
```

1

```
> is_conjugate(point(1+i),point(3+i),point(i),point(3/2+i))
```

1

```
> is_conjugate(line(0,1+i),line(2,3+i),line(3,4+i),line(3/2,5/2+i))
```

1

26.15.15 Checking whether four points form a harmonic division

The `is_harmonic` command determines whether or not four points form a harmonic division.

- `is_harmonic` takes four arguments: P, Q, R, S, four points.
- `is_harmonic`(P, Q, R, S) returns 1 if P, Q, R and S form a harmonic range and 0 otherwise.

Examples

```
> is_harmonic(0,2,3/2,3)
```

1

```
> is_harmonic(0,1+i,1,i)
```

0

26.15.16 Checking whether lines form a bundle

The `is_harmonic_line_bundle` command determines how lines are related.

- `is_harmonic_line_bundle` takes L, a list or sequence of lines.
- `is_harmonic_line_bundle`(L) returns 1 if the lines pass through a common point. 2 if the lines are parallel, 3 if the lines are the same, and 0 otherwise.

Example

```
> is_harmonic_line_bundle([line(0,1+i),line(0,2+i),line(0,3+i),line(0,1)])
```

1

26.15.17 Checking whether circles form a bundle

The `is_harmonic_circle_bundle` command determines how circles are related.

- `is_harmonic_circle_bundle` takes L, a list or sequence of circles.
- `is_harmonic_circle_bundle`(L) returns 1 if the circles pass through a common point, 2 if the circles are concentric, 3 if the circles are the same, and 0 otherwise.

Example

```
> is_harmonic_circle_bundle([circle(0,i),circle(4,i),circle(0,1/2)])
```

1

26.16 Harmonic division

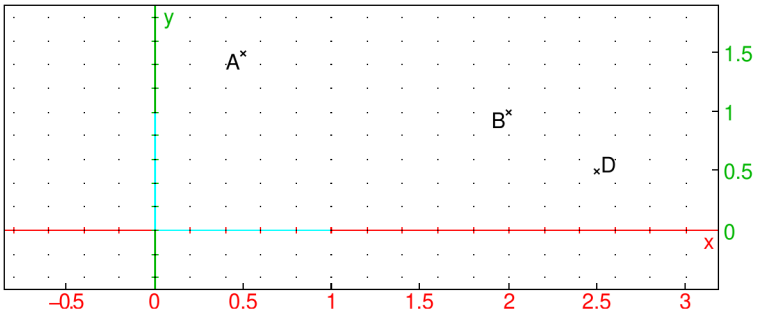
26.16.1 Finding a point dividing a segment in the harminic ratio k

The `division_point` command finds a point dividing a segment is a given ratio.

- `division_point` takes three arguments:
  - a, b, two complex numbers or points.
  - k, a complex number.
- `division_point(`a, b, k`)` returns and draws z where $(z-a)/(z-b) = k$.

Examples

```
> A:=point(1/2+3i/2); B:=point(2+i); D:=division_point(A,B,3+i)
```




```
> affix(D)
```

$$\frac{5}{2} + \frac{\mathrm{i}}{2}$$

26.16.2 Cross ratio of four collinear points

The cross ratio of four numbers a, b, c, d is $[(c-a)/(c-b)]/[(d-a)/(d-b)]$.
The `cross_ratio` command finds the cross ratio.

- `cross_ratio` takes four arguments: a, b, c, d, complex numbers.
- `cross_ratio(`a, b, c, d`)` returns the cross ratio, $[(c-a)/(c-b)]/[(d-a)/(d-b)]$.

Examples

```
> cross_ratio(0,1,2,3)
```

$$\frac{4}{3}$$

```
> cross_ratio(i,2+i,3/2+i, 3+i)
```

$$-1$$

26.16.3 Harmonic division

Four collinear points A, B, C and D are in harmonic division if $|CA|/|CB| = |DA|/|DB|$. In this case, D is called the harmonic conjugate of A, B and C.

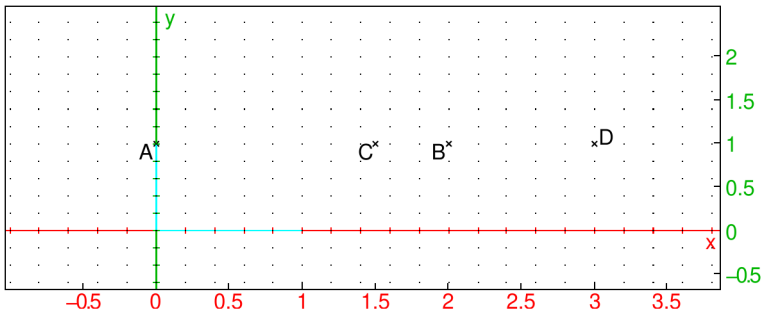
Four concurrent lines or four parallel lines are in harmonic division if the intersection of any fifth line with these four lines consists of four points in harmonic division. The lines are also said to form a *harmonic pencil*. The fourth line is called the harmonic conjugate of the first three.

The `harmonic_division` command finds harmonic conjugates.

- `harmonic_division` takes four arguments:
  - P_1, P_2, P_3, three collinear points or three concurrent lines.
  - *var*, a variable name.
- `harmonic_division`(P_1, P_2, P_3,*var*) returns and draws the three points or lines P_1, P_2 and P_3 together with their harmonic conjugate, and assigns the latter to *var*.

Examples

```
> A,B,C:=point(i),point(2+i),point(3/2+i); harmonic_division(A,B,C,D)
```




```
> affix(D)
```

$$3 + \mathrm{i}$$

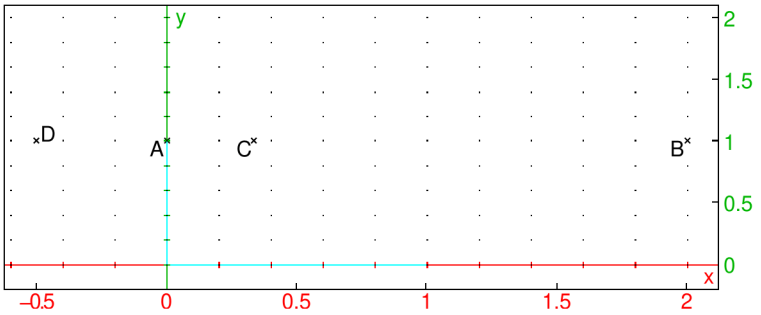
26.16.4 Harmonic conjugate

The `harmonic_conjugate` command finds harmonic conjugates.

- `harmonic_conjugate` takes three arguments: P_1, P_2, P_3, three collinear points or three concurrent lines, or three parallel lines.
- `harmonic_conjugate`(P_1, P_2, P_3) returns and draws the harmonic conjugate of P_1, P_2 and P_3.

Examples

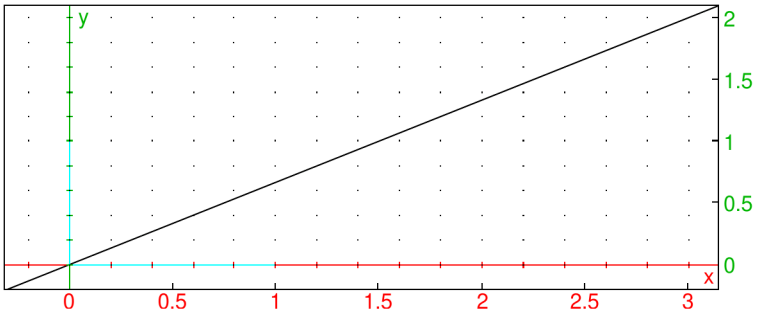
```
> A,B,C:=point(i),point(2+i),point(1/3+i); D:=harmonic_conjugate(A,B,C)
```




```
> evalc(affix(D))
```

$$-\frac{1}{2}+\mathrm{i}$$

```
> harmonic_conjugate(line(0,1+i),line(0,3+i),line(0,i))
```



26.16.5 Pole and polar

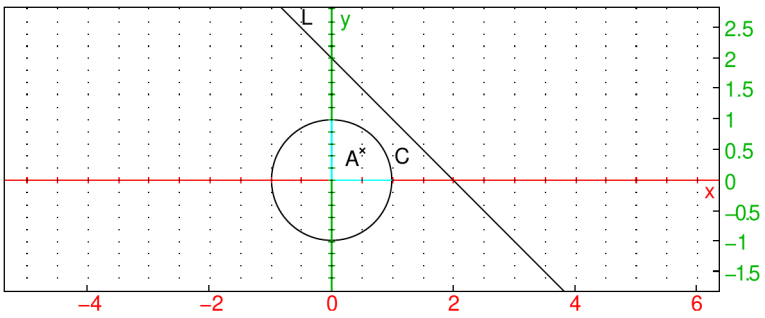
Given a circle centered at O, a point A is a pole and a line L is the corresponding polar if L is the line passing through the inversion of A with respect to the circle (see Section 26.14.7, p. 735) passing through the line $\overleftrightarrow{OA}$.

The `polar` command finds the polar of a point.

- `polar` takes two arguments:
  - C, a circle.
  - A, a point.
- `polar`(C, A) returns and draws the polar of the point A with respect to C.

Example

```
> C:=circle(0,1); A:=point((i+1)/2); L:=polar(C,A)
```

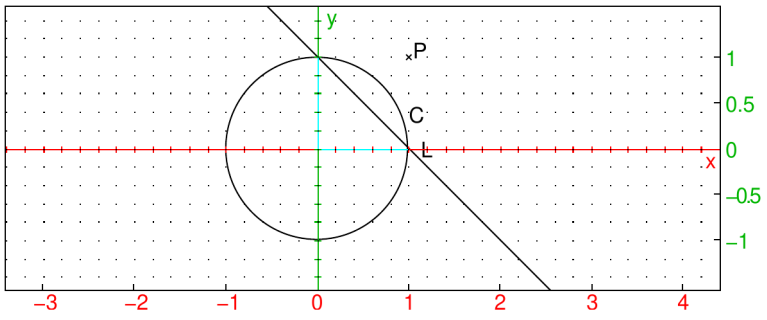


The `pole` command finds the pole of a line.

- `pole` takes two arguments:
  - C, a circle.
  - L, a line
- `pole`(C, L) returns and draws the pole of the line L with respect to C.

Examples

```
> C:=circle(0,1); L:=line(i,1); P:=pole(C,L)
```




```
> affix(P)
```

$$1 + i$$

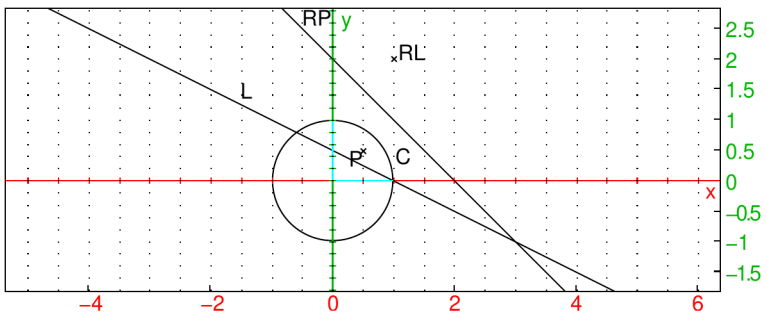
26.16.6 Polar reciprocal

The `reciprocation` command finds poles and polars.

- `reciprocation` takes two arguments:
  - C, a circle.
  - L, a list of points and lines.
- `reciprocation`(C, L) returns the list formed by replaced each point or line in L by its polar or pole with respect to the circle C.

Example

```
> C:=circle(0,1); P:=point((1+i)/2); L:=line(1,-1+i); RP,RL:=op(reciprocation(C,[P,L]))
```



26.17 Loci and envelopes

26.17.1 Loci

The `locus` command draws the locus of points determined by geometric objects moving in the plane, where the object depends on a point moving along a curve. It can draw a locus of points which depends on points on a curve, or the envelope of a family of lines depending on points on a curve.

The locus of points depending on points on a curve.

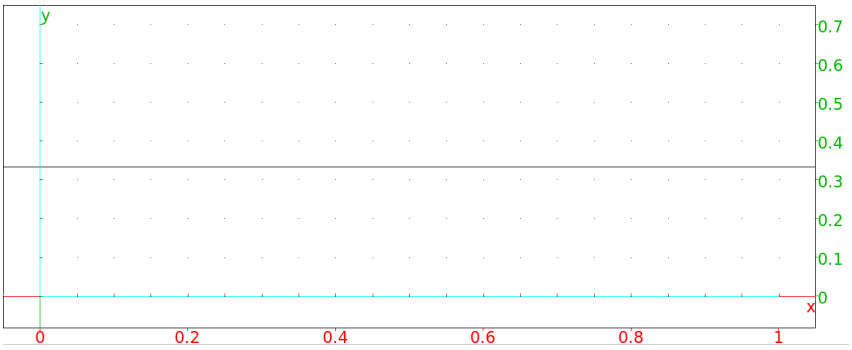
- For drawing the locus of points depending on points on a curve, `locus` takes two mandatory arguments and two optional arguments:
  - var_1, a variable name which has already been assigned to a point, which itself is a function of var_2, the second argument.
  - var_2, a variable name assigned to `element(`C`)` for some curve C (see Section 26.5.15, p. 692).
  - Optionally, $t = a..b$, where t is the parameter of the curve C. (You can double check the name of the parameter for a curve C with the command `parameq(`C`)`.)
  - Optionally, `tstep=`s, to set the step size for the parameter t.
- `locus(`var_1, var_2 ⟨`,tstep=`c⟩`)` draws the locus of points formed by var_1, as var_2 traces over the curve C.

  With the optional arguments, C is limited to the part parameterized from a to b with step size c.

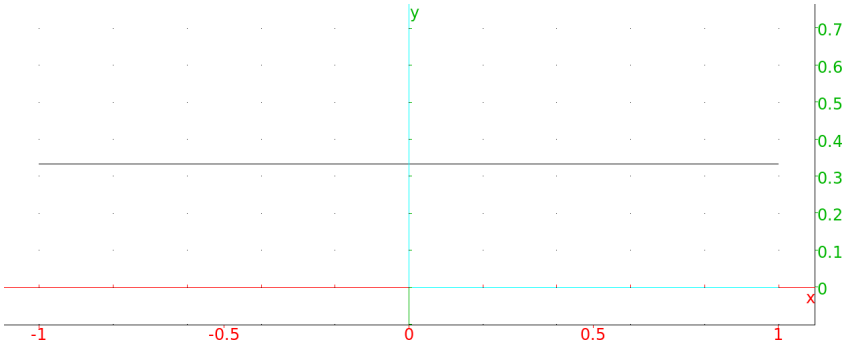
Example

This will draw the set of isobarycenters of the triangles with vertices `-1`, `1` and `P`, where `P` ranges over the line through `i` and `i+1`.

```
> P:=element(line(i,i+1)); G:=isobarycenter(-1,1,P);
  locus(G,P)
```




```
> locus(G,P,t=-3..3,tstep=0.1)
```



The envelope of a family of lines which depend on points on a curve. For drawing the envelope of a family of lines which depend on points on a curve:

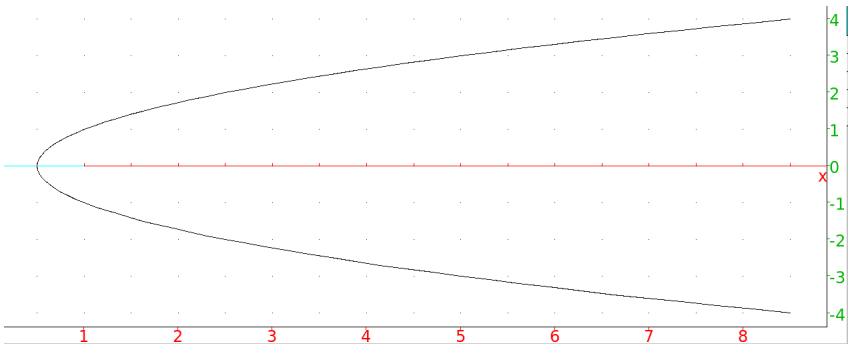
- `locus` takes two mandatory arguments and two optional arguments.
  - var_1, a variable name which has already been assigned to a line, which itself is a function of var_2, the second argument.
  - var_2, a variable name which has already been assigned to `element(`C`)` for some curve C (see Section 26.5.15, p. 692).
  - Optionally, $t = a..b$, where t is the parameter of the curve C. (You can double check the name of the parameter for a curve C with the command `parameq(`C`)`.)
  - Optionally, `tstep=`s, to set the step size for the parameter t.
- `locus(`var_1, var_2 ⟨`,tstep=`c⟩`)` draws the envelope of lines formed by var_1, as var_2 traces over the curve C.

  With the optional arguments, C is limited to the part parameterized from a to b with step size c.

Examples

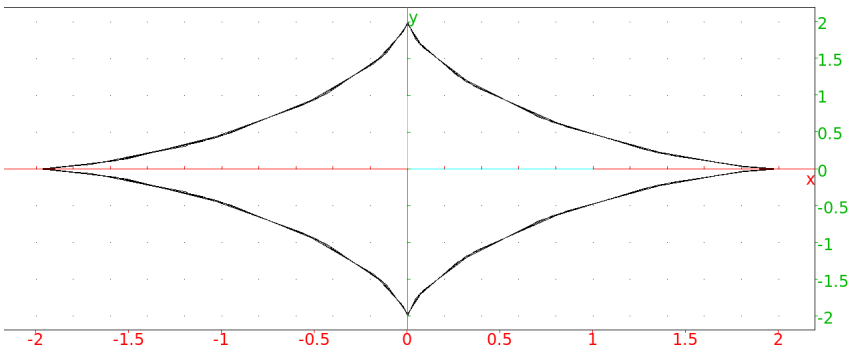
This will draw the envelope of the family of perpendicular bisectors of the segments from the point `1` to the points on the line `x=0`.

```
> F,H:=point(1),element(line(x=0));
  d:=perpen_bisector(F,H);
  locus(d,H)
```

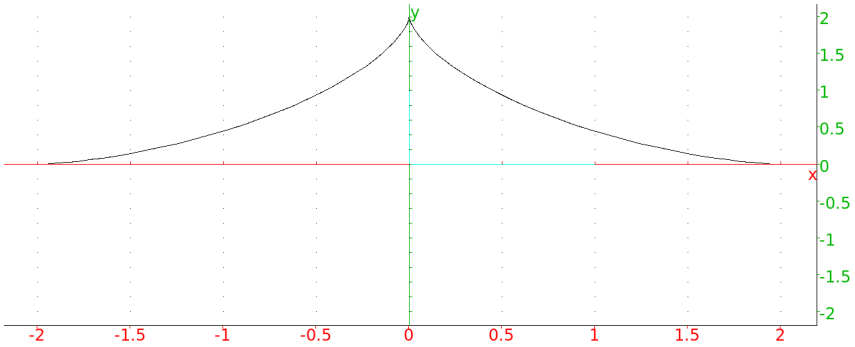


To draw the envelope of a family of lines which depend on a parameter, such as the lines given by the equations $y + x\tan(t) - 2\sin(t) = 0$ over the parameter t, the parameter can be regarded as the affixes of points on the line $y = 0$.

```
> H:=element(line(y=0));
  D:=line(y+x*tan(affix(H))-2*sin(affix(H)));
  locus(D,H)
```



```
> locus(D,H,t=0..pi)
```



26.17.2 Envelopes

The `envelope` command draws the envelope of a family of curves.

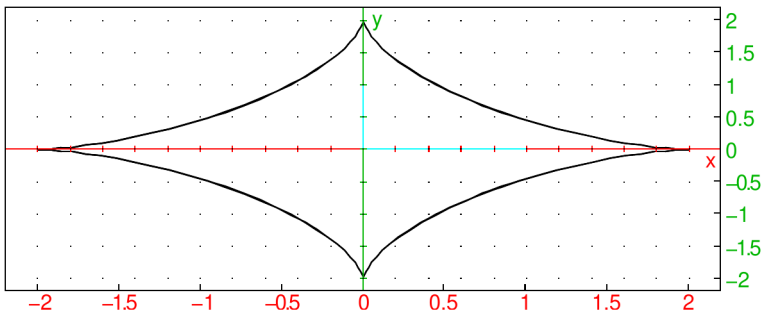
- `envelope` takes two arguments:
  - *expr*, an expression of two variables and one parameter.
  - L, a list of the names of the variables and the parameter. If the variables are `x` and `y`, then L only need to be the name of the parameter.
- `envelope(`*expr*$, L$`)` draws the envelope of the family of curves given by $expr = 0$.

Example

```
> envelope(y+x*tan(t)-2*sin(t),t)
```

or:

```
> envelope(v+u*tan(s)-2*sin(s),[u,v,s])
```



26.17.3 Trace of a geometric object

The `trace` command draws the trace of an object.

- `trace` takes G, a geometric object which depends on a parameter.
- `trace(`G`)` draws the trace of G as the parameter is changed or the object is moved in `Pointer` mode.

Example

For example, to find the locus of points equidistant from a line d and a point F, you can create a point H on the line d. To do this, open a graphic window (Alt+G) and type in the following commands.
First, create a line d (using sample points) and a sample point F.

```
> A,B:=point(-3-i),point(1/2+2*i);
  d:=line(A,B,color=0);
  F:=point(4/3,1/2,color=0)
```

Then create a point H on the line d which you can move around.

```
> assume(a=[0.7,-5,5,0.1]);
  H:=element(d,a)
```

To find a point equidistant from d and F, find the point `M` where the perpendicular to d (at H) intersects the perpendicular bisector of $\overline{HF}$, and trace that point.

```
> T:=perpendicular(H,d);
  M:=single_inter(perpen_bisector(H,F),T));
  trace(M)
```

Then as the point H on the line moves (by changing the value of a with the slider), you will get the trace of M.

To erase traces, add traces, activate or deactivate them, use the Trace menu of the M button located on the right side of the geometry screen.

27 3D graphics

27.1 Introduction

27.1.1 3D geometry graphics screen

The Alt+H command brings up a display screen for 3D graphics (see Figure 27.1). This screen has its own menu and command lines. This screen also automatically appears whenever there is a 3D graphic command.

The plane of vision for a 3D graphic screen is perpendicular to the observer's line of vision. The plane of vision is also indicated by dotted lines showing its intersection with the parallelepiped. The axis of vision for a 3D graphic screen is

The 3D graphic screen starts with the image of a parallelepiped bounding the graphics and vectors in the x, y and z directions. At the top of the screen is the equation of the plane of vision, which is a plane perpendicular to the observer's line of vision. The plane of vision is shown graphically with dotted lines indicating where it intersects the plane of vision.

Clicking in the graphic screen outside of the parallelepiped and dragging the mouse moves the x, y and z directions relative to the observer; these directions are also changed with the `x`, `X`, `y`, `Y`, `z` and `Z` keys. Scrolling the mouse wheel moves the plane of vision along the line of vision. The in and out buttons on the graphic screen menu zoom in and out of the picture.

The graphical features available for 2D graphics (see Section 26.3, p. 678) are also available for 3D graphics, but to see the points the markers must be squares with width (`point_width`) at least 3.

The graphic screen menu has a cfg button which brings up a configuration screen. Among other things, this screen has

- Ortho proj button, which determines whether the drawing uses orthogonal projection or perspective projection.
- Lights button, which determines whether the objects are lit or not; the locations of eight points for lighting are set using the buttons L1, ..., L7, which specify the points with homogeneous coordinates.
- Show axis button, which determines whether or not the outlining parallelepiped is visible.

27.1.2 Changing the view

The depictions of 3D objects are made with a coordinate system $Oxyz$, where the x axis is horizontal and directed right, the y axis is vertical and directed up, and the z axis is perpendicular to the screen and directed out of the screen. The depictions can be transformed by changing to a different coordinate system by setting a quaternion (see Section 26.3.2, p. 679).

27.2 The axes

27.2.1 Drawing unit vectors

The `Ox_3d_unit_vector` command takes no arguments and draws the unit vector in the x-direction on a 3D graphic screen.

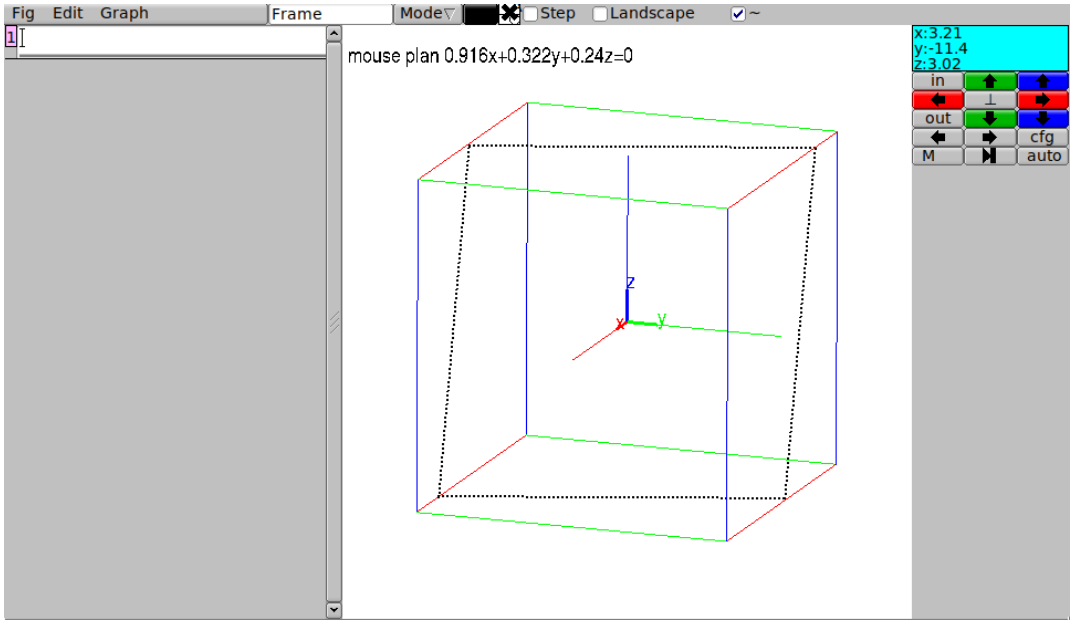
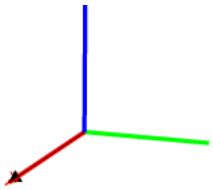


Figure 27.1: A 3D-geometry screen in XCAS

Example

```
> Ox_3d_unit_vector()
```

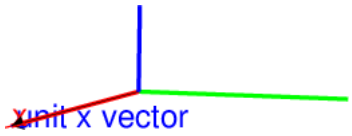


Similarly, the `Oy_3d_unit_vector` and `Oz_3d_unit_vector` commands draw the unit vector in the y and z directions, respectively.

These commands have no parameters, but can be decorated with the `legend` command.

Example

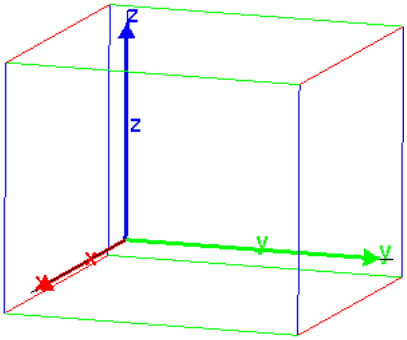
```
> Ox_3d_unit_vector(),legend(point([1,0,0]),"unit x vector",blue)
```



The `frame_3d` command draws all three vectors simultaneously.

Example

```
> frame_3d()
```



27.3 Points in space

27.3.1 Defining a point in three-dimensions

See Section 26.5.2, p. 685 for points in the plane.

With the 3D geometry screen in point mode, clicking on a point with the left mouse button will choose that point. Points chosen this way are automatically named, first with `A`, then `B`, etc.

Alternatively, the `point` command chooses a point.

- `point` takes one or three arguments: *coords*, where *coords* can be one of:
  - a, b, c, a sequence of three coordinates.
  - $[a, b, c]$, a list of three coordinates.
- `point(`*coords*`)` returns and draws the point with the given coordinates.

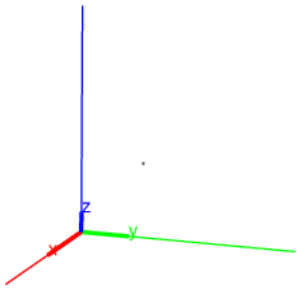
Many commands which takes points as arguments can either take them as `point(a,b,c)` or the list of coordinates `[a,b,c]`.

Example

```
> point(1,2,5)
```

or:

```
> point([1,2,5])
```



(The marker used to indicate the point can be changed; see Section 26.3.2, p. 678.)

27.3.2 Defining a random point in three-dimensions

The `point3d` command defines a random point whose coordinates are integers between -5 and 5.

- `point3d` takes *names*, a sequence of names for the points.

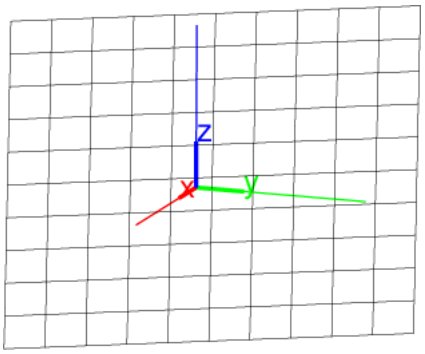
- `point3d(`*names*`)` assigns a random point whose coordinates are integers between −5 and 5 to each name.

Example

```
> point3d(A,B,C)
```

then:

```
> plane(A,B,C)
```



27.3.3 Finding an intersection point of two objects in space

See Section 26.5.6, p. 688 for single points of intersection of objects in the plane.

The `single_inter` or `line_inter` command finds an intersection point of two geometric objects.

- `single_inter` takes two mandatory arguments and one optional argument.
  - obj_1, obj_2, two geometric objects.
  - Optionally, pt, a point or list of points.

  `line_inter(`obj_1, obj_2 ⟨, pt⟩`)` returns one of the points of intersection of obj_1 and obj_2.

  If pt is a single point, then the command returns the point of intersection *closest* to pt.

  If pt is a list of points, then the command tries to return a point not in pt.

Examples

```
> A:=single_inter(plane(point(0,1,1),point(1,0,1),point(1,1,0)),
  line(point(0,0,0),point(1,1,1))):;
  coordinates(A)
```

$$\left[\frac{2}{3}, \frac{2}{3}, \frac{2}{3}\right]$$

```
> B:=single_inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1))):;
  coordinates(B)
```

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right]$$

```
> B1:=single_inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1)),point(-1,0,0)):;
  coordinates(B1)
```

$$\left[-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}\right]$$

```
> C:=single_inter(sphere(point(0,0,0),1),line(point(1,0,0),point(1,1,1))):;
  coordinates(C)
```

$$[1, 0, 0]$$

```
> C1:=single_inter(sphere(point(0,0,0),1),line(point(1,0,0),point(1,1,1)),[point(1,0,0)]):;
  coordinates(C1)
```

$$\left[\frac{1}{3}, \frac{2}{3}, \frac{2}{3}\right]$$

27.3.4 Finding the intersection points of two objects in space

See Section 26.5.7, p. 689 for points of intersection of objects in the plane.

The `inter` command finds the intersection of two geometric objects in $\mathbb{R}^3$.

- `inter` takes two mandatory arguments and one optional argument.
  - obj_1, obj_2, two geometric objects.
  - Optionally, P, a point.

  `inter(`obj_1`,`obj_2 $\langle$`,`$P\rangle$`)` returns a list of points of intersection of obj_1 and obj_2 or the curve of intersection of the two objects.

  With the argument P, the command returns the point of intersection *closest* to P.

Examples

```
> LA:=inter(plane(point(0,1,1),point(1,0,1),point(1,1,0)),line(point(0,0,0),point(1,1,1))):;
  coordinates(LA)
```

$$\left[\left[\frac{2}{3}, \frac{2}{3}, \frac{2}{3}\right]\right]$$

```
> LB:=inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1))):;
  coordinates(LB)
```

$$\left[\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right], \left[-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}\right]\right]$$

To get just one of the points, use the usual list indices.

```
> coordinates(LB[0])
```

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right]$$

To get the point closest to $(1/2, 1/2, 1/2)$:

```
> LB1:=inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1)),point(1/2,1/2,1/2)):;
  coordinates(LB1)
```

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}\right]$$

27.3.5 Finding the midpoint of a segment in space

See Section 26.5.9, p. 690 for midpoints in the plane.

The `midpoint` command finds the midpoint of two points.

- `midpoint` takes two arguments: P, Q, two points (which can also be given as a list).
- `midpoint`(P, Q) draws and returns the midpoint of the segment determined by these points.

Example

```
> MP:=midpoint(point(1,4,0),point(1,-2,0)):;
  coordinates(MP)
```

$$[1, 1, 0]$$

27.3.6 Finding the barycenter of a set of points in space

See Section 26.5.10, p. 690 for barycenters of objects in the plane.

The `barycenter` command returns and draws the barycenter of a set of weighted points.

- `barycenter` takes $L_1, L_2, \ldots, L_n$, a sequence of lists of length two, where each list consists of a point and a weight. This information can also be given as a matrix with two columns (the first column the points and the second column the weights) or a matrix with two rows and more than two columns.
- `barycenter`$(L_1, L_2, \ldots, L_n)$ draws and returns the barycenter of the weighted points.

If the sum of the weights is zero, then this command returns an error.

Examples

```
> BC:=barycenter([point(1,4,0),1],[point(1,-2,0),1])
```

or:

```
> BC:=barycenter([[point(1,4,0),1],[point(1,-2,0),1]])
```

then:

```
> coordinates(BC)
```

$$[1, 1, 0]$$

27.3.7 Finding the isobarycenter of a set of points in space

See Section 26.5.11, p. 691 for isobarycenters of objects in the plane.

The `isobarycenter` command finds the isobarycenter of a list of points; the isobarycenter is the barycenter when all points are equally weighted.

- `isobarycenter` takes L, a list of points. (The points can also be given by a sequence).
- `isobarycenter`(L) draws and returns the isobarycenter of the points.

Example

```
> IB:=isobarycenter(point(1,4,0),point(1,-2,0)):;
  coordinates(IB)
```

$$[1, 1, 0]$$

27.4 Lines in space

27.4.1 Lines and directed lines in space

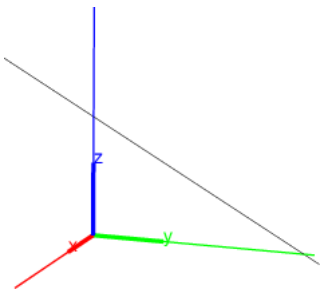
See Section 26.6.1, p. 693 for lines in the plane.

The `line` command returns and draws a directed line. It can take its arguments in three different ways.

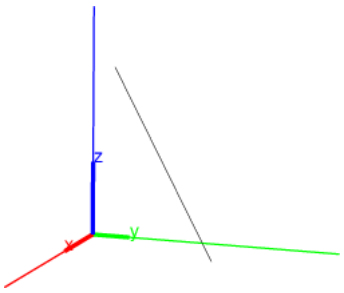
1. **Two points.**
   - `line` takes two arguments: P, Q, two points (which can also be given as a list).
   - `line(`P, Q`)` returns and draws the line whose direction is from the P to Q.
2. **A point and a direction vector.**
   - `line` takes two arguments:
     - P, a point.
     - $[u_1, u_2, u_3]$, a direction vector.
   - `line(`P`,[`u_1, u_2, u_3`])` returns and draws the line through the given point with the direction given by the direction vector.
3. **Two planes.**
   - `line` takes two arguments: eqn_1, eqn_2, the equations of two planes.
   - `line(`eqn_1, eqn_2`)` returns and draws the line which is the intersection of the planes.
   - The direction of this line is given by the cross-product of the normals for the planes. For example, the intersection of the planes $x = y$ (normal $(1, -1, 0)$) and $y = z$ (normal $(0, 1, -1)$) will be $(1, -1, 0) \times (0, 1, -1) = (1, 1, 1)$.

Examples

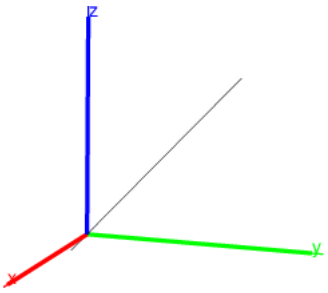
```
> line([0,3,0],point(3,0,3))
```




```
> line([0,3,0],[3,0,3])
```




```
> line(x=y,y=z)
```



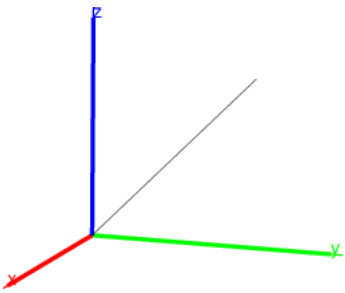
27.4.2 Half lines in space

See Section 26.6.2, p. 694 for half-lines in the plane.
The `half_line` command creates rays.

- `half_line` take two arguments: P, Q, two points (which can also be given as a list).
- `half_line(`P, Q`)` returns and draws the ray from P through Q

Example

```
> half_line(point(0,0,0),point(1,1,1))
```



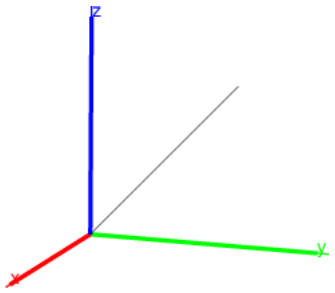
27.4.3 Segments in space

See Section 26.6.3, p. 695 for segments in the plane.
The `segment` command draws line segments.

- `segment` takes two arguments: P, Q, two points (which can also be given as a list).
- `segment(`P, Q`)` returns the corresponding line segment and draws it.

Example

```
> segment(point(0,0,0),point(1,1,1))
```



27.4.4 Vectors in space

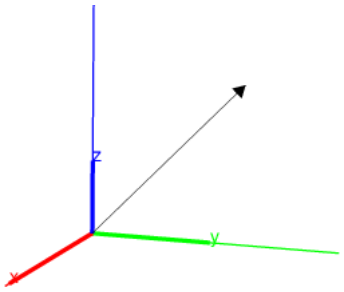
See Section 26.6.4, p. 695 for vectors in the plane.

The `vector` command returns and draws vectors. It can take its arguments in two different ways.

1. **The coordinates of the vector.**
   - `vector` takes L, a list of the coordinates of the vector.
   - `vector(`L`)` returns and draws the vector with the given coordinates, starting from the origin.
2. **Two points or a point and a vector.**
   - `vector` takes two arguments:
     - P, a point.
     - Q, a point or a vector.
   - `vector(`P, Q`)` returns and draws the corresponding vector. If the arguments are two points, the vector goes from P to Q. If the arguments are a point and a vector, then the vector starts at P.

Examples

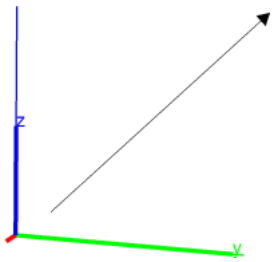
```
> vector([1,2,3])
```



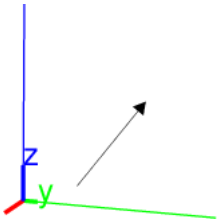

```
> vector(point(-1,0,0),point(0,1,2))
```

or:

```
> vector([-1,0,0],[0,1,2])
```




```
> V:=vector([-1,0,0],[0,1,2]);
  vector(point(-1,2,0),V)
```



27.4.5 Parallel lines and planes in space

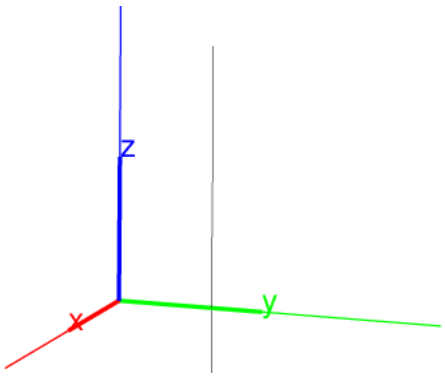
See Section 26.6.5, p. 697 for parallel lines in the plane.

The `parallel` command can take its arguments in different ways. It returns and draws a line or plane depending on the arguments.

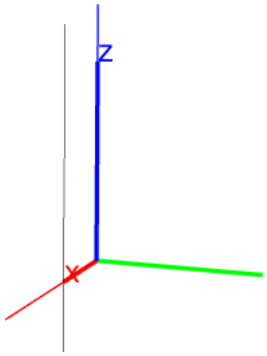
1. **With a point and a line.**
   - `parallel` takes two arguments:
     - P, a point.
     - L, a line.
   - `parallel(`P, L`)` returns and draws the line through P parallel to L.
2. **With two non-parallel lines.**
   - `parallel` takes two arguments: L, M, two lines which aren't parallel.
   - `parallel(`L, M`)` returns and draws the plane containing L which is parallel to M.
3. **With a point and a plane.**
   - `parallel` takes two arguments:
     - P, a point.
     - PL, a plane.
   - `parallel(`P, PL`)` returns and draws the plane through P that is parallel to PL.
4. **With a point and two non-parallel lines.**
   - `parallel` takes three arguments:
     - P, a point.
     - L, M, two non-parallel lines.
   - `parallel(`P, L, M`)` returns and draws the plane through P that is parallel to L and M.

Examples

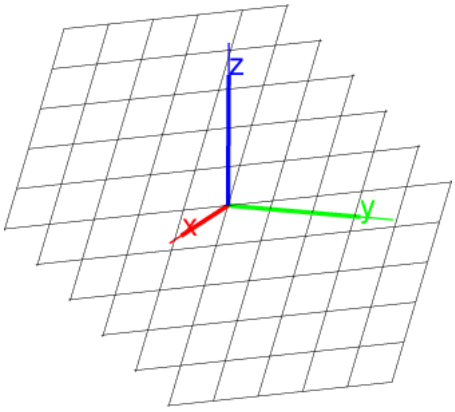
```
> parallel(point(1,1,1),line(point(0,0,0),point(0,0,1)))
```



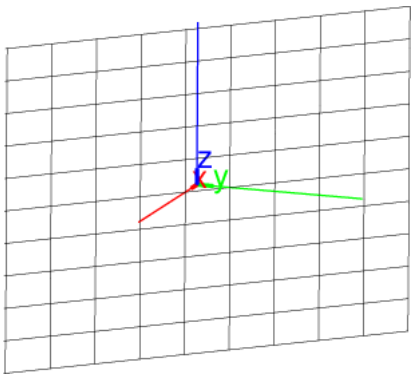

```
> parallel(line(point(1,0,0),point(0,1,0)),line(point(0,0,0),point(0,0,1)))
```




```
> parallel(point(0,0,0),plane(point(1,0,0),point(0,1,0),point(0,0,1)))
```




```
> parallel(point(1,1,1),line(point(0,0,0),point(0,0,1)),
  line(point(1,0,0),point(0,1,0)))
```



27.4.6 Perpendicular lines and planes in space

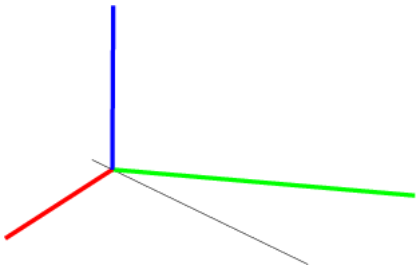
See Section 26.6.6, p. 697 for perpendicular lines in the plane.

The `perpendicular` command can take its arguments in different ways. It returns and draws a line or plane, depending on the arguments.

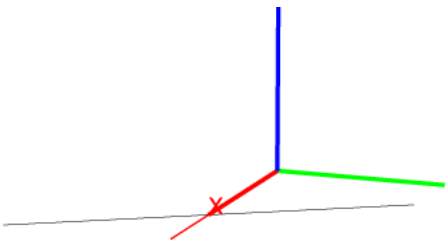
1. **With a point and a line.**
   - `perpendicular` takes two arguments:
     - P, a point.
     - L, a line.
   - `perpendicular`(P, L) returns and draws the line through P that is perpendicular to L.
2. **With a line and a plane.**
   - `perpendicular` takes two arguments:
     - L, a line.
     - P, a plane.
   - `perpendicular`(L, P) returns and draws the plane containing L that is perpendicular to P.

Examples

```
> perpendicular(point(0,0,0),line(point(1,0,0),point(0,1,0)))
```



```
> perpendicular(line(point(0,0,0),point(1,1,0)),
  plane(point(1,0,0),point(0,1,0),point(0,0,1)))
```



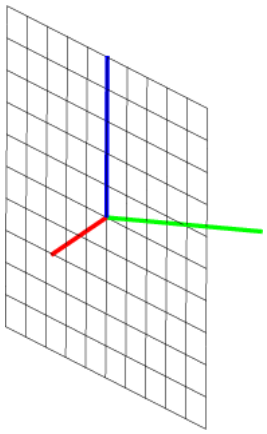
27.4.7 Planes orthogonal to lines and lines orthogonal to planes in space

The `orthogonal` command finds orthogonal objects. It takes its arguments in different ways, and returns and draws a line or plane, depending on the arguments.

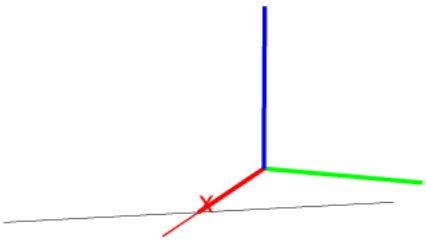
1. **With a point and a line.**
   - `orthogonal` takes two arguments:
     - P, a point.
     - L, a line.
   - `orthogonal(`P, L`)` returns and draws the plane through P orthogonal to L.
2. **With a line and a plane.**
   - `orthogonal` takes two arguments:
     - L, a line.
     - P, a plane.
   - `orthogonal(`L, P`)` returns and draws the plane containing L that is perpendicular to P.

Examples

```
> orthogonal(point(0,0,0),line(point(1,0,0),point(0,1,0)))
```



```
> perpendicular(line(point(0,0,0),point(1,1,0)),
  plane(point(1,0,0),point(0,1,0),point(0,0,1)))
```



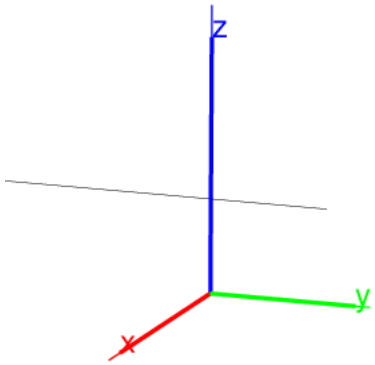
27.4.8 Common perpendiculars to lines in space

The `common_perpendicular` command finds the common perpendicular to two lines.

- `common_perpendicular` takes two arguments: L, M, two lines.
- `common_perpendicular(`L, M`)` returns and draws the common perpendicular to L and M.

Example

```
> L1,L2:=line(point(1,1,0),point(0,1,1)),line(point(0,-1,0),point(1,-1,1)):;
  common_perpendicular(L1,L2)
```



27.5 Planes in space

See also sections 27.4.6 and 27.4.7 for planes perpendicular and orthogonal to lines and planes.

27.5.1 Planes in space

The `plane` command draws and returns a plane. It can take its arguments in three different ways.

1. **Three points.**
   - `plane` takes three arguments: P, Q, R, three points.
   - `plane`(P, Q, R) returns and draws the plane through P, Q and R.
2. **A point and a line.**
   - `plane` takes two arguments:
     - P, a point.
     - L, a line.
   - `plane`(P, L) returns and draws the plane through P and L.
3. **An equation.**
   - `plane` takes *eqn*, the equation of a plane.
   - `plane`$($*eqn*$)$ returns and draws the plane with the given equation.

Example

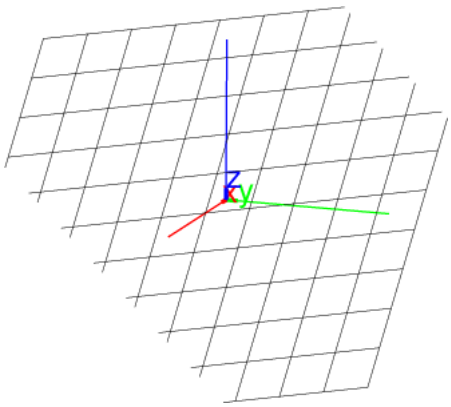
```
> plane(point(0,0,5),point(0,5,0),point(0,0,5))
```

or:

```
> plane(point(0,0,5),line(point(0,5,0),point(0,0,5)))
```

or:

```
> plane(x+y+z=5)
```



27.5.2 Bisector plane in space

See Section 26.6.10, p. 699 for perpendicular bisectors in the plane.

The `perpen_bisector` command finds the perpendicular bisector plane of a line segment.

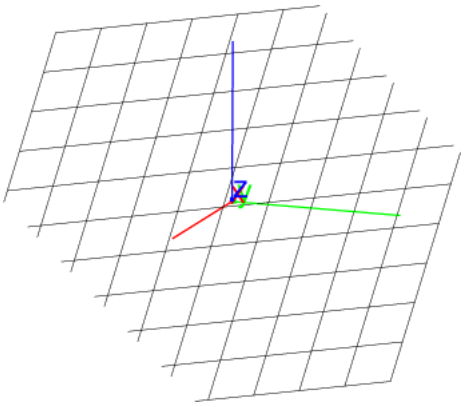
- `perpen_bisector` takes *seg*, a line segment (or the end points of the segment).
- `perpen_bisector(`*seg*`)` returns and draws the perpendicular bisector plane of *seg*.

Example

```
> perpen_bisector(point(0,0,0),point(4,4,4))
```

or:

```
> perpen_bisector(segment([0,0,0],[4,4,4])
```



27.5.3 Tangent planes in space

See Section 26.6.7, p. 698 for tangents in the plane.

The `tangent` command finds tangent planes to surfaces.

- `tangent` takes two arguments:
  - *obj*, an object in space.
  - P, a point in space.

    If *obj* is the graph of a function, then P can be a point in the domain of the function, and the point on the graph will be used.

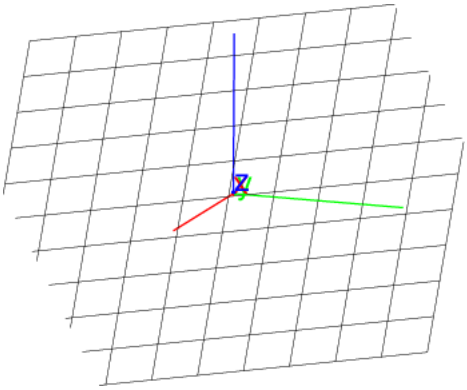
  or

– e, a point defined with `element` (see Section 26.5.15, p. 692) using a curve and parameter value.

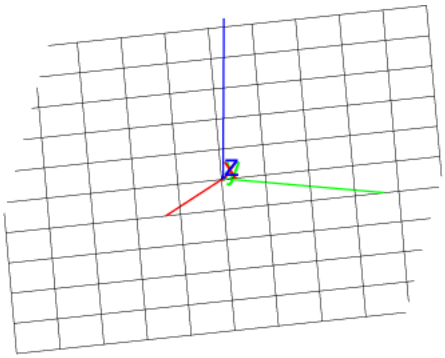
- `tangent(`$objP$`)` returns and draws the plane through P that's perpendicular to obj.

Examples

```
> S:=sphere([0,0,0],3);
  tangent(S,[2,2,1])
```



```
> G:=plotfunc(x^2+y^2,[x,y]);
  tangent(G,[2,2])
```



27.6 Triangles in space

27.6.1 Drawing triangles in space

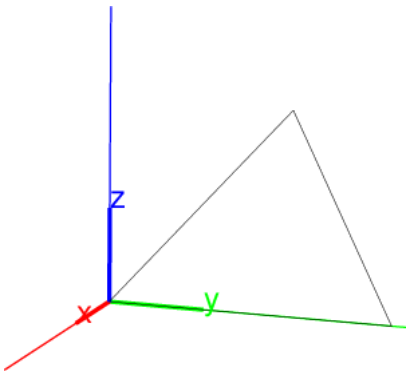
See Section 26.7.1, p. 700 for the `triangle` command in the plane.

The `triangle` command creates triangles.

- `triangle` takes three arguments: A, B, C, three points.
- `triangle(`P, Q, R`)` returns and draws the triangle with vertices A, B and C.

Example

```
> A:=point(0,0,0); B:=point(3,3,3); C:=point(0,3,0);
  triangle(A,B,C)
```



27.6.2 Isosceles triangles in space

See Section 26.7.2, p. 701 for isosceles triangles in the plane.

The `isosceles_triangle` command returns and draws an isosceles triangle. It can take its arguments in two different ways.

1. **Three points.**
   - `isosceles_triangle` takes three mandatory arguments and one optional argument:
     - A, B, P, three points.
     - Optionally, *var*, a variable name.
   - `isosceles_triangle`$(A, B, P\,\langle, var\rangle)$ returns and draws the isosceles triangle ABC in the plane ABP, oriented so that angle BAC is positive and the equal interior angles of the isosceles triangle are determined by angle ABP.

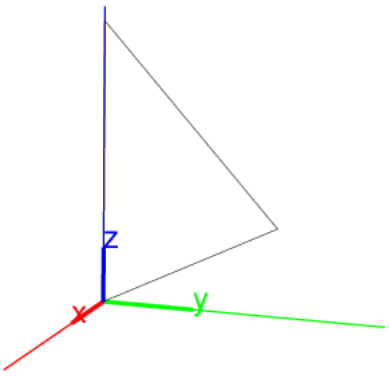
     If the variable name *var* is given, it will be given the value of C, the third vertex of the triangle.

2. **Three points and a real number.**
   - `isosceles_triangle` takes three mandatory arguments and one optional argument:
     - A, B, two points.
     - $[P, c]$, a list consisting of a point R and a real number c.
     - Optionally, *var*, a variable name.
   - `isosceles_triangle`$(A, B, [P, c]\,\langle, var\rangle)$ returns and draws the triangle ABC in plane ABP, oriented so that the angle BAC is positive. The measure of the equal interior angles is c.

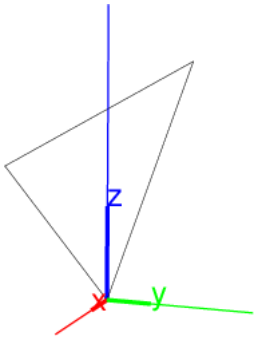
     If the variable name *var* is given, it will be given the value of the third vertex of the triangle.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  isosceles_triangle(A,B,P);
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  isosceles_triangle(A,B,[P,3*pi/4])
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  isosceles_triangle(A,B,[P,3*pi/4],C);
  coordinates(C)
```

$$\left[\frac{-3\sqrt{2}-3}{2}, \frac{-3\sqrt{2}-3}{2}, \frac{-3\sqrt{2}+6}{2}\right]$$

27.6.3 Right triangles in space

See Section 26.7.3, p. 702 for right triangles in the plane.

The `right_triangle` command returns and draws a right triangle. It can take its arguments in two different ways.

1. **Three points.**
   - `right_triangle` takes three mandatory arguments and one optional argument:
     - A, B, P, three points.
     - Optionally, *var*, a variable name.
   - `right_triangle`$(A, B, P\,\langle, var\rangle)$ returns and draws the right triangle BAC in plane ABP with the right angle at vertex A. The triangle is oriented so that the angle BAC is positive. The length of AC equals the length of AP.

     If the variable name *var* is given, it will be assigned to the vertex C.
2. **Three points and a real number.**
   - `right_triangle` takes mandatory three arguments and one optional argument:
     - A, B, two points.

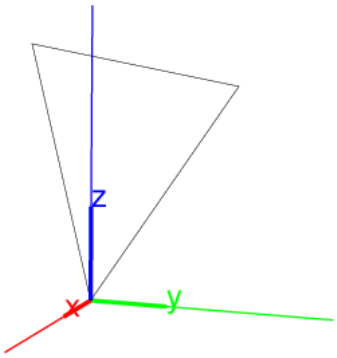
- $[P, k]$, a list consisting of a point P and a real number k.
- Optionally, *var*, a variable name.

- `right_triangle`$(A, B, [P, k]\ \langle, var\rangle)$ returns and draws the right triangle BAC in plane ABP with the right angle at vertex A, and the length of AC equals $|k|$ times the length of AP. Angles BAC and BAP have the same orientation if k is positive; they have opposite orientation if k is negative. So, if β is the angle ABC, then $\tan(\beta) = k$.

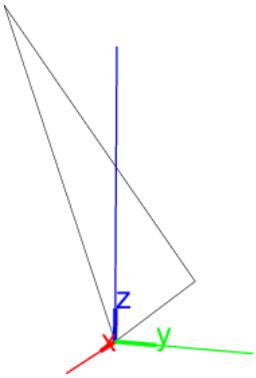
  If the variable name *var* is given, it will be assigned to the vertex C.

Examples

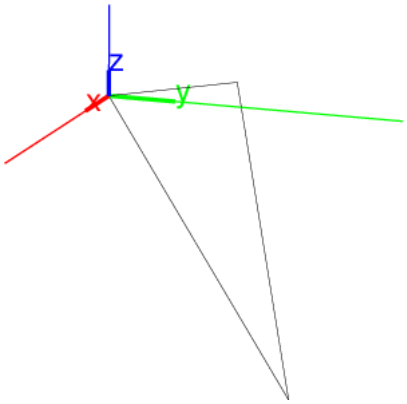
```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  right_triangle(A,B,P);
```




```
> right_triangle(A,B,[P,2])
```




```
> right_triangle(A,B,[P,-2])
```



```
> right_triangle(A,B,[P,2],C);
  coordinates(C)
```

$$\left[-3\sqrt{2}, -3\sqrt{2}, 6\sqrt{2}\right]$$

27.6.4 Equilateral triangles in space

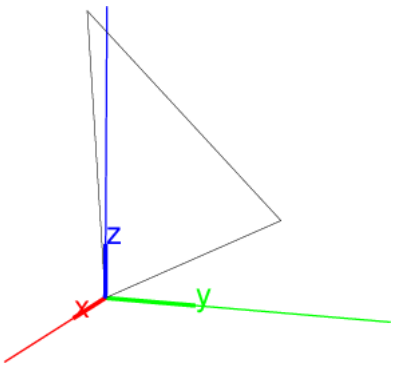
See Section 26.7.4, p. 703 for equilateral triangles in the plane.
The `equilateral_triangle` command returns and draws equilateral triangles.

- `equilateral_triangle` takes three mandatory arguments and one optional argument:
  - A, B, P, three points.
  - Optionally, *var*, a variable name.
- `equilateral_triangle`$(A, B, P\,\langle, var\rangle)$ returns and draws equilateral triangle ABC, where C and P are on the same side of line AB in plane ABP.

  If the argument *var* is given, it will be assigned the value of C.

Examples

```
> A,B,p:=point(0,0,0),point(3,3,3),point(0,0,3);
  equilateral_triangle(A,B,P)
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  equilateral_triangle(A,B,P,C);
  simplify(coordinates(C))
```

$$\left[\frac{-3\sqrt{6}+6}{4}, \frac{-3\sqrt{6}+6}{4}, \frac{3\sqrt{6}+3}{2}\right]$$

27.7 Quadrilaterals in space

See Section 26.8, p. 704 for quadrilaterals in the plane.

27.7.1 Squares in space

See Section 26.8.1, p. 704 for squares in the plane.
The `square` command creates squares.

- `square` takes three mandatory arguments and two optional arguments:

– A, B, P, three points.

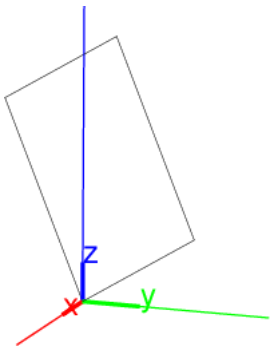
– Optionally, var_1, var_2, two variable names.

- `square`$(A, B, P\,\langle, var_1, var_2\,\rangle)$ returns and draws the square with one side AB and the remaining sides in the same half-plane as P.

  If the arguments *var1* and *var2* are given, they will be assigned to the new vertices.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  square(A,B,P)
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  square(A,B,P,C,D);
  coordinates(C),coordinates(D)
```

$$\left[\frac{-3\sqrt{2}+6}{2}, \frac{-3\sqrt{2}+6}{2}, 3\sqrt{2}+3\right], \left[-\frac{3}{2}\sqrt{2}, -\frac{3}{2}\sqrt{2}, 3\sqrt{2}\right]$$

27.7.2 Rhombuses in space

See Section 26.8.2, p. 705 for rhombuses in the plane.

The `rhombus` command returns and draws a rhombus. It takes it arguments in two different ways.

1. **Three points.**

   - `rhombus` takes three mandatory arguments and two optional arguments:

     – A, B, P, three points.

     – Optionally, var_1, var_2, two variable names.

   - `rhombus`$(A, B, P\,\langle, var_1, var_2\,\rangle)$ returns and draws the rhombus $ABCD$, which is in the plane ABP, oriented so that angle BAP is positive, and D is on the ray AP.

     If the arguments *var1* and *var2* are given, they will be assigned to the vertices C and D.

2. **Three points and a real number.**

   - `rhombus` takes three mandatory arguments and two optional argument:

     – A, B, two points.

     – $[P, a]$, a list consisting of a point P and a real number a.

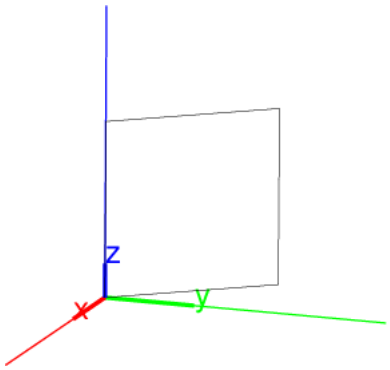
     – Optionally, var_1, var_2, two variable names.

- `rhombus`$(A, B, [P, a]\,\langle, var_1, var_2\,\rangle)$ returns and draws the rhombus $ABCD$, which is in the plane ABP, oriented so that angle BAP is positive, and angle BAD equals a.

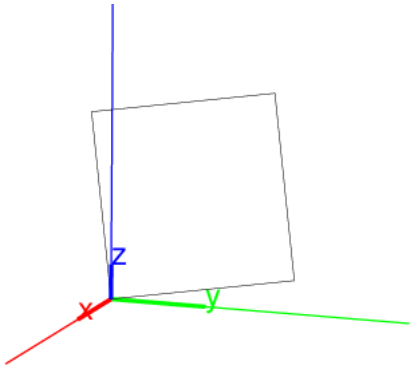
  If the arguments *var1* and *var2* are given, they will be assigned to the vertices C and D.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  rhombus(A,B,P)
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  rhombus(A,B,[P,pi/3])
```




```
> rhombus(A,B,[P,pi/3],C,D);
  simplify(coordinates(C)), simplify(coordinates(D))
```

$$\left[\frac{-3\sqrt{6}+18}{4}, \frac{-3\sqrt{6}+18}{4}, \frac{3\sqrt{6}+9}{2}\right], \left[\frac{-3\sqrt{6}+6}{4}, \frac{-3\sqrt{6}+6}{4}, \frac{3\sqrt{6}+3}{2}\right]$$

27.7.3 Rectangles in space

See Section 26.8.3, p. 705 for rectangles in the plane.

The `rectangle` command returns and draws a rectangle. It can take its arguments in two different ways.

1. **Three points.**
   - `rectangle` takes three mandatory arguments and two optional arguments:
     - A, B, P, three points.
     - Optionally, var_1, var_2, two variable names.

- `rectangle`$(A, B, P \langle, var_1, var_2 \rangle)$ returns and draws the rectangle $ABCD$, in the plane ABP, oriented to that angle BAP is positive, and with the length of side AD equals AP.

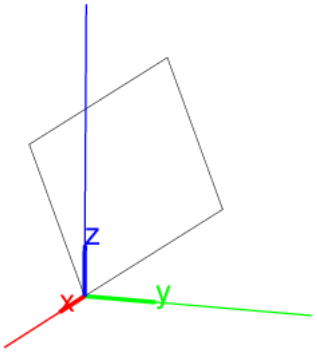
  If the arguments *var1* and *var2* are given, they will be assigned to the vertices C and D.

2. **Three points and a real number.**
   - `rectangle` takes three mandatory arguments and two optional argument:
     - A, B, two points.
     - $[P, k]$, a list consisting of a point P and a real number k.
     - Optionally, var_1, var_2, two variable names.
   - `rectangle`$(A, B, [P, k] \langle, var_1, var_2 \rangle)$ returns and draws the rectangle $ABCD$, which is in the plane ABP, and with the length of AD equal to $|k|$ times the length of AB. Angle BAD and angle BAP have the same orientation if k is positive and opposite orientation if k is negative.

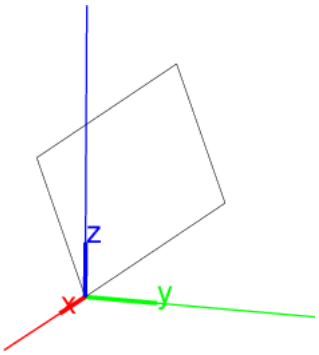
     If the arguments *var1* and *var2* are given, they will be assigned to the vertices C and D.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  rectangle(A,B,P)
```




```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  rectangle(A,B,[P,1/2])
```




```
> rectangle(A,B,P,C,D);
  simplify(coordinates(C)),simplify(coordinates(D))
```

$$\left[-\frac{\sqrt{6}}{2}, -\frac{\sqrt{6}}{2}, \sqrt{6}\right], \left[\frac{-\sqrt{6}+6}{2}, \frac{-\sqrt{6}+6}{2}, \sqrt{6}+3\right]$$

27.7.4 Parallelograms in space

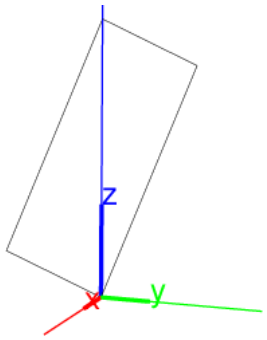
See Section 26.8.4, p. 707 for parallelograms in the plane.
The `parallelogram` command creates parallelograms in space.

- `parallelogram` takes three mandatory arguments and one optional argument:
  - A, B, C, three points.
  - *var*, a variable name.
- `parallelogram(`$A, B, C \langle, var\rangle$`)` returns and draws the parallelogram $ABCD$ determined by A, B and C.

  If the option *var* is given, the point D will be assigned to it.

Examples

```
> A,B,C:=point(0,0,0),point(3,3,3),point(0,0,3);
  parallelogram(A,B,C)
```




```
> parallelogram(A,B,C,D);
  coordinates(D)
```

$$[-3, -3, 0]$$

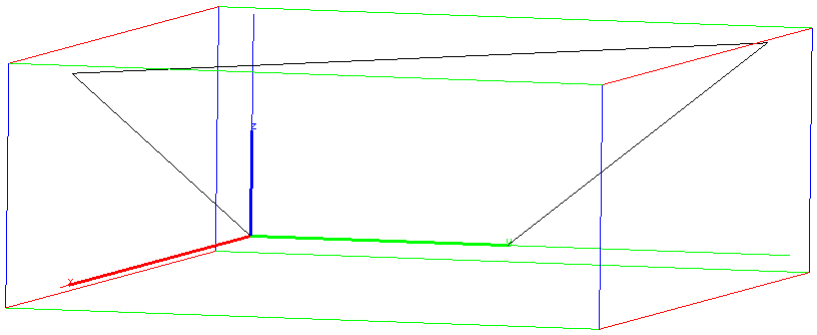
27.7.5 Arbitrary quadrilaterals in space

See Section 26.8.5, p. 707 for quadrilaterals in the plane.
The `quadrilateral` command creates arbitrary quadrilaterals.

- `quadrilateral` takes four arguments: A, B, C, D, four points.
- `quadrilateral(`A, B, C, D`)` returns and draws quadrilateral $ABCD$.

Example

```
> quadrilateral(point(0,0,0),point(0,1,0),point(0,2,2),point(1,0,2))
```



27.8 Polygons in space

See Section 26.9, p. 708 for polygons in the plane.

27.8.1 Hexagons in space

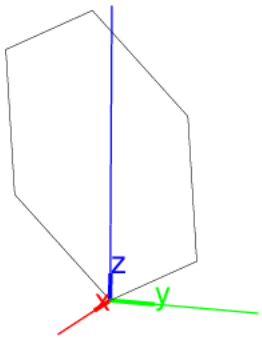
See Section 26.9.1, p. 708 for hexagons in the plane.

The `hexagon` command creates hexagons in space.

- `hexagon` takes three mandatory arguments and four optional arguments:
  - A, B, P, four points.
  - Optionally, $var_1, var_2, var_3, var_4$, four variable names.
- `hexagon`$(A, B, P\,\langle, var_1, var_2, var_3, var_4\,\rangle)$ returns and draws the regular hexagon $ABCDEF$ in the plane ABP, oriented so that angle ABC is positive.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  hexagon(A,B,P)
```




```
> hexagon(A,B,P,C,D,E,F);
  simplify(coordinates(C))
```

$$\left[\frac{-3\sqrt{6}+18}{4}, \frac{-3\sqrt{6}+18}{4}, \frac{3\sqrt{6}+9}{2}\right]$$

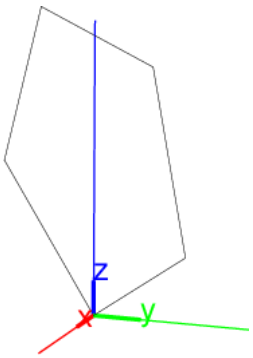
27.8.2 Regular polygons in space

See Section 26.9.2, p. 709 for regular polygons in the plane.
The `isopolygon` command creates regular polygons in space.

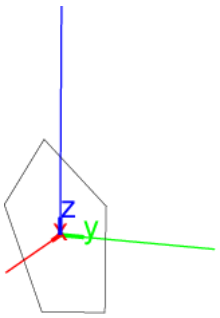
- `isopolygon` takes four arguments:
  - A, B, P, three points.
  - k, an integer.
- `isopolygon`(A, B, P, k) returns and draws a regular polygon with one edge AB in the plane ABP with $|k|$ sides. If $|k|$ is positive, then the polygon is positively oriented, otherwise it is negatively oriented.

Examples

```
> A,B,P:=point(0,0,0),point(3,3,3),point(0,0,3);
  isopolygon(A,B,P,5)
```




```
> isopolygon(A,B,P,-5)
```



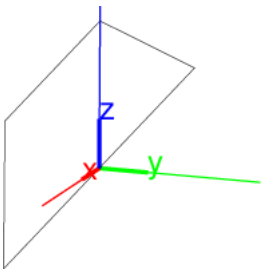
27.8.3 General polygons in space

See Section 26.9.3, p. 710 for general polygons in the plane.
The `polygon` command creates general polygons in space.

- `polygon` takes S, a sequence of points.
- `polygon`(S)returns and draws the polygon whose vertices are the given points.

Example

```
> A,B,C,D,E:=point(0,0,0),point(3,3,3),point(0,0,3),point(-3,-3,0),point(-3,-3,-3);
  polygon(A,B,C,D,E)
```



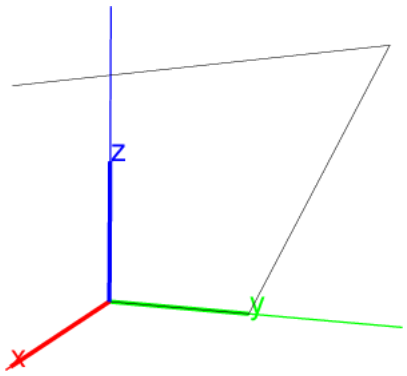
27.8.4 Polygonal lines in space

See Section 26.9.4, p. 710 for polygonal lines in the plane.
The `open_polygon` command creates polygonal lines in space.

- `open_polygon` takes S, a sequence of points.
- `open_polygon`(S)returns and draws the polygon line whose vertices are the given points.

Example

```
> open_polygon(point(0,0,0),point(0,1,0),point(0,2,2),point(1,0,2))
```



27.9 Circles and conics in space

27.9.1 Circles in space

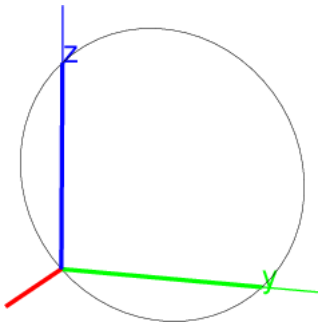
See Section 26.10.1, p. 711 for circles in the plane.
The `circle` command returns and draws a circle. It can take its arguments in two different ways.

1. **Three points.**
   - `circle` takes three arguments: A, B, C, three points.
   - `circle`(A, B, C) returns and draws the circle in plane ABC with a diameter AB.
2. **Two points and a vector.**
   - `circle` takes three points:
     - C, a point (which can be given by its coordinates).

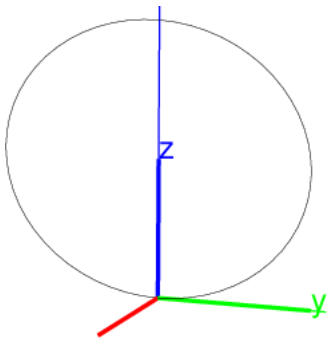
  - v, a vector.
  - A, a point (which can be given by its coordinates).
- `circle`(C, v, A) returns and draws the circle in plane $C(C+v)C$ with center C and containing $C + v$.

Examples

```
> circle(point(0,0,1),point(0,1,0),point(0,2,2))
```




```
> circle(point(0,0,1),vector(0,1,0),point(0,2,2))
```



27.9.2 Ellipses in space

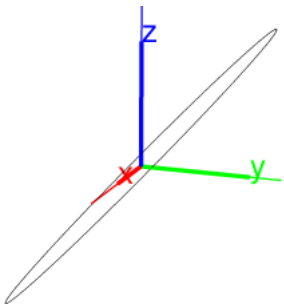
See Section 26.11.1, p. 717 for ellipses in the plane.

The `ellipse` command creates ellipses in space.

- `ellipse` takes three arguments: A, B, C three non-collinear points.
- `ellipse`(A, B, C) returns and draws the ellipse with foci A and B passing through C.

Example

```
> ellipse(point(-1,0,0),point(1,0,0),point(1,1,1))
```



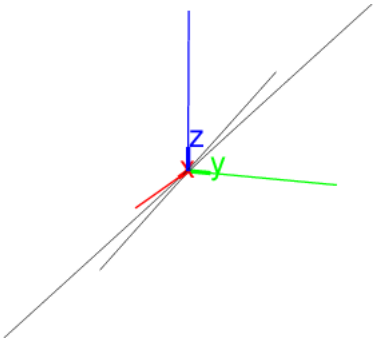
27.9.3 Hyperbolas in space

See Section 26.11.2, p. 718 for hyperbolas in the plane.
The `hyperbola` command creates hyperbolas in space.

- `hyperbola` takes three arguments: A, B, C three non-collinear points.
- `hyperbola(`A, B, C`)` returns and draws the hyperbola with foci A and B passing through C.

Example

```
> hyperbola(point(-1,0,0),point(1,0,0),point(1,1,1))
```



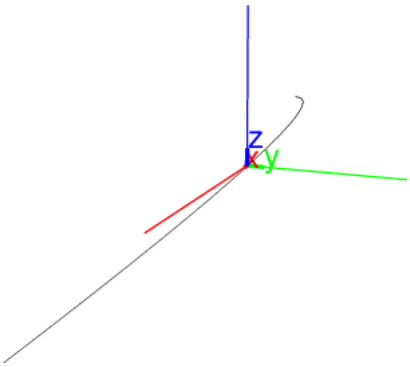
27.9.4 Parabolas in space

See Section 26.11.3, p. 719 for parabolas in the plane.
The `parabola` command creates parabolas in space.

- `parabola` takes three arguments: A, B, C three non-collinear points.
- `parabola(`A, B, C`)` returns and draws the parabola in plane ABC with focus A and vertex B.

Example

```
> parabola(point(0,0,0),point(-1,0,0),point(1,1,1))
```



27.10 3D coordinates

27.10.1 Abscissa of a 3D point

See Section 26.12.2, p. 721 for abscissas in 2D geometry.
The `abscissa` command finds the abscissa (x-coordinate) of a point.

- `abscissa` takes P, a point.
- `abscissa(`P`)` returns the abscissa of P.

Example

```
> abscissa(point(1,2,3))
```

1

27.10.2 Ordinate of a 3D point

See Section 26.12.3, p. 721 for ordinates in 2D geometry.

The `ordinate` command finds the ordinate (y-coordinate) of a point.

- `ordinate` takes P, a point.
- `ordinate(`P`)` returns the ordinate of P.

Example

```
> ordinate(point(1,2,3))
```

2

27.10.3 Cote of a 3D point

The `cote` command finds the cote (z-coordinate) of a point.

- `cote` takes P, a point.
- `cote(`P`)` returns the cote of P.

Example

```
> cote(point(1,2,3))
```

3

27.10.4 Coordinates of a point, vector or line in space

See Section 26.12.4, p. 722 for coordinates in 2D geometry.

The `coordinates` command takes finds the coordinates of a point.

- `coordinates` takes P, which can be a point (or a sequence or list of points), a vector, or a line.
- If P is a point, then `coordinates(`P`)` returns a list consisting of the abscissa, ordinate and cote. If P is a list or sequence of points, then the command returns a list or sequence of such lists.
- If P is a vector, for example from A to B, then `coordinates(`P`)` returns a list of the coordinates of $B - A$.
- If P is a line, then `coordinates(`P`)` returns a list of two points on the line, in the order determined by the direction of the line.

Examples

```
> coordinates(point(1,2,3))
```

$$[1, 2, 3]$$

```
> coordinates(point(0,1,2),point(1,2,4))
```

$$[0, 1, 2], [1, 2, 4]$$

Note that if the argument is a list of real numbers, it is interpreted as a list of points on the real axis of the plane.

```
> coordinates([1,2,4])
```

$$\begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 4 & 0 \end{bmatrix}$$

```
> coordinates(vector(point(1,2,3),point(2,4,7)))
```

$$[1, 2, 4]$$

```
> coordinates(line(point(-1,1,0),point(1,2,3)))
```

$$[[-1, 1, 0], [1, 2, 3]]$$

```
> coordinates(line(x-2*y+3=0, 6*x+3*y-5*z+3=0))
```

$$[[-1, 1, 0], [9, 6, 15]]$$

27.10.5 Cartesian equation of an object in space

See Section 26.12.7, p. 724 for Cartesian equations of 2D objects.

The `equation` command finds equations for geometric objects.

- `equation` takes G, a geometric object.
- `equation(`G`)` returns Cartesian equations in `x`, `y` and `z` which specify the object G.

  The variables `x`, `y` and `z` must be unassigned. If they have assignments, they can be unassigned with `purge(x,y,z)`.

Examples

```
> equation(line(point(0,1,0),point(1,2,3)))
```

$$x - y + 1 = 0, 3x + 3y - 2z - 3 = 0$$

```
> equation(sphere(point(0,1,0),2))
```

$$x^2 + y^2 - 2y + z^2 - 3 = 0$$

27.10.6 Parametric equation of an object in space

See Section 26.12.8, p. 724 for parametric equations in 2D geometry.
The `parameq` command finds parameterizations for geometric objects.

- `parameq` takes G, a geometric object.
- `parameq(`G`)` returns a parameterization for the object G.

  For a curve, the parameter is `t`, for a surface, the parameters are `u` and `v`. These variables must be unassigned. If they have assignments, they can be unassigned with `purge(t)` and `purge(u,v)`.

Examples

```
> parameq(line(point(0,1,0),point(1,2,3)))
```

$$[t, t+1, 3t]$$

```
> parameq(sphere(point(0,1,0),2))
```

$$[2\cos u\cos v, 1+2\cos u\sin v, 2\sin u]$$

```
> normal(parameq(ellipse(point(-1,1,1),point(1,1,1),point(0,1,2))))
```

$$\left[\sqrt{2}\cos t, 1, \sin t+1\right]$$

27.10.7 Length of a segment in space

See Section 26.13.2, p. 726 for distances in 2D geometry.
The `distance` command finds the distance between two points.

- `distance` takes two arguments: P, Q, two points or two lists with the coordinates of the points.
- `distance(`P, Q`)` returns the distance between P and Q.

Example

```
> distance(point(-1,1,1),point(1,1,1))
```

or:

```
> distance([-1,1,1],[1,1,1])
```

$$2$$

27.10.8 Squared length of a segment in space

See Section 26.13.3, p. 727 for squares of lengths in 2D geometry.
The `distance2` command finds the square of the distance between two points.

- `distance2` takes two arguments: P, Q, two points or two lists with the coordinates of the points.
- `distance2(`P, Q`)` returns the square of the distance between P and Q.

Example

```
> distance2(point(-1,1,1),point(1,1,1))
```

or:

```
> distance2([-1,1,1],[1,1,1])
```

$$4$$

27.10.9 Measure of an angle in space

See Section 26.13.4, p. 727 for angle measures in 2D geometry.

The `angle` command finds the measures of angles in space. It can take its arguments in three different ways.

1. **Three points.**
   - `angle` takes three arguments: A, B, C, three points.
   - `angle`(A, B, C) returns the measure of the undirected angle BAC.
2. **Two intersecting lines.**
   - `angle` takes two arguments: L, M, two lines which intersect.
   - `angle`(L, M) returns the measure of the angle between the lines L and M.
3. **A line and a plane.**
   - `angle` takes two arguments:
     - L, a line.
     - P, a plane.
   - `angle`(L, P) returns the measure of the angle between L and P.

Examples

```
> angle(point(0,0,0),point(1,0,0),point(0,0,1))
```

$$\frac{1}{2}\pi$$

```
> angle(line([0,0,0],[1,1,0]),line([0,0,0],[1,1,1]))
```

$$\arccos\left(\frac{\sqrt{6}}{3}\right)$$

```
> angle(line([0,0,0],[1,1,0]),plane(x+y+z=0))
```

$$\arccos\left(\frac{\sqrt{6}}{3}\right)$$

27.11 Properties

27.11.1 Checking whether object in space is in another object

See Section 26.15.1, p. 737 for checking elements in 2D geometry.

The `is_element` command determines whether or not a geometric object is contained in another.

- `is_element` takes two arguments: G, H, two geometric objects.
- `is_element`(G, H) returns 1 if G is contained in H and 0 otherwise.

Examples

```
> P:=plane([0,0,0],[1,2,-3],[1,1,-2]);
  is_element(point(2,3,-5),P)
```

1

```
> L,P:=line([2,3,-2],[-1,-1,-1]),plane([-1,-1,-1],[1,2,-3],[1,1,-2]);
  is_element(L,P)
```

0

27.11.2 Checking whether points and/or lines in space are coplanar

The `is_coplanar` command determines whether or not several points or several lines are coplanar.

- `is_coplanar` takes S, a sequence where each element is a point or a line.
- `is_coplanar`(S) returns 1 is the elements of S are coplanar and 0 otherwise.

Examples

```
> is_coplanar([0,0,0],[1,2,-3],[1,1,-2],[2,1,-3])
```

1

```
> is_coplanar([-1,2,0],[1,2,-3],[1,1,-2],[2,1,-3])
```

0

```
> is_coplanar([0,0,0],[1,2,-3],line([1,1,-2],[2,1,-3]))
```

1

```
> is_coplanar(line([-1,2,0],[1,2,-3]),line([1,1,-2],[2,1,-3]))
```

0

27.11.3 Checking whether lines and/or planes in space are parallel

See Section 26.15.11, p. 743 for checking for parallels in 2D geometry.

The `is_parallel` command determines if two objects are parallel.

- `is_parallel` takes two arguments: L, P, each one either a line or a plane.
- `is_parallel`(L, P) returns 1 if L and P are parallel and 0 otherwise.

Examples

```
> L1,L2:=line([0,0,0],[-1,-1,-1]),line([2,3,-2],[-1,-1,-1]);
  is_parallel(L1,L2)
```

0

```
> P:=plane([-1,-1,-1],[1,2,-3],[0,0,0]);
  is_parallel(P,L2)
```

1

```
> P1,P2:=plane([0,0,0],[1,2,-3],[1,1,-2]),plane([1,1,0],[2,3,-3],[2,2,-2]);
  is_parallel(P1,P2)
```

1

27.11.4 Checking whether lines and/or planes in space are perpendicular

See Section 26.15.12, p. 743 for checking for perpendicularity in 2D geometry.

The `is_perpendicular` command determines if two objects are perpendicular.

- `is_perpendicular` takes two arguments: L, P, each one either a line or a plane.
- `is_perpendicular`(L, P) returns 1 if L and P are perpendicular and 0 otherwise.

Note that two lines must be coplanar to be perpendicular.

Examples

```
> is_perpendicular(line([2,3,-2],[-1,-1,-1]),line([1,0,0],[1,2,8]))
```

0

```
> P1,P2:=plane([0,0,0],[1,2,-3],[1,1,-2]),plane([-1,-1,-1],1,2,-3],[0,0,0]);
  is_perpendicular(P1,P2)
```

1

```
> L:=plane([2,3,-2],[-1,-1,-1]);
  is_perpendicular(L,P1)
```

0

27.11.5 Checking whether two lines or two spheres in space are orthogonal

See Section 26.15.13, p. 743 for checking for orthogonality in 2D geometry.

The `is_orthogonal` command determines whether or not two objects are orthogonal.

- `is_orthogonal` takes two arguments: L, P, which can be two lines, two spheres, two planes or a line and a plane.
- `is_orthogonal`(L, P) returns 1 is the objects are orthogonal and 0 otherwise.

Examples

```
> is_orthogonal(line([2,3,-2],[-1,-1,-1]),line([1,0,0],[1,2,8]))
```

1

```
> is_orthogonal(line([2,3,-2],[-1,-1,-1]),plane([-1,-1,-1],[-1,0,3],[-2,0,0]))
```

1

```
> is_orthogonal(plane([0,0,0],[1,2,-3],[1,1,-2]),plane([-1,-1,-1],[1,2,-3],[0,0,0]))
```

1

```
> is_orthogonal(sphere([0,0,0],sqrt(2)),sphere([2,0,0],sqrt(2)))
```

1

27.11.6 Checking whether points in space are collinear

See Section 26.15.2, p. 738 for checking for collinearity in 2D geometry.

The `is_collinear` command determines whether or not points in space are collinear.

- `is_collinear` takes L, a list or sequence of points.
- `is_collinear(`L`)` returns 1 if the points in L are collinear, it returns 0 otherwise.

Examples

```
> is_collinear([2,0,0],[0,2,0],[1,1,0])
```

1

```
> is_collinear([2,0,0],[0,2,0],[0,1,1])
```

0

27.11.7 Checking whether points in space are concyclic

See Section 26.15.3, p. 738 for checking for concyclicity in 2D geometry.

The `is_concyclic` command determines whether or not points are lying on the same circle.

- `is_concyclic` takes L, a list or sequence of points.
- `is_concyclic(`L`)` returns 1 if the points in L all lie on the same circle, and 0 otherwise.

Examples

```
> is_concyclic([2,0,0],[0,2,0],[sqrt(2),sqrt(2),0],
               [0,0,2],[2/sqrt(3),2/sqrt(3),2/sqrt(3)])
```

1

```
> is_concyclic([2,0,0],[0,2,0],[1,1,0],[0,0,2],[1,1,1])
```

0

27.11.8 Checking whether points in space are cospherical

The `is_cospherical` command determines whether or not points are cospherical, i.e. whether they all lie on the same sphere.

- `is_cospherical` takes L, a list or sequence of points.
- `is_cospherical(`L`)` returns 1 if the points in L all lie on the same sphere, and 0 otherwise.

Examples

```
> is_cospherical([2,0,0],[0,2,0],[sqrt(2),sqrt(2),0],
                 [0,0,2],[2/sqrt(3),2/sqrt(3),2/sqrt(3)])
```

1

```
> is_cospherical([2,0,0],[0,2,0],[1,1,0],[0,0,2],[1,1,1])
```

0

27.11.9 Checking whether an object in space is an equilateral triangle

See Section 26.15.5, p. 739 for checking for equilateral triangles in 2D geometry.

The `is_equilateral` command determines whether or not a geometric object is an equilateral triangle.

- `is_equilateral` takes G, a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_equilateral(`G`)` returns 1 if the object is an equilateral triangle and 0 otherwise.

Examples

```
> is_equilateral([2,0,0],[0,0,0],[1,sqrt(3),0])
```

1

```
> T:=triangle_equilateral([2,0,0],[0,0,0],[1,sqrt(3),0]);
  is_equilateral(T)
```

1

```
> is_equilateral([2,0,0],[0,2,0],[1,1,0])
```

0

27.11.10 Checking whether an object in space is an isosceles triangle

See Section 26.15.6, p. 739 for checking for isosceles triangles in 2D geometry.

The `is_isosceles` command determines whether or not a geometric object is an isoceles triangle.

- `is_isosceles` takes G, a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_isosceles(`G`)` returns $1, 2$ or 3 if the object is an isoceles triangle (the number indicates which vertex is on two equal sides), returns 4 if the object is an equilateral triangle, and returns 0 otherwise.

Examples

```
> is_isosceles([2,0,0],[0,0,0],[0,2,0])
```

2

```
> T:=triangle_isosceles([0,0,0],[2,2,0],[2,2,2]);
  is_isosceles(T)
```

1

```
> is_isosceles([1,1,0],[-1,1,0],[-1,0,0])
```

0

27.11.11 Checking whether an object in space is a right triangle or a rectangle

See Section 26.15.7, p. 740 for checking for right triangles and rectangles in 2D geometry.

The `is_rectangle` command determines whether or not a geometric object is an rectangle or a right triangle.

- `is_rectangle` takes G, a geometric object or a sequence of three or four points assumed to be the vertices of a triangle or a quadrilateral.
- `is_rectangle(`G`)` returns:
  - (for triangle G) $1, 2$ or 3 if G is a right triangle (the number indicates which vertex is has the right angle).
  - (for quadrilaterals G) 1 if G is a rectangle but not a square.
  - (for quadrilaterals G) 2 if G is square.
  - 0 otherwise.

Examples

```
> is_rectangle([2,0,0],[2,2,0],[0,2,0])
```

2

```
> is_rectangle([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

1

27.11.12 Checking whether an object in space is a square

See Section 26.15.8, p. 741 for checking for squares in 2D geometry.

The `is_square` command determines whether or not a geometric object is a square.

- `is_square` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square(`G`)` returns 1 if the object is a square and returns 0 otherwise.

Examples

```
> is_square([2,2,0],[-2,2,0],[-2,-2,0],[2,-2,0])
```

1

```
> S:=square([0,0,0],[2,0,0],[0,0,1]);
  is_square(S)
```

1

```
> is_square([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

0

27.11.13 Checking whether an object in space is a rhombus

See Section 26.15.9, p. 741 for checking for rhombuses in 2D geometry.

The `is_rhombus` command determines whether or not a geometric object is a rhombus.

- `is_rhombus` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square`(G) returns 1 if G is a rhombus but not a square, returns 2 if G is a square and returns 0 otherwise.

Examples

```
> is_rhombus([2,0,0],[0,1,0],[-2,0,0],[0,-1,0])
```

1

```
> R:=rhombus([0,0,0],[2,0,0],[[0,0,1],pi/4]);
  is_rhombus(S)
```

1

```
> is_rhombus([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

0

27.11.14 Checking whether an object in space is a parallelogram

See Section 26.15.10, p. 742 for checking for parallelograms in 2D geometry.

The `is_parallelogram` command determines whether or not an object is a parallelogram.

- `is_parallelogram` takes G, a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_parallelogram`(G) returns 1 if G is a parallelogram, but not a rhombus or a rectangle, returns 2 if G is a rhombus but not a rectangle, returns 3 if G is a rectangle but not a square, returns 4 is G is a square, and returns 0 otherwise.

Examples

```
> is_parallelogram([0,0,0],[2,0,0],[3,1,0],[1,1,0])
```

1

```
> is_parallelogram([-1,0,0],[0,1,0],[2,0,0],[0,-1,0])
```

0

```
> P:=parallelogram([0,0,0],[2,0,0],[1,1,0]);
  is_parallelogram(P)
```

1

Note that

```
> P:=parallelogram([0,0,0],[2,0,0],[1,1,0],D)
```

defines `P` to be a list consisting of the parallelogram and the point `D`; to test if the object is a parallelogram, the first component of `P` needs to be tested.

```
> is_parallelogram(P[0])
```

1

```
> is_parallelogram([-1,0,0],[0,1,0],[2,0,0],[0,-1,0])
```

0

27.12 Transformations in space

27.12.1 General remarks

The transformations in this section operate on any geometric object. They take as arguments parameters to specify the transformation. They can optionally take a geometric object as the last argument, in which case the transformed object is returned. Without the geometric object as an argument, these transformations will return a new command which performs the transformation. For example, to move an object `P` 3 units up, either

```
> translation([0,0,3],P)
```

or

```
> t:=translation([0,0,3]); t(P)
```

works.

27.12.2 Translation in space

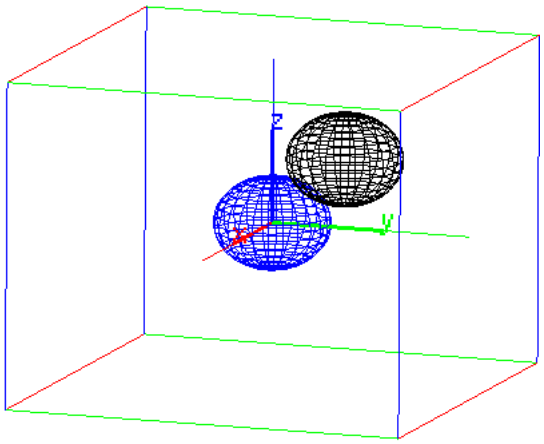
See Section 26.14.2, p. 732 for translations in the plane.

The `translation` command creates a translation.

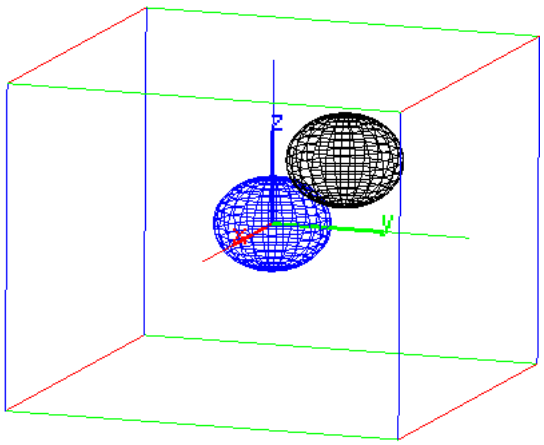
- `translation` takes one mandatory argument and one optional argument:
  - v, the translation vector, which can be given as a vector or a list of coordinates.
  - Optionally, G, a geometric object.
- `translation`(v) returns a new command which translates by v.
- `translation`(v, G) returns and draws the translation G by the vector v.

Examples

```
> t:=translation([1,1,1]);
  S:=sphere([0,0,0],0.5);
  color(S,blue),t(S)
```



```
> translation([1,1,1],S)
```



27.12.3 Reflection in space with respect to a plane, line or point

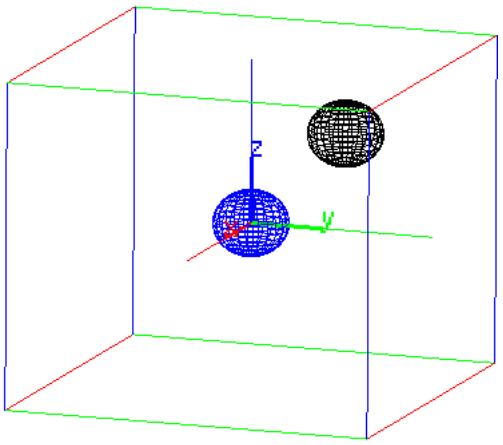
See Section 26.14.3, p. 733 for reflections in the plane.

The `reflection` command creates a reflection.

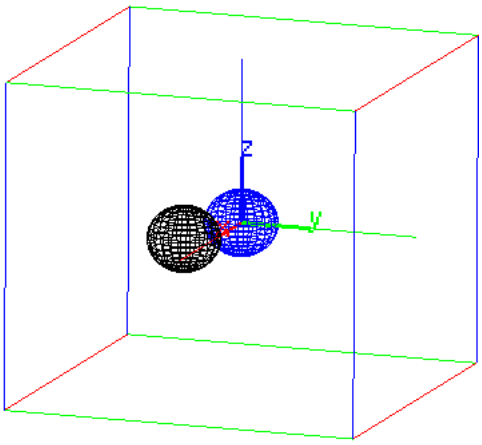
- `reflection` takes one mandatory argument and one optional argument:
  - P, a point, line or plane.
  - Optionally, G, a geometric object.
- `reflection`(P) returns a new command which reflects about P.
- `reflection`(P, G) returns and draws the reflection of G about P.

Examples

```
> S:=sphere([0,0,0],0.5);
  r:=reflection([1,1,1]);
  color(S,blue),r(S)
```




```
> reflection(line([1,1,0],[-1,-3,0]),point(-1,2,4))
```



27.12.4 Rotation in space

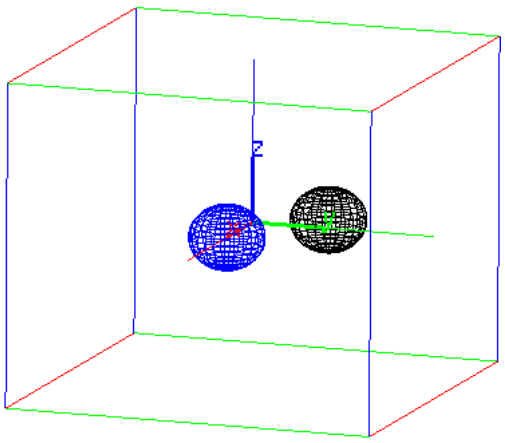
See Section 26.14.4, p. 733 for rotations in the plane.

The `rotation` command creates a rotation.

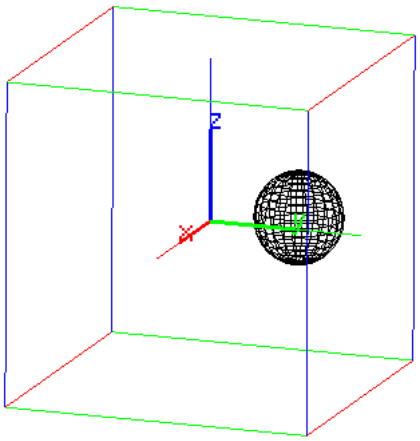
- `rotation` takes two mandatory arguments and one optional argument:
  - L, a line (to rotate about).
  - θ, the angle of rotation.
  - Optionally, G, a geometric object.
- `rotation`(L, θ) returns a new command which rotations about L through an angle of θ.
- `reflection`(L, θ, G) returns and draws the rotation of G about L through an angle of θ.

Examples

```
> S:=sphere([1,0,0],0.5);
  r:=rotation(line(point(0,0,0),point(0,0,1)), 2*pi/3);
  color(S,blue),r(S)
```



```
> rotation(line(point(0,0,0),point(0,0,1)), 2*pi/3,S)
```



27.12.5 Homothety in space

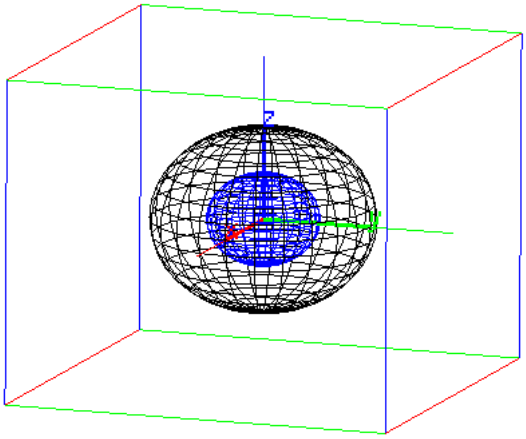
See Section 26.14.5, p. 734 for homotheties in the plane.

A homothety is a dilation about a given point. The `homothety` command creates a homothety.

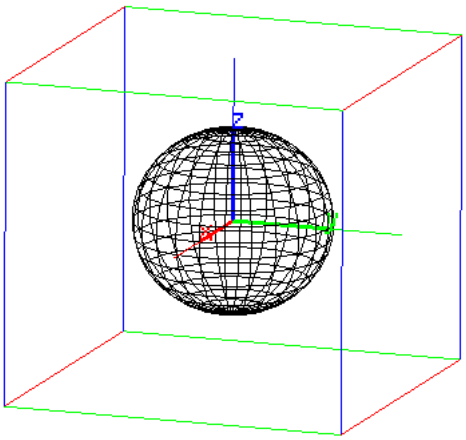
- `homothety` takes two mandatory arguments and one optional argument:
  - P, a point (the center of the homothety).
  - r, a number (the scaling ratio).
  - Optionally, G, a geometric object.
- `homothety`(P, r) returns a new command which dilates about P by a factor of r. If r is complex, this will rotate as well as scale.
- `homothety`(P, r, G) returns and draws the dilation of G about P by a factor or r.

Examples

```
> h:=homothety(point(0,0,0),2);
  S:=sphere([0,0,0],0.5);
  color(S,blue),h(S)
```



```
> homothety(point(0,0,0), 2, S)
```



27.12.6 Similarity in space

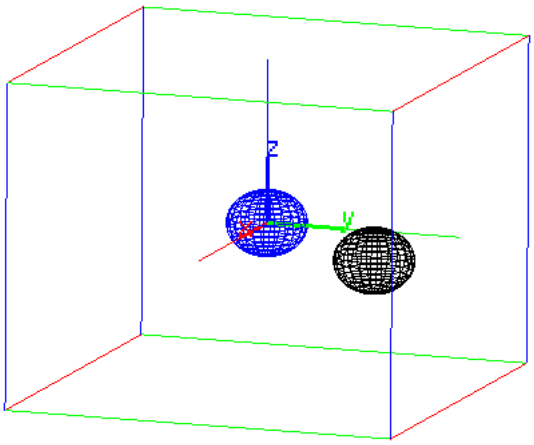
See Section 26.14.6, p. 735 for similarities in the plane.

The `similarity` command creates a command to rotate and scale about a given line.

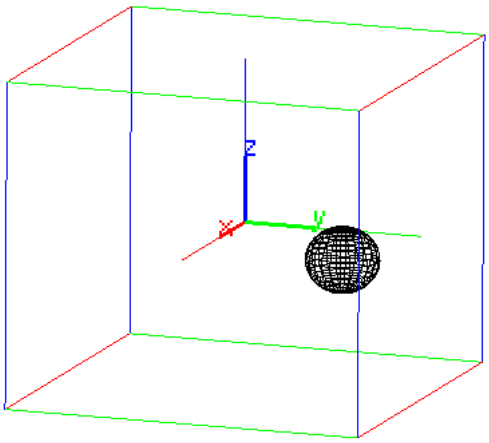
- `similarity` takes three mandatory arguments and one optional argument:
  - L, a line (the axis of the rotation).
  - r, a real number (the scaling ratio).
  - θ, a real number (the angle of rotation).
  - Optionally, G, a geometric object.
- `similarity`(L, r, θ) returns a new command which rotates about L through an angle of θ and scales about L by a factor of r. If r is negative, the direction of rotation is reversed.
- `similarity`(L, r, θ, G) returns and draws the transformation of G.

Examples

```
> S:=sphere([0,0,0],0.5);
  s:=similarity(line(point(0,1,0),point(0,1,1)), 2, 2*pi/3);
  color(S,blue),s(S)
```




```
> similarity(line(point(0,1,0),point(1,1,1)),2,2*pi/3,S))
```



27.12.7 Inversion in space

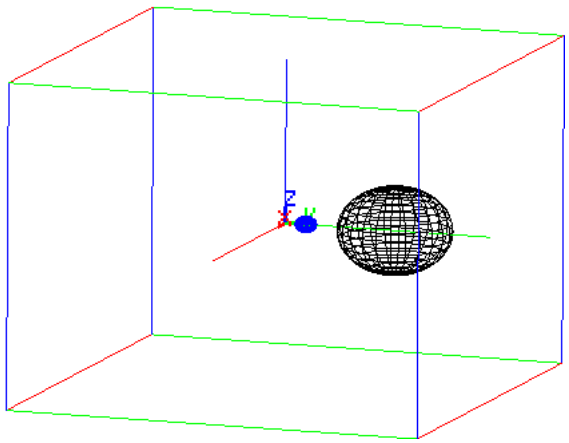
See Section 26.14.7, p. 735 for inversions in the plane.

Given a point P and a real number k, the corresponding *inversion* of a point A is the point A' on the ray $\overrightarrow{PA}$ satisfying $\overline{PA} \cdot \overline{PA'} = k^2$. The `inversion` command creates inversions.

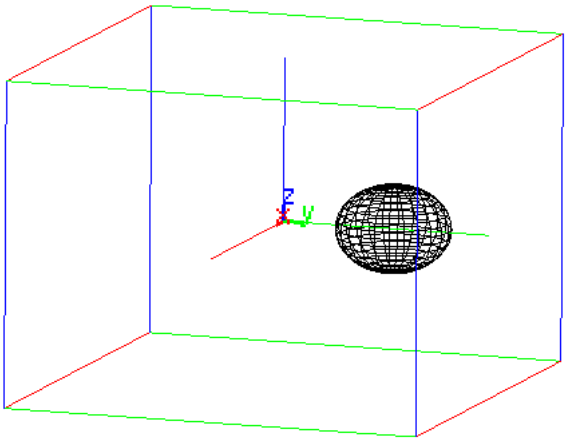
- `inversion` takes two mandatory and one optional argument:
  - P, a point.
  - k, the inversion ratio.
  - Optionally, G, a geometric object.
- `inversion(`P, k`)` returns a new command which does an inversion about P with a ratio k.
- `inversion(`P, k, G`)` returns and draws the inversion of G.

Examples

```
> S:=sphere([0,1,0],0.5);
  inver:=inversion(point(0,0,0), 2);
  color(S,blue),inver(S)
```



```
> inversion(point(0,0,0),2,S)
```



27.12.8 Orthogonal projection in space

See Section 26.14.8, p. 736 for projections in the plane.
The `projection` command creates a projection.

- `projection` takes one mandatory argument and one optional argument:
  - O, a geometrix object.
  - Optionally, P, a point.
- `projection(`O`)` returns a new command which projects points onto O.
- `projection(`O, P`)` returns and draws the projection of P onto O.

Examples

```
> P:=point(0,0,1);
  p1:=projection(line(point(0,0,0),point(1,1,1)));
  coordinates(p1(P))
```

$$\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right]$$

which is the projection of $(0, 0, 1)$ onto the line.

```
> coordinates(projection(plane(point(1,0,0),point(0,0,0),point(1,1,1)),point(0,0,1)))
```

$$\left[0, \frac{1}{2}, \frac{1}{2}\right]$$

which is the projection of the point $(0, 0, 1)$ onto the plane.

27.13 Surfaces

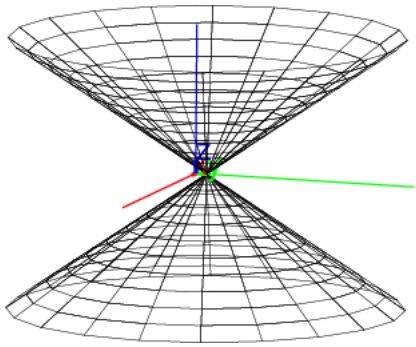
27.13.1 Cones

The `cone` command creates cones.

- `cone` takes three arguments:
  - A, a point.
  - v, a direction vector.
  - θ, a real number.
- `cone`(A, v, θ) returns and draws the cone with vertex A, opening in the direction v with an aperture of 2θ.

Example

```
> cone([0,1,0],[0,0,1],pi/3)
```



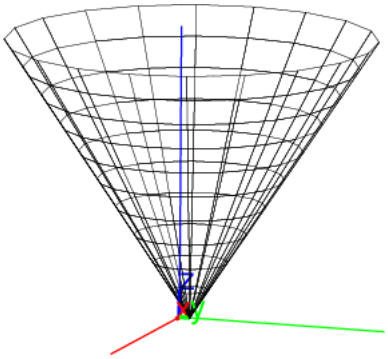
27.13.2 Half-cones

The `half_cone` command creates half-cones.

- `half_cone` takes three arguments:
  - A, a point.
  - v, a direction vector.
  - θ, a real number.
- `half_cone`(A, v, θ) returns and draws the half cone with vertex A, opening in the direction v with an aperture of 2θ.

Example

```
> half_cone([0,1,0],[0,0,1],pi/3)
```



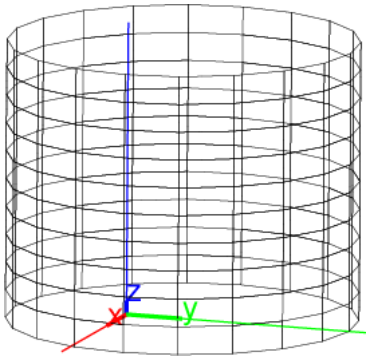
27.13.3 Cylinders

The `cylinder` command creates cylinders.

- `cylinder` takes three arguments:
  - A, a point.
  - v, a direction vector.
  - r, a real number.
- `cylinder`(A, v, r) returns and draws the cylinder with axis through A in the direction v with a radius of r. 2θ.

Example

```
> cylinder([0,1,0],[0,0,1],3)
```



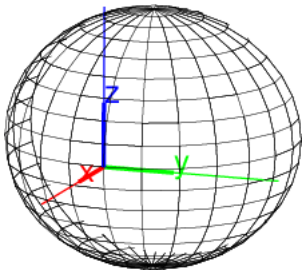
27.13.4 Spheres

The `sphere` command creates spheres.

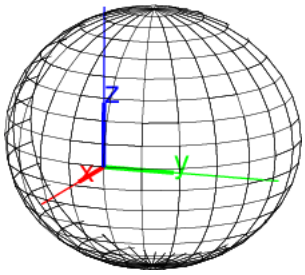
- `sphere` takes two arguments: P, R, either two points or a point and a real number.
- `sphere`(P, R) returns:
  - a sphere with diameter PR, if R is a point.
  - a sphere with center P and radius R, if R is a number.

Examples

```
> sphere([-2,0,0],[2,0,0])
```




```
> sphere([0,0,0],2)
```



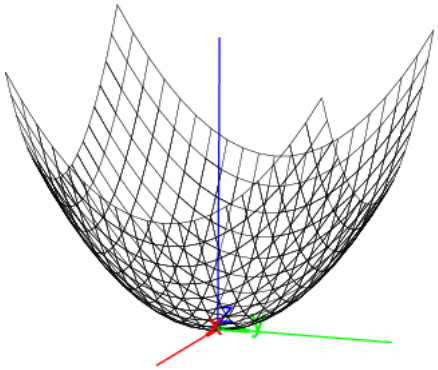
27.13.5 Graph of a function of two variables

The `funcplot` or `plotfunc` command can draw the graphs of two variable functions (see Section 19.2.2, p. 465 for a full description).

`funcplot` can take two arguments, an expression with two variables and a list of the two variables (possibly with bounds) and it returns and draws the graph of the expression.

Example

```
> funcplot(x^2+y^2,[x,y])
```



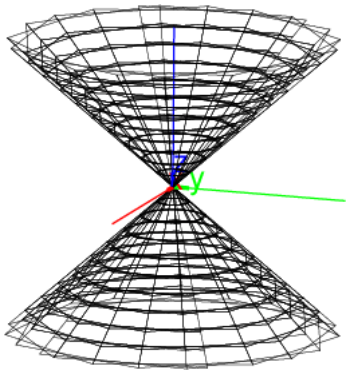
27.13.6 Graph of parametric equations in space

The `paramplot` command can draw a parametric surface in $\mathbb{R}^3$ (see Section 19.7.2, p. 483 for a full description).

`paramplot` can take two arguments; a list of three expressions involving two parameters and a list of the parameters (possibly with bounds), and it returns and draws the parameterized surface.

Example

```
> paramplot([u*cos(v),u*sin(v),u],[u,v])
```



27.14 Solids

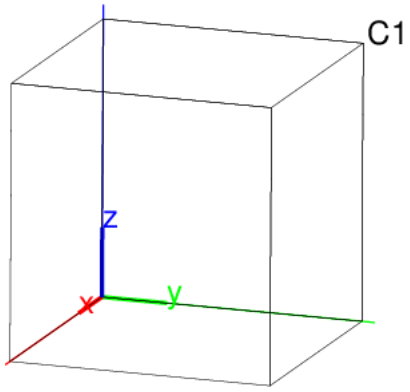
27.14.1 Cubes

The `cube` command creates cubes.

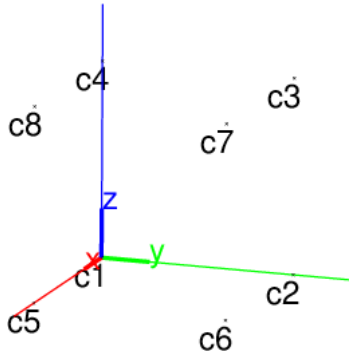
- `cube` takes three arguments: A, B, C, three points.
- `cube`(A, B, C) returns and draws the following cube:
  - One edge is AB.
  - One face is in the plane ABC, on the same side of line AB as is C.
  - The cube is on the side of plane ABC that makes the points A, B and C counterclockwise.

Examples

```
> C1:=cube([0,0,0],[0,4,0],[0,0,1])
```



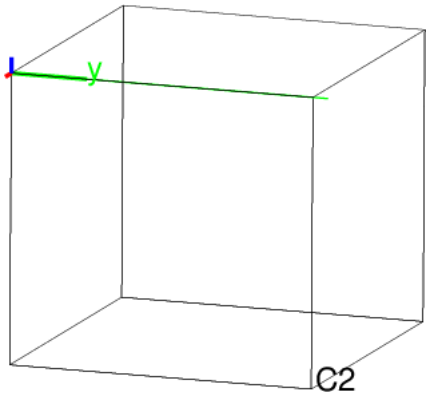

```
> c1,c2,c3,c4,c5,c6,c7,c8:=vertices(C1)
```



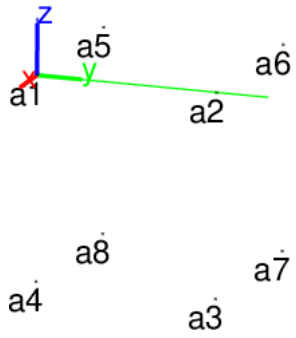

```
> faces(C1)
```

$$[[[0,0,0],[0,4,0],[0,4,4],[0,0,4]],$$
$$[[4,0,0],[4,4,0],[4,4,4],[4,0,4]],$$
$$[[0,0,0],[4,0,0],[4,0,4],[0,0,4]],$$
$$[[0,0,0],[0,4,0],[4,4,0],[4,0,0]],$$
$$[[0,4,0],[0,4,4],[4,4,4],[4,4,0]],$$
$$[[0,0,4],[4,0,4],[4,4,4],[0,4,4]]]$$

```
> C2:=cube([0,0,0],[0,4,0],[0,0,-1])
```




```
> a1,a2,a3,a4,a5,a6,a7,a8:=vertices(C2)
```



27.14.2 Tetrahedrons

The `tetrahedron` or `pyramid` command creates tetrahedra.

- `tetrahedron` command takes three or four arguments:

  - A, B, C, three points.
  - Optionally D, another point.

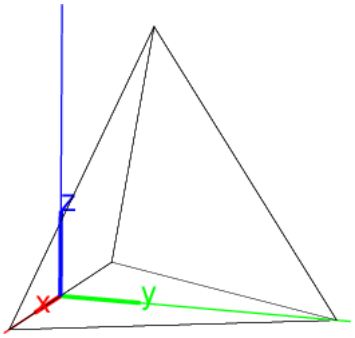
- `tetrahedron`(A, B, C) returns and draws the regular tetrahedron given by:
  - One edge is AB.
  - One face is in the plane ABC, on the same side of line AB as is C.
  - The tetrahedron is on the side of plane ABC that makes the points A, B and C counter-clockwise.
- `tetrahedron`(A, B, C, D) returns and draws the tetrahedron $ABCD$.

Examples

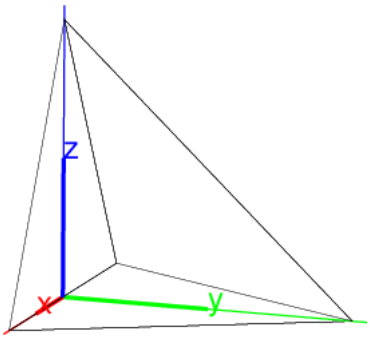
```
> tetrahedron([-2,0,0],[2,0,0],[0,2,0])
```

or:

```
> pyramid([-2,0,0],[2,0,0],[0,2,0])
```




```
> tetrahedron([-2,0,0],[2,0,0],[0,2,0],[0,0,2])
```



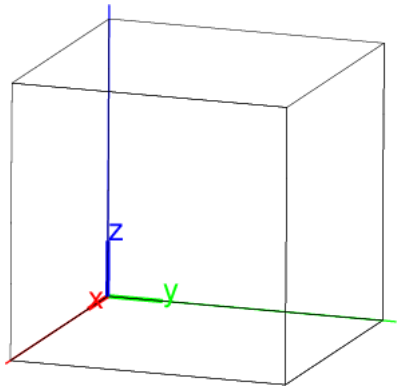
27.14.3 Parallelepipeds

The `parallelepiped` command creates parallelepipeds.

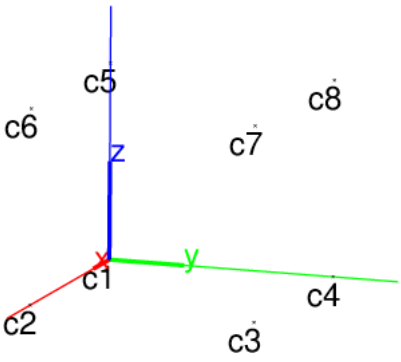
- `parallelepiped` takes four arguments: A, B, C, D, four points.
- `parallelepiped`(A, B, C, D) returns and draws the parallelepiped determined by the edges AB, AC and AD.

Examples

```
> parallelepiped([0,0,0],[5,0,0],[0,5,0],[0,0,5])
```




```
> p:=parallelepiped([0,0,0],[5,0,0],[0,3,0],[0,0,2]):;
  c1,c2,c3,c4,c5,c6,c7,c8:=vertices(p);
```



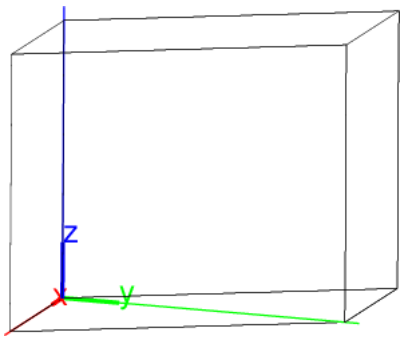
27.14.4 Prisms

The `prism` command creates prisms.

- `prism` takes two arguments:
  - a list of coplanar points `[A,B,...]`.
  - An additional point `A1`.
- `prism` returns and draws the prism whose base is the polygon determined by the points `A`, `B`, ..., and with edges parallel to `AA1`.

Example

```
> prism([[0,0,0],[5,0,0],[0,5,0],[-5,5,0]],[0,0,5])
```

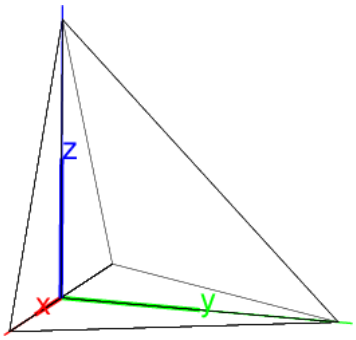


27.14.5 Polyhedra

The `polyhedron` command takes as argument a sequence of points.

`polyhedron` returns and draws the convex polygon whose vertices are from the list of points such that the remaining points are inside or on the surface of the polyhedron. For example:

```
> polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2])
```



27.14.6 Vertices

The `vertices` command takes as argument a polyhedron.

`vertices` returns and draws a list of the vertices of the polyhedron. For example:

```
> V:=vertices(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))
```

then:

```
> coordinates(V)
```

$$[[0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]]$$

27.14.7 Faces

The `faces` command takes as argument a polyhedron.

`faces` returns a list of the faces of the polyhedron. For example:

```
> faces(polyhedron([1,-1,0],[1,1,0],[0,0,2],[0,0,-2],[-1,1,0],[-1,-1,0]))
```

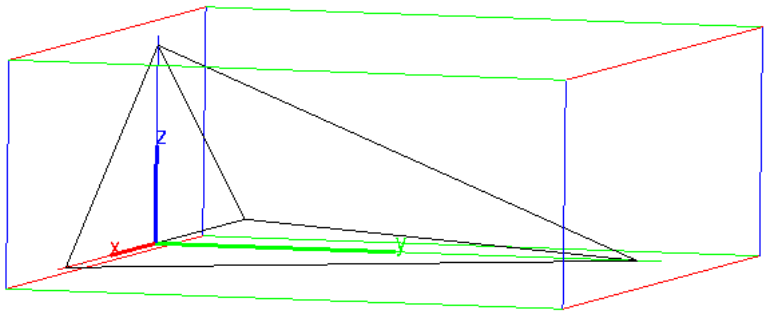
$$[[[1,-1,0],[1,1,0],[0,0,2]],[[1,-1,0],[1,1,0],[0,0,-2]],$$
$$[[1,-1,0],[0,0,2],[-1,-1,0]],[[1,-1,0],[0,0,-2],[-1,-1,0]],$$
$$[[1,1,0],[0,0,2],[-1,1,0]],[[1,1,0],[0,0,-2],[-1,1,0]],$$
$$[[0,0,2],[-1,1,0],[-1,-1,0]],[[0,0,-2],[-1,1,0],[-1,-1,0]]]$$

27.14.8 Edges

The `line_segments` command takes as argument a polyhedron.

`line_segments` returns and draws the list of the edges of a polyhedron. For example:

```
> line_segments(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))
```



Individual line segments can be obtained by using the `[]` operator. For example, the command line

```
> line_segments(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))[1]
```

returns the second segment.

27.15 Platonic solids

To specify a Platonic solid, Xcas works with the center, a vertex and a third point to specify a plane of symmetry. To speed up calculations, it may be useful to use approximate calculations, which can be ensured with the `evalf` command. For example, instead of:

```
> centered_cube([0,0,0],[3,2,1],[1,1,0])
```

it would typically be better to use:

```
> centered_cube(evalf([0,0,0],[3,2,1],[1,1,0]))
```

since it is faster.

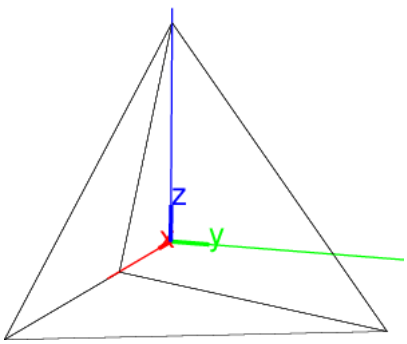
27.15.1 Centered tetrahedra

The `centered_tetrahedron` command creates a regular tetrahedron.

- `centered_tetrahedron` command takes three arguments: A, B, C, three points.
- `centered_tetrahedron`(A, B, C) returns and draws the tetrahedron centered at A, with a vertex at B and another vertex on the plane ABC.

Example

```
> centered_tetrahedron([0,0,0],[0,0,6],[0,1,0])
```



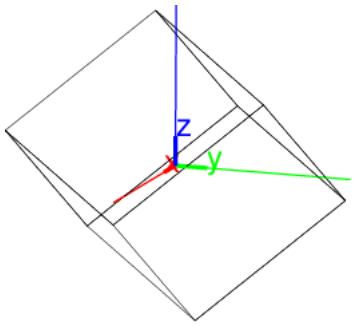
27.15.2 Centered cubes

The `centered_cube` command draws a cube.

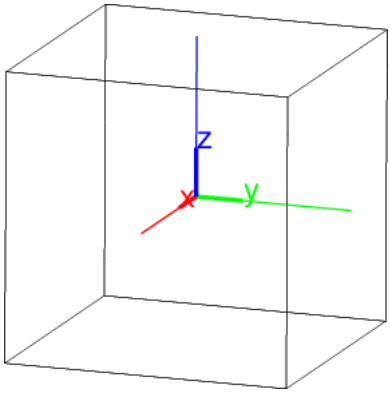
- `centered_cube` takes three arguments: A, B, C, three points.
- `centered_cube`(A, B, C) returns and draws the cube centered at A with a vertex at B and plane of symmetry ABC. This plane of symmetry has an edge of the cube containing B, the other endpoint of this edge is on the same side of line AB as C.

Examples

```
> centered_cube([0,0,0],[3,3,3],[0,1,0])
```




```
> centered_cube([0,0,0],[3,3,3],[0,-1,0])
```



Note that there are two cubes centered at A with a vertex at B and with a plane of symmetry ABC. Each cube has an edge containing B that's contained in plane of symmetry, these edges are on opposite sides of the line AB. The cube that `cube`(A, B, C) returns is the cube whose edge is on the same side of AB as the point C.

27.15.3 Octahedra

The `octahedron` command creates regular octahedra.

- `octahedron` takes three arguments: A, B, C, three points.
- `octahedron`(A, B, C) returns and draws the octahedron centered at A which has a vertex at B and with four vertices in the plane ABC.

Example

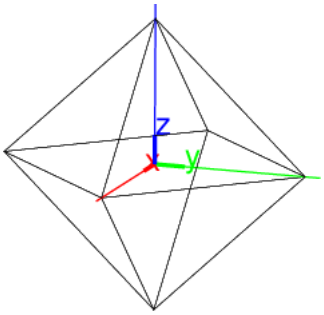
```
> octahedron([0,0,0],[0,0,5],[0,1,0])
```

or:

```
> octahedron([0,0,0],[0,5,0],[0,0,1])
```

or:

```
> octahedron([0,0,0],[5,0,0],[0,0,1])
```



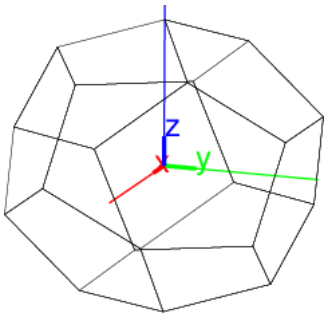
27.15.4 Dodecahedra

The `dodecahedron` command creates a regular dodecahedron.

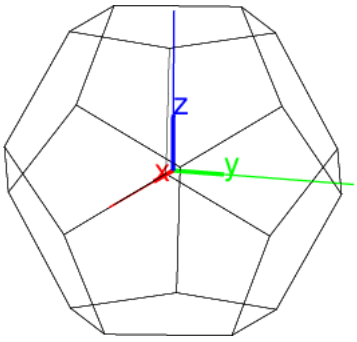
- `dodecahedron` takes three arguments: A, B, C, three points.
- `dodecahedron`(A, B, C) returns and draws the dodecahedron centered at A with a vertex at B and with an axis of symmetry in the plane ABC. (Note that each face is a pentagon, but will be drawn with one of its diagonals and so will show up as a trapezoid and a triangle.)

Examples

```
> dodecahedron([0,0,0],[0,0,5],[0,1,0])
```




```
> dodecahedron([0,0,0],[0,2,sqrt(5)/2+3/2],[0,0,1])
```



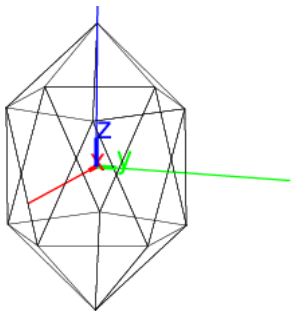
27.15.5 Icosahedra

The `icosahedron` command creates regular icosahedra.

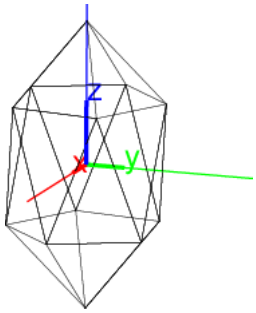
- `icosahedron` takes three arguments: A, B, C, three points.
- `icosahedron`(A, B, C) returns and draws the icosahedron centered at A with a vertex at B and such that the plane ABC contains one of the vertices closest to B.

Examples

```
> icosahedron([0,0,0],[0,0,5],[0,1,0])
```




```
> icosahedron([0,0,0],[0,0,sqrt(5)],[2,1,0])
```



28 Multimedia

28.1 Images

28.1.1 Image structure in Xcas

Starting from version 1.9.0, images in Xcas are special objects which can be loaded from PNG, JPEG or GIF files and written to PNG files. Such objects allow for visualization, extraction and modification of the image data. An image object evaluates to a string describing the structure of the image.

In earlier versions of Xcas, an image is a list with the following elements (referred to as the *legacy format* in the further text):

1. A list of three integers:
   - the number of channels (3 or 4, depending on whether a transparency channel is present),
   - the number n, representing the height of the image in pixels,
   - the number p, representing the width of the image in pixels.

   Each channel will be an $n \times p$ matrix of integers between 0 and 255.
2. A matrix of red channel values.
3. A matrix of green channel values.
4. A matrix of transparency channel values.
5. A matrix of blue channel values.

The color of the point at line i and column j is determined by the values of the (i, j)th entry of the matrices. Note that the transparency channel data comes before the blue channel data since color values in Xcas are `red` $= 1$, `green` $= 2$ and `blue` $= 4$, which allows for extraction of channel data using colors as indices.

Note that the number of points in the structure is not necessarily the same as the number of pixels on the screen when it is drawn. A single point in the structure can appear as a (small) rectangle of pixels when displayed on the screen.

Loading images to the legacy format is supported by the `readrgb` command.

28.1.2 Creating and loading images

The `image` command creates image objects from files or data.

- `image` accepts either a string *filename* from which the image is loaded or two mandatory arguments for creating images from data:
  - n, a positive integer in $\{1, 2, 3, 4\}$.
  - *data*, a sequence of numeric matrices of the same size which contain the channel data. Each matrix must have integral values between 0 and 255, representing 8-bit color values.
- `image(`*filename*`)` or `image(`n, *data*`)` returns the corresponding image object.

- The parameter n represents the color type of the image (1 – grayscale image, 2 – grayscale image with alpha channel, 3 – RGB image, 4 – RGBA image).
- If the number of matrices in *data* is smaller than n, then it can be either:
  - 1, for writing the same data to all color channels and 255 to the alpha channel if there is one.
  - 2, for writing the same data from the first matrix to all color channels and the values from the second matrix to the alpha channel (this requires $n \in \{2, 4\}$).
  - 3, for writing the data to R, G, and B channels and the value 255 to the alpha channel.

Examples

```
> img:=image("/usr/local/share/giac/doc/logo.png")
```

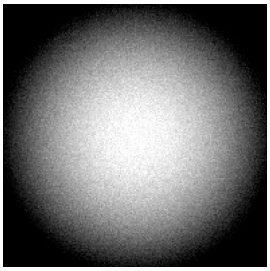
an image of size 64×64 (RGBA)

The command lines below create a simple noisy grayscale image with $n \times n$ pixels.

```
> n:=200:;
  p:=(j,k)->round(max(0,min(255,1000*((j-n/2)^2+(k-n/2)^2)/n^2+randnorm(0,10)))):;
  data:=matrix(n,n,p):;
  img:=image(1,data)
```

an image of size 200×200 (grayscale)

```
> display(inv(img))
```



We are displaying the negative image (`inv` is applied) to have an impression of a square-shaped picture.

28.1.3 Loading images to the legacy format

The `readrgb` command reads an XCAS image structure (see Section 28.1.1, p. 812).

- `readrgb` command takes *filename*, the name of an image file in JPEG, PNG or GIF format.
- `readrgb(`*filename*`)` returns the legacy XCAS image structure for the image in *filename*.

Example

Assume the the image shown in Figure 28.1 is stored in file `image1234.jpg`. After reading it into a variable name with `readrgb` using the legacy XCAS format:

```
> a:=readrgb("image1234.jpg")
```

the variable `a` will contain a list.

- `a[0]` will be `[4,250,500]`, the number of channels, height, and width of the image.
- `a[1]`, the red channel,

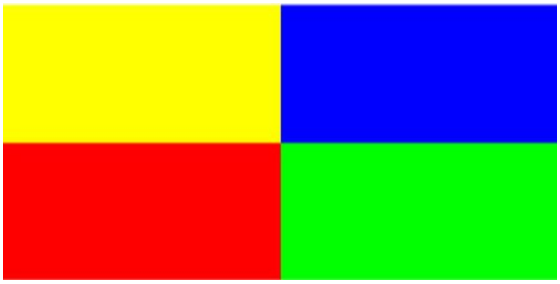


Figure 28.1: An image of size 250 × 500 with colors

- `a[2]`, the green channel,
- `a[3]`, the transparency channel,
- `a[4]`, the blue channel.

28.1.4 Viewing images

Starting from Xcas version 1.9.0, image objects are viewed by using the `display` command. The legacy Xcas image structure can be visualized by exporting it to a file by using the `writergb` command (see Section 28.1.5, p. 816).

- `display` takes one mandatory arguments and one optional argument:
  - *img*, an image object.
  - Optionally, either a, b, a pair of real numbers, or $a + \mathrm{i}b$, a complex number (by default, $a = b = 0$).
- `display(`*img*$\langle, a, b\rangle$`)` or `display(`*img*$\langle, a + \mathrm{i}b\rangle$`)` returns a graphic object showing *img* with (a, b) corresponding to its lower-left corner. The axes are automatically hidden and the view is orthonormalized in order to preserve the original aspect ratio of the image.

In the examples that follow, we assume that the current directory in Xcas is changed (by using the `cd` command, see Section 4.2.1, p. 48) to `/usr/local/share/giac/examples` in Linux resp. to `C:\xcaswin\examples` in Windows.

Example

```
> img:=image("Exemples/demo/terre.jpg")
```

an image of size 512×256 (RGB)

```
> display(img)
```

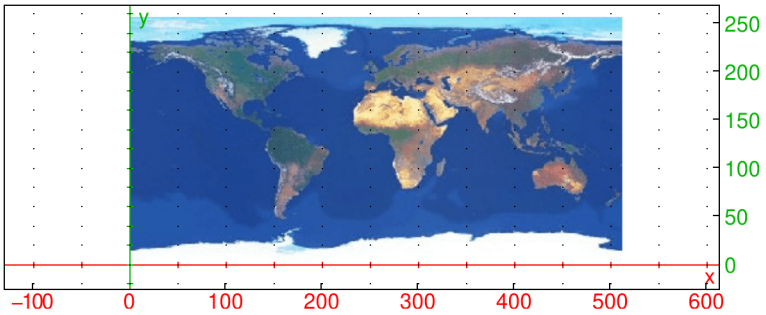
Multiple images can be shown together by calling `display` several times within a single command line (delimited by `;`) and adjusting a and b to appropriate values in each call. If desired, individual image captions can be inserted by using the `legend` command (See Section 21.4.10, p. 605 for an example.) There is also a possibility of combining images with other graphical objects.

```
> display(img);
  legend(285+125i,"Africa",color=gold); legend(270+200i,"Europe",color=gold);
  legend(380+190i,"Asia",color=yellow); legend(430+90i,"Australia",color=yellow);
  legend(100+190i,"North America",color=gold); legend(160+80i,"South America",color=gold);
```

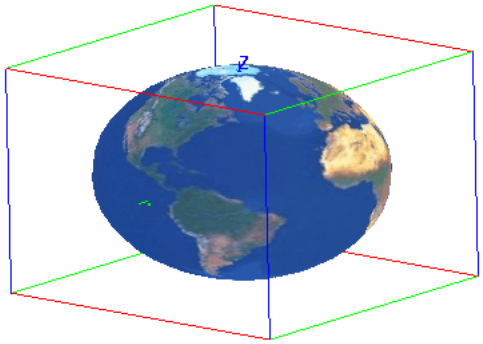
Viewing images as textures. Xcas can also display image files in rectangles in 2D or on surfaces in 3D with the `gl_texture` property of the object (see Section 19.1.2, p. 454). This procedure does not use image objects, but reads images from disk.

Examples

```
> rectangle(0,200,1/2,gl_texture="Exemples/demo/terre.jpg")
```




```
> sphere([0,0,0],1,gl_material=[gl_texture,"Exemples/demo/terre.jpg"])
```



28.1.5 Exporting images

The `writergb` command writes images to PNG files. The image can be either an image object or given by the full Xcas legacy image structure (see Section 28.1.1, p. 812, this is what is read in with `readrgb`) or a simplified version of this structure.

- `writergb` takes two arguments:
  - *filename*, a file name.
  - *image*, an image in Xcas format (either the new or legacy one).
- `writergb(`*filename*, *image*`)` writes the image *image* to the file *filename*. If *image* is an image object, `writergb` returns 1 if the export was successful, otherwise it returns 0. If *image* is the image structure (either full or simplified), then `writergb` returns its graphical representation.

Examples

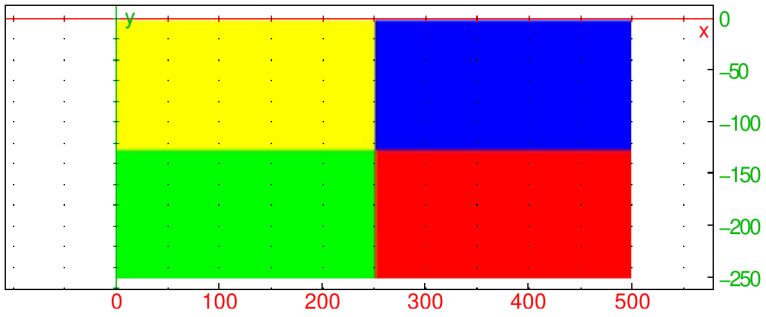
With image objects. Assuming that the `img` variable contains an image object created with the `image` command (see Section 28.1.2, p. 812), input:

```
> writergb("image.png",img)
```

1

With the legacy format. Assume that the variable `a` represents a legacy image structure loaded from the file `image1234.jpg` by using `readrgb` (see the example in Section 28.1.3, p. 813). Then:

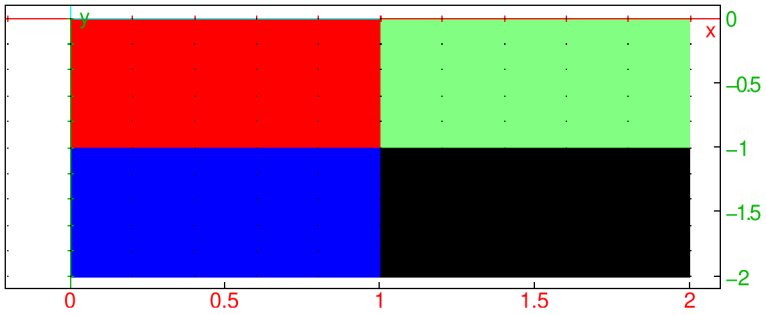
```
> writergb("image2134.png",[a[0],a[2],a[1],a[3],a[4]])
```



The image file `image2134.png` is thus created. This image is simply the `image1234.png` image with the green and red colors switched.

In simpler cases, you can type the Xcas image format in by hand.

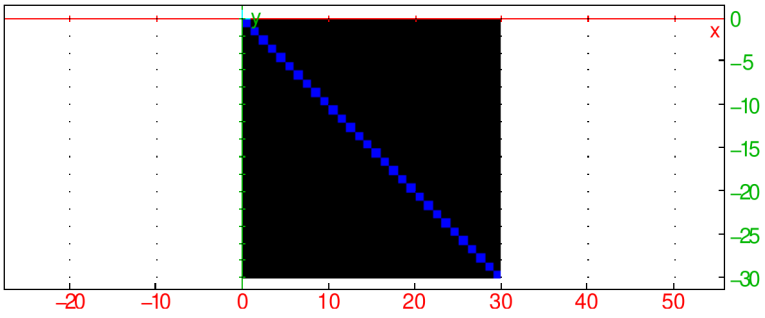
```
> writergb("image1.png",[[4,2,2],[[255,0],[0,0]],[[0,255],[0,0]],
                         [[255,125],[255,255]],[[0,0],[255,0]]])
```



The transparency value of 125 for the upper right point makes it partially transparent and mutes the color. Observe that individual pixels are shown as rectangles due to the automatic scaling.

For larger images, in some cases the matrix operations of Xcas can be used to create the channels. The following command line creates the file image2.png containing the image shown below (the view is orthonormalized).

```
> writergb("image2.png",[[4,30,30],makemat(0,30,30),makemat(0,30,30),
    makemat(255,30,30),makemat(0,30,30)+idn(30)*255])
```

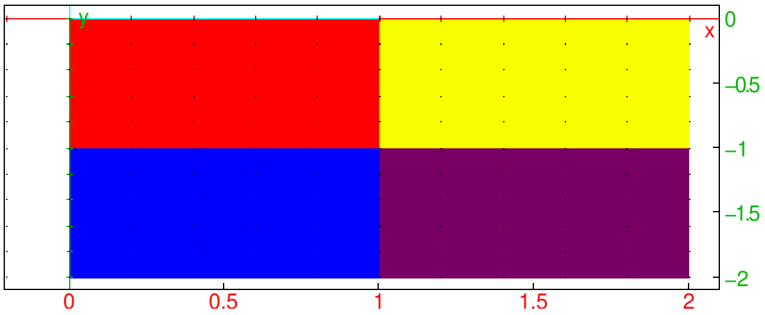


Writing images with simplified image description. The simplified version of the Xcas image description does not involve stating the number of channels, the size of the image, or the transparency. There is a full color version of this simplified form and a grayscale version.

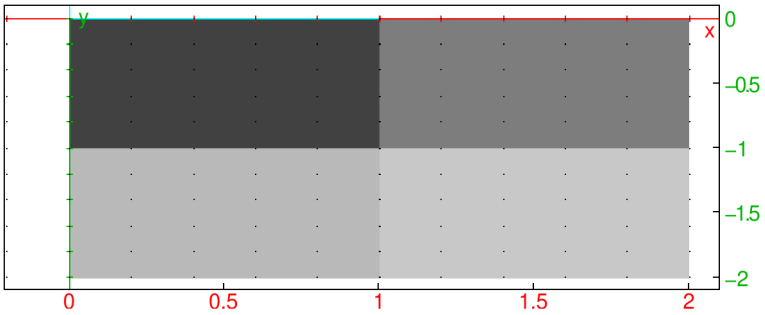
- To create a full color image using the simple description, writergb takes four arguments:
  - filename, the name of the file to store the image.
  - R, a matrix for the red channel.
  - G, a matrix for the green channel.
  - B, a matrix for the blue channel.
- writergb(*filename*, R, G, B) draws the image given by the matrices to the file filename.
- To create a grayscale image using the simple description, writergb takes two arguments:
  - *filename*, the name of the file to store the image.
  - M, a matrix representing how dark each point is (where 0 is black and 255 is white).
- writergb(*filename*, M) draws the image given by M to the file filename.

Examples

```
> writergb("image3.png",[[255,250],[0,120]],[[0,255],[0,0]],[[0,0],[255,100]])
```



```
> writergb("image4.png",[[65,125],[185,200]])
```



28.1.6 New images from existing ones

The `()` operator can be used to create image objects from an existing one. With an image object `img`:

- `img(grey)` returns a copy of `img` converted to grayscale.
- `img(`α`)`, where α is a positive real number, returns a copy `img` with size scaled to $\alpha w \times \alpha h$.
- `img(`w, h`)` returns a copy of `img` resized to $w \times h$ (possibly changing the aspect ratio).
- `img(`x, y, w, h`)` returns a new image corresponding to the portion of `img` with dimensions $w \times h$ and the upper left corner at (x, y).

Note that `img` is unaffected by the above operations.

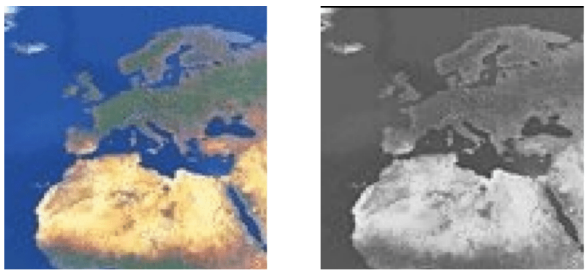
Examples

Assume that the image `terre.jpg` (see Section 28.1.4, p. 814) is loaded to the variable `img`. Then:

```
> cropped:=img(220,20,100,100)
```

an image of size 100×100 (RGB)

```
> display(cropped,0); display(cropped(grey),120)
```



```
> title:="resized from 512 x 256 to 102 x 51"; display(img(0.2))
```

resized from 512 x 256 to 102 x 51

28.1.7 Modifying images

Setting pixel data explicitly. Existing image objects can be modified by using the `set_channel_data` command.

- `set_channel_data` takes four or five arguments:
  - *img*, an image object.
  - *chn*, a channel index, which should be either a nonnegative integer or `red` for the red channel, `green` for the green channel, `blue` for the blue channel, `grey` for the grayscale channel resp. `tran` for the transparency channel.

    For grayscale images without a transparency channel, this argument should be omitted since there is only one channel.
  - x, an integer.
  - y, an integer.
  - *data*, a matrix of integers between 0 and 255.
- `set_channel_data(`*img*$\langle$`,` *chn*$\rangle$`,` x`,` y`,` *data*`)` returns a modified copy of *img* in which the indicated channel components of pixels in the rectangle with upper left corner (x, y) and lower right corner $(x + w - 1, y + h - 1)$, where w resp. h is the number of columns resp. rows of the matrix *data*, are set to the values in *data*; i.e. the *chn* component in the pixel $(x + j, y + i)$ in is set to $data_{ij}$ for $i = 0, \ldots, h - 1$ and $j = 0, \ldots, w - 1$.

Inverting pixel data. To produce a negative of an image, i.e. to apply the function $p \mapsto 255 - p$ to all channel data entries, use the unary operator `-` or the command `inv`.

28.1.8 Extracting data from images

Printing out information about an image. The `about` command can be used to print out a brief description of an image object, which it takes as its only argument.

Obtaining the size of an image. The size of an image can be obtained by using the `size` command.

- `size` takes *img*, an image object.
- `size(`*img*`)` returns a list of two integers representing the width and the height of *img* (in that order).

Obtaining pixel data. Channel data can be extracted from image objects to matrices by using the `[]` operator or the `channel_data` command.

- Let *img* be an image object.
- The command *img*`[`*chn*$\langle$`,` x`,` y $\langle$`,` w`,` h $\rangle\rangle$`]`, where
  - *chn* is a channel index, which should be either a nonnegative integer or `red` for the red channel, `green` for the green channel, `blue` for the blue channel, `grey` for the grayscale channel resp. `tran` for the transparency channel,
  - (optionally) x, y are integers and
  - (optionally) w, h are positive integers (if these are given, then x and y must also be given),

  returns
  - the full matrix of the channel *chn* if no optional arguments are given,

  - the *chn* component of the pixel (x, y) if w and h are not given,
  - the matrix of pixel data for the channel *chn* in the rectangle with upper left corner (x, y) and lower right corner $(x + w - 1, y + h - 1)$ if x, y, w and h are given.

- If *chn* = `grey` and *img* is a color image, then the returned pixel value(s) are obtained by averaging the color channels.

- The `channel_data` command extracts a channel (sub)matrix. Precisely, `channel_data(`*img*, *args*`)` is equivalent to *img*`[`*args*`]`, where *img* is an image object and *args* is a sequence of arguments as described above.

Flattening images. The `flatten` command returns a list representation of an image, which is suitable for e.g. passing the image to a neural network. It accepts a single argument, an image object *img*, and returns the list in which the pixel data is stored row by row, with individual channel values for the same pixel occupying consecutive entries in the list. (See the MNIST example in Section 22.3.2, p. 625.)

28.1.9 Trimming image margins

Margins of an image, consisting of "empty" pixels with value 0 in all color channels, can be found and removed by using the `trim` command (see Section 5.2.5, p. 57, Section 21.2.2, p. 569 and Section 28.2.2, p. 823 for other usages of `trim`).

- `trim` takes one mandatory argument and one or two optional arguments:
  - *img*, an image object.
  - Optionally, `inv`, the symbol.
  - Optionally, `index`, the symbol.

- `trim(`*img*⟨`,` `inv`⟩⟨`,` `index`⟩`)` finds largest zero-pixel margins in *img* or its inversion if `inv` is given. If `index` is given, then the trimmed slice location (x, y) and dimensions (w, h) are returned as a sequence (x, y, w, h). Otherwise, a trimmed version of *img* is returned.

- To remove white margins, include the option `inv` when calling `trim`.

Example

```
> img:=image("/home/luka/Downloads/mnist_png/training/1/552.png")
```

an image of size 28×28 (grayscale)

```
> display(img)
```

```
> display(trim(img))
```

```
> trim(img,index)
```

$$7, 3, 11, 20$$

28.1.10 Blending images

The `interp` command interpolates two images linearly in the XYZ color space.

- `interp` takes three arguments:
  - *img1*, an image object with dimensions $w_1 \times h_1$ and color depth d_1.
  - *img2*, an image object with dimensions $w_2 \times h_2$ and color depth d_2.
  - t, a real number in $[0, 1]$.
- `interp`(*img1*, *img2*, t) returns the image with dimensions $w \times h$ and color depth d, obtained by linear interpolation between *img1* and *img2*. The width an height of the resulting image are computed by solving the system of equations

$$wh = (1-t)w_1h_1 + tw_2h_2, \quad \ln\frac{w}{h} = (1-t)\ln\frac{w_1}{h_1} + t\ln\frac{w_2}{h_2}.$$

  The resulting color depth d is the smallest one that is able to fit data from each image. For example, if *img1* is a grayscale image with alpha channel ($d_1 = 2$) and *img2* is a RGB image ($d_2 = 3$) with three channels and no alpha channel, then the resulting depth will be RGBA, i.e. $d = 4$ (three color channels with alpha channel).

  To interpolate pixel data, `interp` first resizes both images to $w \times h$. The RGB values of individual pixels are then computed by a perceptually uniform linear interpolation between the corresponding values in *img1* and *img2* (see Section 19.1.3, p. 462).

Example

To begin, load two images:

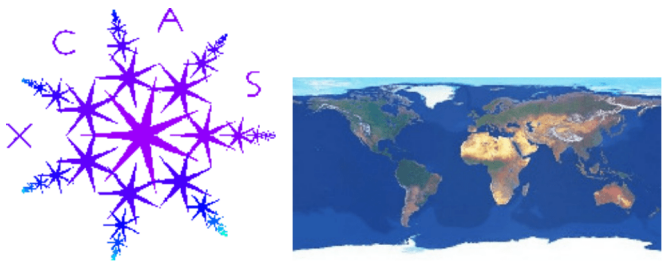
```
> img1:=image("/home/luka/Pictures/xcas-logo.png")
```

an image of size 377×353 (RGB)

```
> img2:=image("/usr/local/share/giac/examples/Exemples/demo/terre.jpg")
```

an image of size 512×256 (RGB)

```
> display(img1); display(img2,400)
```

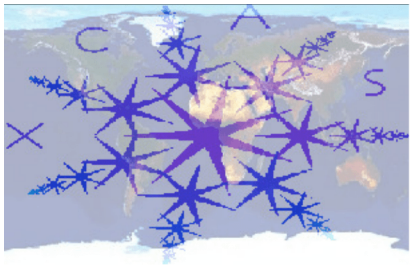


Now blend the two images together with $t = 0.6$:

```
> img:=interp(img1,img2,0.6)
```

an image of size 474×277 (RGB)

```
> display(img)
```



28.2 Audio

Xcas has support for audio objects in `wav` (Waveform Audio File) format. The following commands are provided for working with audio objects.

- For creating and playing audio objects, there are:
  - `createwav`, for creating audio objects (see Section 28.2.1, p. 823).
  - `playsnd`, for playing audio objects (see Section 28.2.14, p. 833).
- For reading and writing wave files, there are:
  - `readwav`, for reading audio files from disk to audio objects (see Section 28.2.3, p. 824).
  - `writewav`, for writing audio files to disk from audio objects (see Section 28.2.4, p. 824).
- For manipulating audio objects, there are:
  - `set_channel_data`, for modifying the contents of an audio clip (see Section 28.2.9, p. 827).
  - `normalize`, for scaling the amplitude of an audio clip (see Section 28.2.15, p. 833).
  - `stereo2mono`, for averaging channels to a single channel (see Section 28.2.5, p. 825).
  - `resample`, for changing of the sample rate (see Section 28.2.11, p. 829).
  - `reverse`, for reversing audio in time (see Section 28.2.2, p. 823).
  - `inv` or `-` (unary operator), for inverting channel data (see Section 28.2.2, p. 823).
  - `+` (infixed operator), for concatenation of audio clips (see Section 28.2.2, p. 823).
  - `splice`, for splicing audio clips together with a crossfade (see Section 28.2.17, p. 834).
  - `mixdown`, for mixing several clips down to one clip (see Section 28.2.18, p. 835).
- For getting information from an audio object, there are:
  - `about`, for printing out the properties of an audio clip (see Section 28.2.6, p. 825).
  - `length` or `size`, for obtaining the audio clip length in samples (see Section 28.2.6, p. 825).
  - `channels`, for finding the number of channels (see Section 28.2.6, p. 825).
  - `bit_depth`, for finding the number of bits in each sample value (see Section 28.2.6, p. 825).
  - `samplerate`, for finding the number of samples per second (see Section 28.2.6, p. 825).
  - `duration`, for finding the duration of the audio in seconds (see Section 28.2.6, p. 825).
  - `channel_data`, for extracting channel data from an audio clip (see Section 28.2.8, p. 826).
  - `rms`, for computing the RMS (root mean square) of an audio clip (see Section 28.2.16, p. 833).
  - `plotwav`, for visualizing a waveform (see Section 28.2.12, p. 829).
  - `plotspectrum`, for visualizing power spectra (see Section 28.2.13, p. 831).

28.2.1 Creating audio clips

The `createwav` command creates an audio object with specified parameters.

- `createwav` takes its arguments as `key=`*value* pairs, in no particular order. The following arguments are all optional:
  - `size=`n resp. `duration=`T, where n resp. T is the total number of samples resp. the length in seconds.
  - `bit_depth=`b, where b is the number of bits reserved for each sample value and may be 8, 16 or 24 (by default 16).
  - `samplerate=`r, where r is the number of samples per second (by default 44100).
  - `channels=`c where c is the number of channels (by default 1).
  - D or `channel_data=`D, where D is a list or a matrix.

    If D is a matrix, it should contain the kth sample in the jth channel at position (j, k). The value of each sample must be a real number in range $[-1.0, 1.0]$. Any value outside this interval is clamped to it (this effect is called *clipping* and it will distort the signal).

    If D is a single list, it is copied across channels.

    D may be truncated or padded with zeros to match the appropriate number of samples or seconds.
  - `normalize=`db, where $db \leq 0$ is a real number representing the amplitude peak level in dB FS (decibel "full scale") units. If this option is given, audio data is normalized to the specified level prior to conversion. This can be used to avoid clipping. By default, the resulting audio is normalized to the -1 dB level.
- `createwav(`⟨*keys*=*values*⟩`)` returns an audio object with the requested parameters.
- If no sound data is provided, the resulting audio will be silent (all samples will be initialized to 0).

Examples

(See Section 28.2.14, p. 833 for a description of `playsnd` and Section 28.2.7, p. 825 for a description of `soundsec`.)

```
> playsnd(createwav(channel_data=sin(2*pi*440*soundsec(2)),samplerate=48000))
```

Output: Two seconds of the 440 Hz sine wave at rate 48000.

```
> t:=soundsec(3):;
  L,R:=sin(2*pi*440*t),sin(2*pi*445*t):;
  s:=createwav([L,R]):;
  playsnd(s)
```

Output: Three seconds of a vibrato effect on a sine wave (stereo).

28.2.2 New audio clips from existing ones

You can create audio clips from existing ones by using some standard operators.

Slicing. Audio clips can be obtained from existing clips by slicing. To extract a portion of length n samples starting at kth sample in an audio clip `clip`, use the `()` operator like this:

```
> subclip:=clip(k,n)
```

The second argument may be omitted, in which case $n = L - k$ is used, where L is the length of `clip` in samples. You can also use

```
> subclip:=clip(t1..t2)
```

to extract the portion between timestamps t_1 and t_2 (in seconds).

Concatenation. Existing audio clips can be concatenated together by using the `+` operator. Note that clips to be concatenated must have the same number of channels, bit depth and sample rate. For example, to concatenate two audio clips `clip1` and `clip2`, input:

```
> clip:=clip1+clip2
```

This is useful for e.g. padding audio with zeros (silence).

Inverting and reversing. Audio clips can be inverted (which is equivalent to multiplying sample data with -1 for each channel) by using the unary sign operator `-` or the `inv` command which takes an audio clip as its only argument.

To reverse an audio clip in the time domain, use the `reverse` command which takes an audio clip as its only argument.

28.2.3 Reading `wav` files from disk

The `readwav` command loads a `wav` file.

- `readwav` takes *file*, a string containing the name of a `wav` file.
- `readwav(`*file*`)` loads *file* and returns an audio clip object.

Example

Assuming that the file `example.wav` is stored in the directory `sounds`, input:

```
> s:=readwav("/path/to/sounds/example.wav")
```

You can now play the audio clip object `s`:

```
> playsnd(s)
```

Output: The sound of the audio file `example.wav`.

28.2.4 Writing `wav` files to disk

The `writewav` command writes `wav` files to disk.

- `writewav` takes two arguments:
  - *filename*, a string containing a file name.
  - A, an audio clip object.
- `writewav(`*filename*$, A$`)` writes the clip A as a `wav` file named *filename*.

  It returns 1 on success and `0` on failure.

Example

```
> s:=createwav(sin(2*pi*440*soundsec(1))):;
  writewav("sounds/sine.wav",s)
```

1

The `sounds` directory now contains the file `sine.wav`.

28.2.5 Averaging channel data

The `stereo2mono` command converts a multichannel audio clip to a single channel audio clip.

- `stereo2mono` takes A, a multichannel audio clip.
- `stereo2mono(`A`)` returns an audio clip with the input channels in A mixed down to a single channel.

  Every sample in the output is the arithmetic mean of the samples at the same position in the input channels.

Example

Enter:

```
> t:=soundsec(3):;
  L,R:=sin(2*pi*440*t),sin(2*pi*445*t):;
  s:=stereo2mono(createwav([L,R]))
```

a sound clip with 132300 samples at 44100 Hz (16 bit, mono)

Then:

```
> playsnd(s)
```

Output: The sound of the single channel of the combination of L and R is played.

28.2.6 Audio clip properties

The `about`, `channels`, `bit_depth`, `samplerate` and `duration` commands find properties of an audio clip.

- `about`, `channels`, `bit_depth`, `samplerate` and `duration` each take one argument: A, an audio clip.
- `about(`A`)` prints the information pertinent to A (bit depth, number of channels, sample rate, length in samples, duration in seconds, the peak dB level, associated file name) and returns A.
- `channels(`A`)` returns the number of channels in A.
- `bit_depth(`A`)` returns the number of bits reserved for each sample value (8, 16, or 24) in A.
- `samplerate(`A`)` returns the number of samples per second in A.
- `length(`A`)` and `size(`A`)` both return the length of A in samples.
- `duration(`A`)` returns the duration of A in seconds.

28.2.7 Preparing digital sound data

The `soundsec` command prepares sound data in the form of a vector.

- `soundsec` takes one mandatory argument and one optional argument:
  - d, a real number (the duration).
  - Optionally, f, a real number (the sampling frequency, by default 44100).
- `soundsec(`d $\langle$`,` $f\rangle$`)` returns sound data with duration d seconds, and with sampling frequency f, as a vector whose ith element is the time corresponding to index i.

Examples

```
> soundsec(2.5)
```

Output: Sound data 2.5 seconds long sampled at the default frequency of 44100 Hz.

```
> soundsec(1,22050)
```

Output: Sound data 1 second long sampled at the frequency of 22050 Hz.

```
> sin(2*pi*440*soundsec(1.3))
```

Output: A sinusoidal signal with frequency 440 Hz sampled 44100 times per second for 1.3 seconds.

28.2.8 Extracting samples from audio clips

The `channel_data` command gets samples from an audio clip.

- `channel_data` takes one mandatory argument and up to two optional argument:
  - A, an audio clip.
  - Optionally `options`, which can be the following (the order is unimportant):
    - n, a postive integer (channel number, zero-based) or `matrix`, the symbol.
    - `range=[`*offset*,*length*`]` for nonnegative integer *offset* and positive integer *length*, or `range=`a`..`b for floating point numbers a and b.

    Instead of channel number n, the symbol `left` resp. `right` may be used to specify the left resp. right channel.

- `channel_data(`A ⟨`,`*options*⟩`)` returns data from the channels as a sequence of lists. The returned sample values are all within the interval $[-1.0, 1.0]$, i.e. the amplitude of the returned signal is relative. The maximum possible amplitude is represented by the value 1.0.
  - With no options, data from all channels are returned as a sequence of lists.
  - With the option n (the channel number) or if there is only one channel, only data from this channel is returned in a single list.
  - With the option `matrix`, the lists representing the channel data are returned as the rows of a matrix.
  - With the option `range=[`*offset*,*length*`]`, the block of *length* samples is extracted starting at *offset*.
  - With the option `range=`a`..`b, with floating point numbers a and b, then a and b are bounds in seconds.

- Alternatively, channel data can be extracted by using the `[]` operator on an audio clip A with one to three arguments:
  - n, a channel number (`left` and `right` are allowed).
  - Optionally, *offset*, an integer specifying the offset in samples (by default, $\mathit{offset} = 0$).
  - Optionally, *len*, a positive integer (by default, $\mathit{len} = L - \mathit{offset}$, where L is the length of A in samples).

  Instead of specifying *offset* and *len* parameters, you can enter an interval t_1`..`t_2 as the second argument. In that case, the channel data between timestamps t_1 and t_2 (in seconds) is returned. The `floor` command is used for rounding the corresponding offsets during conversion to sample units.

Example

Assuming that the directory `sounds` contains `example.wav`, a `wav` file with three seconds of stereo sound, input:

```
> snd:=readwav("/path/to/sounds/example.wav"):;
  L:=channel_data(snd,left,range=1.2..1.5)
```

Output: A list `L` resp. `R` containing data between 1.2 and 1.5 seconds in the left resp. right channel of the original file.

```
> L:=snd[left,27400,100]
```

Output: A list of 100 samples from the left channel, starting from the sample with index 27400.

```
> L:=snd[left,1.2..1.5]
```

Output: A list of samples from the left channel falling between 1.2 and 1.5 seconds.

28.2.9 Setting samples in audio clips

The `set_channel_data` command sets the audio clip samples to desired values.

- `set_channel_data` takes two mandatory arguments and two optional arguments:
  - A, an audio clip.
  - *data*, a matrix or vector containing sample data.
  - Optionally, *offset*, a position in A at which *data* should be written (by default, $\mathit{offset} = 0$).
  - Optionally, *chn*, a channel index or specification (`left` or `right`) (by default, the same data is written to all channels or *data* must be a matrix with one row for each channel).
- `set_channel_data(`A, *data* $\langle$, *offset*, *chn* $\rangle$`)` returns a modified copy of A in which *data* is written to A at the desired *offset* and channel *chn*.
- If *data* is a matrix, then it must have n rows where n is the number of channels in A. In this case, the fourth argument should be dropped.
- If *data* is a vector, then it is written to the specified channel or to all channels if *chn* is not given.

Example

In the following example we create an audio clip containing the sine wave and add random noise to its portion.

```
> snd:=createwav(sin(2*pi*440*soundsec(4)))
```

a sound clip with 176400 samples at 44100 Hz (16 bit, mono

We extract the data after the first and before the third second of the audio.

```
> data:=channel_data(snd,0,range=1.0..3.0):;
```

Now we create the noise which fades in and out.

```
> noise:=hamming_window(ranv(length(data),normald(0,0.1))):;
```

Finally we replace the portion of the original clip with the mix of the sine wave and noise.

```
> snd1:=set_channel_data(snd,noise+data,44100,0)
```

a sound clip with 176400 samples at 44100 Hz (16 bit, mono)

28.2.10 Trimming silence

You can find leading and/or trailing silence and remove it by using the `trim` command (see Section 5.2.5, p. 57, Section 21.2.2, p. 569 and Section 28.1.9, p. 820 for other usages of `trim`).

- `trim` takes one mandatory argument and up to four optional arguments:
  - A, an audio clip object.
  - Optionally, *threshold*, a negative real number specifying the silence threshold in dB (by default, $threshold = -30$).
  - Optionally, *mavglen*, a nonnegative real number specifying the time interval (in seconds) for a moving-average filter (by default, $mavglen = 0.001$).
  - Optionally, either `left` or `right`, the symbol specifiying one-sided trimming.
  - Optionally, `index`, the symbol.
- `trim(`A ⟨, *threshold* ⟨, *mavglen* ⟩⟩ ⟨, `left`|`right` ⟩ ⟨, `index` ⟩`)` finds the start l of the first nonsilent portion of A and the start u of the trailing silence (in samples). Audio is considered silent at offset k if the instantaneous energy at k is below *threshold* decibels. The energy vector is smoothed out by applying a moving-average filter with length $n = \lfloor$`samplerate(`A`)` $\cdot\, mavglen + 1/2 \rfloor$ (i.e. the length in samples corresponding to *mavglen* seconds). To disable the filtering, set *mavglen* to 0.

  If `index` is given, then the return value is either l if `left` is given, u if `right` is given, or $(l, u - l)$ otherwise (the last sequence contains the start and length of the nonsilent part, in samples). If `index` is omitted, then the return value is an audio clip obtained from A by removing either the leading silence if `left` is given, trailing silence if `right` is given, or both leading and trailing silence otherwise.

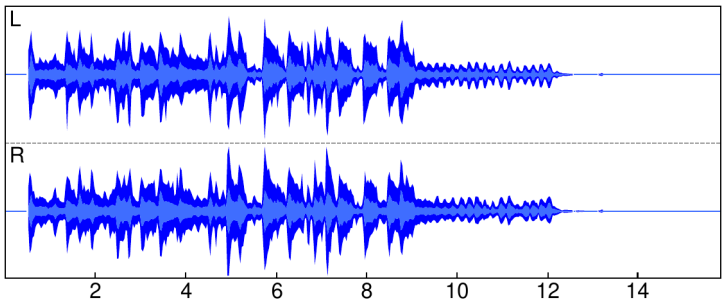
Example

With the file `gs-16b-2c-44100hz.wav` downloaded from here (an excerpt from a piece of band music):

```
> clip1:=readwav("/home/luka/Downloads/gs-16b-2c-44100hz.wav")
```

a sound clip with 698194 samples at 44100 Hz (16 bit, stereo)

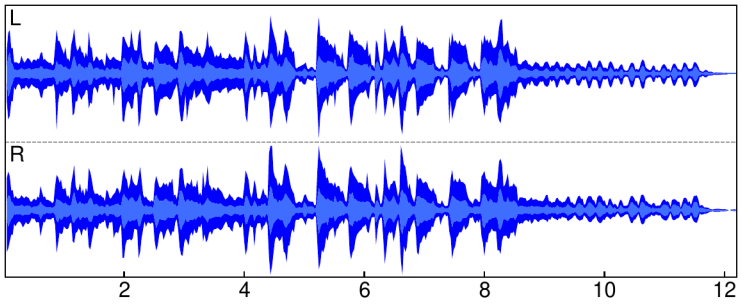
```
> plotwav(clip1)
```




```
> clip2:=trim(clip1)
```

a sound clip with 538796 samples at 44100 Hz (16 bit, stereo)

```
> plotwav(clip2)
```



28.2.11 Changing the sampling rate of an audio clip

The `resample` command resamples a clip to a desired rate.

- `resample` takes one mandatory argument and two optional arguments:
  - A, an audio clip.
  - Optionally, r, the target sample rate in Hz (by default $r = 44100$).
  - Optionally, n, an integer specifying the quality level, from 0 (poor) to 4 (best) (by default, $n = 2$).
- `resample(`$A\,\langle, r, n\rangle$`)` returns the audio clip A resampled to rate r.
- Xcas does resampling by using the `libsamplerate` library. For more information see the library documentation.

Example

Assuming that the directory `sounds` contains `example.wav`, a `wav` file with a sample rate of 44100:

```
> clip:=readwav("/path/to/sounds/example.wav"):; samplerate(clip)
```

44100

```
> res:=resample(clip,48000):; samplerate(res)
```

48000

28.2.12 Visualizing waveforms

The `plotwav` command displays the waveform of an audio clip.

- `plotwav` takes one mandatory argument and a sequence of optional arguments:
  - A, an audio clip.
  - Optionally, *opts*, a sequence of options each of which is one of:
    - `tstep=`*dt*, where *dt* is the size (in seconds) of intervals at which the nodes of the polygonal waveform representation will be computed (by default, *dt* = `duration(A)`/500).
    - *range*, which may be one of:
      - `range=[`*offset*,*length*`]` for integers *offset* and *length* (in samples).
      - `range=`a`..`b for floating point numbers a and b (in seconds).
    - *color*, which may be one of:
      - `color=[`*outercolor*, *innercolor*`]`, where *outercolor* and *innercolor* are Xcas colors (by default, *outercolor* = 216 and *innercolor* = 218).

- `color=`*col*, where *col* is a single color specifying the value for both *outercolor* and *innercolor*.

- `plotwav(`*A* ⟨, *opts*⟩`)` displays the waveform *A*, in its entirety or over the optional specified range. If more than one channel is present, they are drawn individually and stacked upon one another from top to bottom. If *A* is a stereo clip, then the upper waveform belongs to the left channel and the lower waveform to the right channel. The *t*-axis unit is equal to one second.

- `plotwav` computes the maximum amplitude and the mean amplitude for positive and negative signal values in intervals of size *dt*. The obtained points are connected into polygons which are drawn on screen. The maximum amplitude is colored with *outercolor* and the mean amplitude with *innercolor*.

Examples

With the file `gs-16b-2c-44100hz.wav` downloaded from here (an excerpt from a piece of band music):

```
> clip1:=readwav("/home/luka/Downloads/gs-16b-2c-44100hz.wav")
```

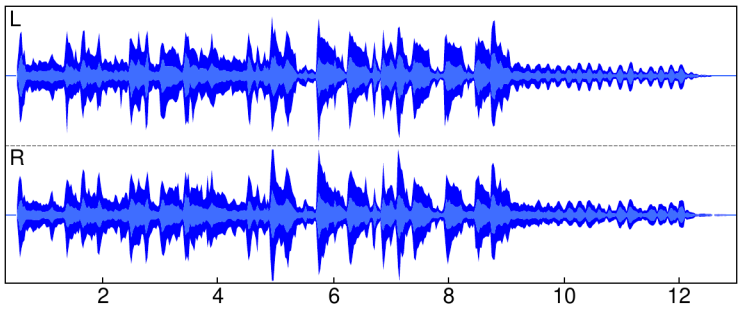
a sound clip with 698194 samples at 44100 Hz (16 bit, stereo)

```
> duration(clip1)
```

15.8320634921

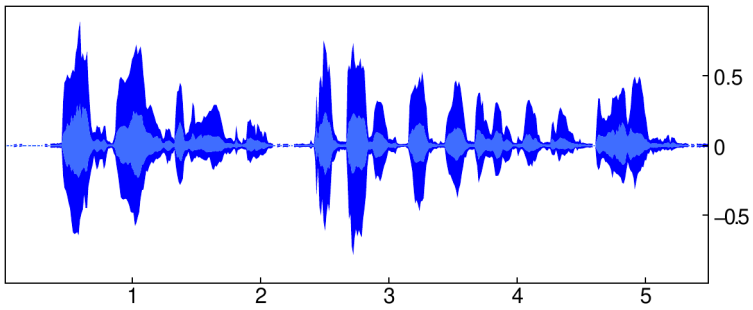
`plotwav` draws the waveform of each channel separately by going from top to bottom:

```
> plotwav(clip1,range=0.5..12.5)
```



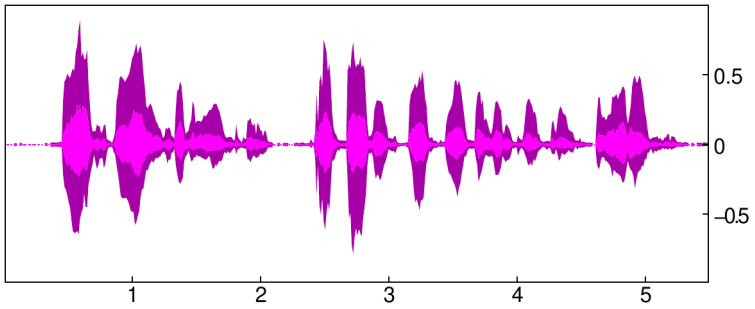
With the file `gettysburg10.wav` downloaded from here (a 10 seconds speech):

```
> clip2:=normalize(readwav("/home/luka/Downloads/gettysburg10.wav"),-1):;
  plotwav(clip2,range=0.0..5.5)
```



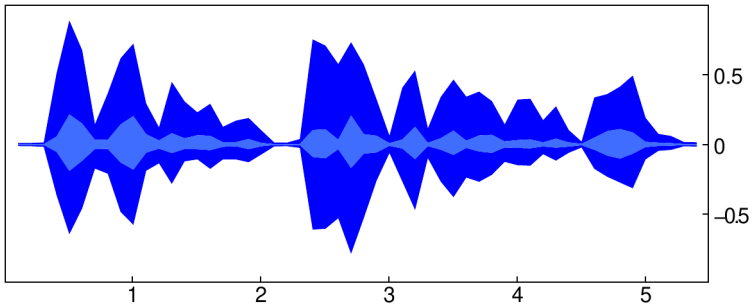
You can choose different colors for the waveform, for example:

```
> plotwav(clip2,range=0.0..5.5,color=[purple,magenta])
```



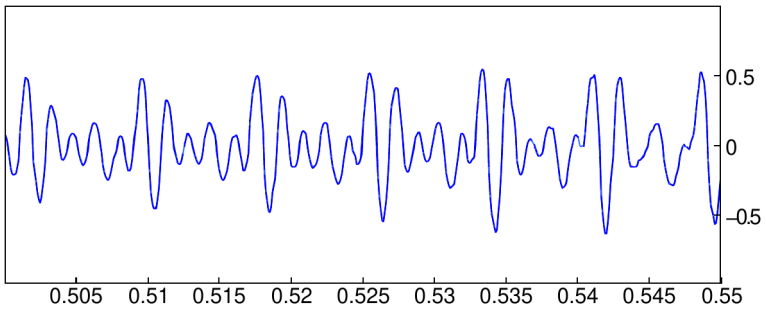
By setting the `tstep` option you can control the "resolution" at which the waveform is displayed. For example:

```
> plotwav(clip2,range=0.0..5.5,tstep=0.1)
```



To "zoom-in" the waveform, use the `range` option rather than the mouse wheel:

```
> plotwav(clip2,range=0.5..0.55)
```



28.2.13 Visualizing power spectra

The `plotspectrum` command displays the power spectrum of an audio clip.

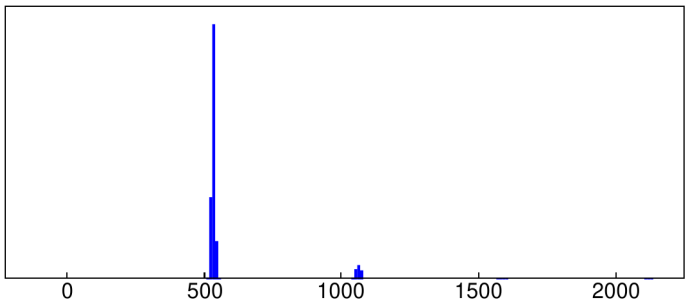
- `plotspectrum` takes one mandatory argument and a sequence of optional arguments:
  - *A*, an audio clip.
  - Optionally, *opts*, a sequence of options each of which is one of:
    - *span*, which is either a list `[`*fmin*, *fmax*`]` or `frequencies=`*fmin*`..`*fmax*, specifying the frequency band in Hertz (by default, *fmin* = 0 and *fmax* = `samplerate(`*A*`)/2`).
    - `bins=`*n*, where $n > 2$ is a positive integer which must be a power of two and specifies the number of bins for the frequency range $[0, $`samplerate(`*A*`)/2`$]$ (by default, $n = 512$).
    - `color=`*disp*, where *disp* is a sum of plot attributes (e.g. an XCAS color). By default, the spectrum is drawn in blue.
- `plotspectrum(`*A* $\langle$`,` *opts* $\rangle$`)` displays the power spectrum of *A* in the frequency range [*fmin*, *fmax*].

- If the audio clip has more than one channel, the channels are mixed down into a single channel before computing the spectrum.

Examples

With the file `flute-C5.wav` donwloaded from here:

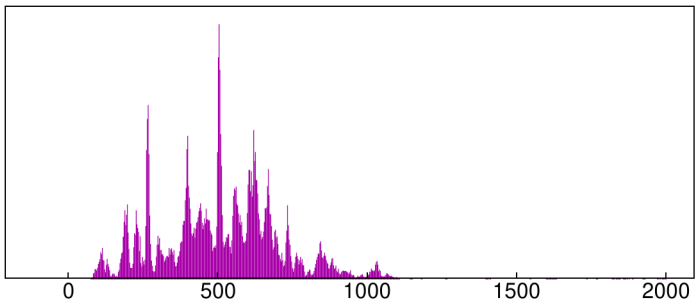
```
> flute:=readwav("/home/luka/Downloads/flute-C5.wav"):;
  plotspectrum(flute)
```



The audio file `flute-C5` consists of a single C5 tone played by a flute. In the graph above, the fundamental of that tone is clearly visible at the corresponding frequency of about 523 Hz. You can also see its first overtone at about 1046 Hz (the next two overtones are barely visible).

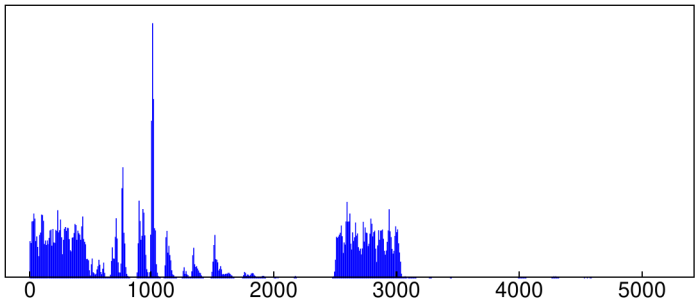
With the file `gettysburg10.wav` downloaded from here:

```
> clip:=readwav("/home/luka/Downloads/gettysburg10.wav"):;
  plotspectrum(clip,[0,2000],bins=4096,color=purple)
```



With the file `sf1_n0H.wav` downloaded from here:

```
> clip:=readwav("/home/luka/Downloads/sf1_n0H.wav"):;
  plotspectrum(clip,frequencies=0..5000,bins=1024)
```



In the above graph you can see the two components of the stationary noise contained in ranges 0-440 Hz and 2500-3000 Hz. The rest of the spectrum belongs to the female speaker. Use the `playsnd` command to hear the audio (see Section 28.2.14, p. 833).

28.2.14 Listening to a digital sound

The `playsnd` command plays digitized sound data.

- `playsnd` takes one mandatory argument and up to three optional arguments:
  - A, an audio clip.
  - Optionally, *offset*, *len*, a sequence of two integers, or $a..b$, an interval of real numbers, determining the portion of the clip to be played (by default, the entire clip is played).
  - Optionally, r, a positive integer (by default, $r = 1$).
- `playsnd(`A`)` plays the given (portion of) audio clip over the computer speakers and repeats it r times (setting r to a higher value loops the sound r times).
- The portion to be played is either the block of *len* samples starting at *offset* or the sound between a and b seconds.
- To stop the playback, press the STOP button.

28.2.15 Normalizing audio

The `normalize` command scales the amplitude of an audio clip.

- `normalize` takes two mandatory arguments:
  - A, an audio clip.
  - *level*, a negative real number specifying the maximum audio level in decibels (setting a positive value would result in clipping, i.e. audio distortion).
- `normalize` returns the copy of A normalized to the desired *level*.

Example

To normalize a sinusoidal sound to −6 dB:

```
> t:=soundsec(3):;
  L,R:=sin(2*pi*440*t),sin(2*pi*445*t):;
  s:=normalize(createwav([L,R]),-6):;
```

Indeed, since `about` computes the level of audio:

```
> about(s)
```

```
Bit depth: 16
Channels: 2 (stereo)
Sample rate: 44100
Length: 132300 samples
Duration: 3 seconds
Level: -6.00048 dB
```

28.2.16 Loudness estimation

The `rms` command computes the RMS (root mean square) of the audio data (see Section 21.2.1, p. 569 for other uses of `rms`).

- `rms` command takes one mandatory argument and two optional arguments:
  - A, an audio clip.
  - Optionally, *offset*, a nonnegative integer (by default, *offset* = 0).

  - Optionally, *len*, a positive integer (by default, $len = L - offset$, where L is the length of A).

- `rms(`A ⟨, *offset*, *len*⟩`)` computes RMS for each channel (optionally restricted to the portion of length *len* starting at *offset*) and returns the result as $\sqrt{\sum_c \mathrm{RMS}_c^2}$, where RMS_c is the RMS of channel c.
- `rms` operates with sample values mapped to $[-1, 1]$, hence the result is in $[0, 1]$. To convert the result to decibels, use the formula $\mathrm{RMS}_{\mathrm{dB}} = 20 \log_{10}(\mathrm{RMS})$.
- RMS is a measure of the average loudness of the audio.

Example

With the file `CantinaBand3.wav` downloaded from here:

```
> clip:=normalize(readwav("/home/luka/Downloads/CantinaBand3.wav"),0)
```

a sound clip with 66150 samples at 22050 Hz (16 bit, mono)

```
> r:=rms(clip)
```

0.152788493422

Average loudness in dB:

```
> 20*log10(r)
```

−16.3181870293

28.2.17 Joining audio clips together

The `splice` command joins two audio clips together with an overlap and optionally a crossfade.

- `splice` takes two mandatory arguments and one or two optional arguments:
  - A, an audio clip.
  - B, an audio clip with the same number of channels, bit depth and sample rate as A.
  - Optionally, *len*, a positive integer defining the overlap size (by default, *len* is the minimum of $0.5 \cdot s$, $L_A/2$, and $L_B/2$, where L_A and L_B are the lengths of A and B in samples and s is the sample rate).
  - Optionally, p, a real number defining the crossfade curve (by default, p is computed automatically, see below).
- `splice(`A, B ⟨, *len*, p⟩`)` returns the audio clip obtained by splicing A and B with an overlap of *len* samples and a crossfade which is computed using the function $f(x) = x^p$, $x \in [0, 1]$, for $p > 0$.
- For positively correlated audio data on the overlap, it is best to set $p = 1.0$ (constant gain crossfade), while for zero-correlated data $p = 0.5$ (constant power crossfade) works better. If p is not specified, it is computed automatically in $[0.5, 1.0]$ such that $p - 0.5$ is proportional to the absolute value of the cosine distance between the overlapping data. Setting $p = 0$ disables crossfading and thus the overlapped data is simply added together.

Example

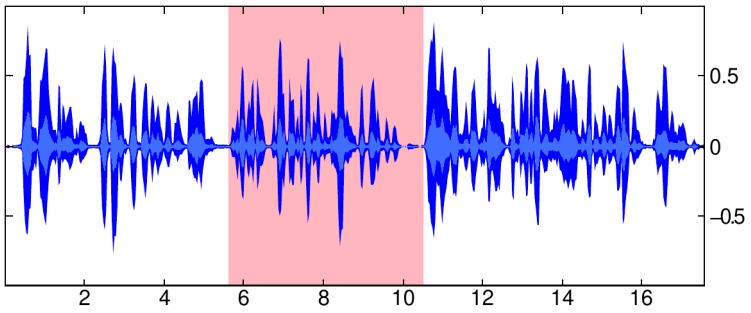
With the audio file `gettysburg.wav` downloaded from here:

```
> clip:=normalize(readwav("/home/luka/Downloads/gettysburg.wav"),-1)
```

a sound clip with 387574 samples at 22050 Hz (16 bit, mono)

To remove the portion of the speech between 5.6 and 10.5 seconds, splice the portions on the left and right by using an overlap of size 0.25 seconds. To visualize the audio and the selected portion, you can enter:

```
> t1:=5.6:; t2:=10.5:; d:=duration(clip):; cfl:=0.25:;
  rectangle(t1-i,t2-i,2/(t2-t1),display=pink+filled); plotwav(clip);
```



Extract the portions from 0 to t_1 and from t_2 to the end and splice them with an automatic crossfade:

```
> snd:=splice(clip(0.0..t1),clip(t2..d),cfl)
```

a sound clip with 274016 samples at 22050 Hz (16 bit, mono)

Use `playsnd` (see Section 28.2.14, p. 833) to hear the result.

28.2.18 Mixing audio clips together

The `mixdown` command is used for mixing several audio clips down together into one clip.

- `mixdown` takes an even number of arguments. Each pair of consecutive arguments is a sequence consisting of the following:
  - A, a stereo or mono audio clip.
  - *parm*, which is either a nonnegative real number or a list consisting of at most five elements. In the first case, *parm* is the offset at which A is mixed in, and in the second case *parm* consists of zero to five elements in the following order:
    - *offset*, a nonnegative real number representing the clip offset in seconds (by default 0).
    - *gain*, a real number representing the clip gain in dB (by default 0).
    - *pan*, a real number in $[-1, 1]$ (by default 0).
    - *fadein*, a nonnegative real number representing the fade-in length in seconds (by default 0).
    - *fadeout*, a nonnegative real number representing the fade-out length in seconds (by default 0).
- `mixdown(`A, *parm*$\langle, \ldots \rangle$`)` returns an audio clip which is the mix of the given clips. The audio is not automatically normalized, so calling `normalize` (see Section 28.2.15, p. 833) may be required afterwards. If clipping occurs while mixing, a warning is printed in the message area.

- All audio clips involved in the mix must have the same sample rate, but may have different bit depths. The largest bit depth is selected for the resulting audio. If at least one of the clips is a stereo (two-channel) sound, then the entire mix is a stereo sound as well, otherwise it is a mono (single-channel) sound.
- Each of the *offset*, *fadein* and *fadeout* parameters must be entered either as a nonnegative integer or as a floating-point number. In the first case the value is assumed to be in sample units, while in the second case it is assumed to be in seconds.
- The *gain* parameter is entered in decibels (dB) with negative values resulting in a softer mix, and positive values resulting in a louder mix.
- If *fadein* and/or *fadeout* parameters are larger than zero, then fade-in and/or fade-out will be applied to the respective clip while mixing. The fade function is $f(x) = \cos(\pi(1-x)/2)^2$ for $0 \le x \le 1$.
- In case of clipping, try calling `mixdown` again with lower gain values.

Example

In the following example we create a music track with six individual tones. First we write a function for creating tones with vibrato.

```
vibrato:=proc(f,t)
  local f1,f2,dl,dr,s;
  purge(s);
  f1:=rand(0.5,2.0);
  f2:=rand(0.5,2.0);
  dl:=[];
  dr:=[];
  for s in t do
    dl.append(sin(2*pi*(f+0.2*sin(2*pi*f1*s))*s));
    dr.append(sin(2*pi*(f+5+0.2*sin(2*pi*f1*s))*s));
  od;
  return createwav([dl,dr]);
end:;
```

Now create several tones of different lengths...

```
> s:=2.0^(1/12):;
  clip1:=vibrato(880,soundsec(7.5)):;     clip2:=vibrato(880*s,soundsec(5)):;
  clip3:=vibrato(880/s^2,soundsec(2.5)):; clip4:=vibrato(880/s^3,soundsec(6)):;
  clip5:=vibrato(880/s^11,soundsec(6)):;  clip6:=vibrato(880/(2*s^7),soundsec(6)):;
```

... and mix them down at different positions.

```
> g:=-6:;
  mix:=mixdown(clip1,[0,g,-0.3,2.0,0.2],    clip2,[5.0,g,0.7,0.5,2.0],
               clip3,[7.3,g,-0.3,0.2,0.2],  clip4,[9.6,g,-0.3,0.2,3.0],
               clip5,[9.8,-6.0,0.0,0.5,4.0],clip6,[10.0,-8.0,0.15,0.5,2.0])
```

a sound clip with 705600 samples at 44100 Hz (16 bit, stereo)

Finally, play the music:

```
> playsnd(normalize(mix,-1))
```

Output: a 16 seconds of synthesized music.

Index