Graphics in the wxMaxima GUI
(Document version 0.78)

by David E. Brown, based heavily on wxMaxima help files

March 19, 2012

THIS DOCUMENT IS UNDER CONSTRUCTION

However, the content is complete enough to be useful.

Contents

Inquiries should be directed to me at `brownd@byui.edu` or at:

David E. Brown
RCKS 232 H
Brigham Young University–Idaho
525 S. Center Street
Rexburg, ID 83440
U.S.A.

This document was prepared using the LaTeXdocument preparation (electronic typesetting) system, with liberal help from wxMaxima itself.

[1] I regret being unable to indicate exactly which parts I have written and which parts I have not. Either you or I should check with a lawyer before drawing any conclusions, but I'd like to suggest the possibility that it might be better if you saved or copied some other documentation, until I can get this resolved.

Chapter 1

Introduction

1.1 About this document

This document aims to be a fairly comprehensive reference manual and a sort of "field guide" to the graphics features of wxMaxima. It is neither a tutorial nor a "how-to," though it would hopefully be useful as a reference for writing one such a document. It is based on Chapter 48 of the Maxima manual for wxMaxima version 5.20.1 (updated to reflect the content of Chapter 51 of the Maxima manual for wxMaxima version 5.24.0) and Chapters 45, 48, 52, 56, 57, 59, 62, and 75 of the Maxima manual for wxMaxima version 5.24.0. Indeed, it contains many, many uncited direct quotations from these chapters.

I have done some experimenting to identify behaviors and bounds on behaviors not explicitly described in the official documentation. Someday, I will actually contact those who have written the code or make time to sift through it myself to make sure I've got my stories straight. **Disclaimer: Until this is done, you may find that some things don't work quite the way I say they do.** Please note that some discrepancies between the results you get and the results I claim you will get are bound to be due to the characteristics of different operating systems or the use of different flags at the time of compilation of Maxima or wxMaxima. As usual, the best method of getting as nuisance-free a copy of the software as you can is to compile it yourself, optimizing the compilation for your system. I refer you to the official documentation for instructions on compiling wxMaxima.[1] (Try starting with the README file.)

The lines along which this document is organized are these: The chapters are in alphabetical order, by the names of the packages or commands treated therein. If there is enough material on a given package or command, its chapter is divided into sections, typically starting with a bit of introduction to the package or command. If a command is being treated, a fairly careful description of it is next. In either case, the next section is about relevant graphics objects, if any, followed by a section on relevant functions, if any. Graphics and other options come next, with relevant file handling functions and any other topics ending the chapter.

Within any given section or subsection, the order in which the entries for functions, graphics objects, and so on appear is alphabetical order. Each entry begins with the name of the item to be described, in the context of appropriate syntax or syntaxes, along with an indication of the nature of the item: system variable, global graphics option, etc. I will make some attempt to render the notation consistent in these entry headers, at some point. For now, there are bigger fish to fry.

After the entry header come the following, in this order, as applicable: default values, allowable values, hyperlinks to applicable options, hyperlinks to any functions or objects to which the item applies, and hyperlinks to the entries of related items. The things in each of these lists are again

[1] Speaking of compiling, all the versions of wxMaxima I've used so far have made use of only one processor core at a time. I would be interested to hear whether this limitation has been overcome.

listed alphabetically. I have done my best to make all this as comprehensive as I can in the time I've had, based on the available documentation. However, the available documentation is not perfectly comprehensive. If you are or become aware of options, functions, or what have you that are relevant to a given entry but do not appear here, please contact me.

Examples are scattered throughout the document. Each example is part of the entry for the function, graphics object, graphics option, or variable in question. Many of the examples use options or objects or what have you, other than the one being illustrated. Sometimes this is unavoidable and sometimes it's to provide variety for the purpose of comparing and contrasting. Of course, sometimes, it's just to have nicer-looking graphics. It will take some effort and patience to work with such examples, but they will point you to possibilities you might not have considered otherwise. This will help you use the software more to its fullest and have a more satisfying experience with it.

This document is fully searchable in any pdf viewer that has a search function, and is generously cross-referenced with active hyperlinks to help you navigate more easily. It also has features designed to make it readable.

(Confession: The above is only inconsistently true at this point. This document is still very much under construction. Even the parts that are not labeled as being under construction still await various kinds of improvements. Feedback is requested via email to me, especially feedback about errors and inconsistencies.)

1.2 About wxMaxima and graphics (under construction)

wxMaxima is a free, open-source graphical user interface, written using wx widgets, for the free, open-source computer algebra system (CAS) Maxima.[2] Neither Maxima nor wxMaxima plots anything. They delegate plotting tasks to gnuplot, gnuplot_pipes, or Xmaxima, in some cases making use of Tcl and other programs. Maxima preprocesses some kinds of plots before handing them over to the plotting software. For the kinds of plots that Maxima can preprocess, you can use the `plot2d` or `plot3d` or related commands. Most of the remaining types of plots are preprocessed (or even pre-prepreocessed) by third-party packages such as `draw`, `graphs`, and `plotdf`. In turn, these packages typically call `draw` or one of the `plot` variants.

It appears that `draw` and the `plot` commands pass some graphics options directly to gnuplot, Xmaxima, or whichever plotting program it calls. Other graphics options are parsed and processed and then passed to the plotting program. I believe most CAS users will feel more comfortable with the command syntax wxMaxima uses than that which gnuplot uses. In any case, familiarity with the documentation for gnuplot, gnuplot_pipes, Tcl, and Xmaxima can add to your understanding of wxMaxima's graphics capabilities.

1.3 System requirements (under construction)

Table 1.1 indicates the additional software (beyond wxMaxima) that must be on your system. Note that if you want to use graphviz with the `graphs` package, it must be installed separately.

[2] After two decades of teaching college mathematics and statistics, I have had it with commercial software written for these disciplines. I won't give my reasons here, but I have broken with my colleagues and now use wxMaxima in mathematics courses in which I feel the use of a CAS is warranted. wxMaxima is not as elegant as many of the commercial CAS's, and sometimes I wish it had more functionality. However, in courses from Calculus I and Linear Algebra, through Partial Differential Equations and Real and Complex Analysis, to Abstract Algebra, wxMaxima meets almost all my needs, and does some things better than the commercial CAS's. Needless to say, my students also appreciate not having to pay for a CAS, though some of them buy a license for whatever CAS their science or engineering professors use. I allow my students to use any CAS they like, but I use wxMaxima exclusively in the classroom, except on those rare occasions when it cannot yet supply the need.

Table 1.1: Software dependencies for various wxMaxima graphics commands and packages.

To use these with wxMaxima:	You must also have these installed:
`bode`	N/A
`descriptive`	N/A
`draw` and its variants	gnuplot 4.2 or later
`drawdf`	gnuplot 4.2 or later
`dynamics`	N/A
`finance`	N/A
`fractal`	N/A
`graph`	N/A, but can use graphviz programs
`implicit_plot`	N/A
`picture`	N/A
`plot` and its variants	gnuplot, gnuplot_pipes, mgnuplot, or Xmaxima
`plotdf`	Xmaxima
`worldmap`	N/A

1.4 Plotting formats for wxMaxima

Maxima currently uses gnuplot and Xmaxima for plotting. (Xmaxima supercedes the older Openmathprogram.) There are various different formats for those programs. Package `draw` uses only gnuplot. If you're using `plot2d` or `plot3d`, the default is gnuplot_pipes, except in Windows™systems, for which the default is gnuplot (indeed, gnuplot_pipes is inherehntly incompatible with Windows™systems). The format can be changed by using the option `plot_format`.

The plotting formats are briefly described below.

1.4.1 gnuplot

gnuplot is a program external to wxMaxima, and must be installed separately on your system to be used. When wxMaxima plots anything using gnuplot, it saves all plotting commands and data in the file maxout.gnuplot. gnuplot is the default plotting program for installations of wxMaxima on Windows™systems.

1.4.2 gnuplot_pipes

This plotting program, the default for wxMaxima when installed on non-Windows™ systems, is not available for Windows™ platforms. It is similar to the gnuplot format except that the commands are sent to gnuplot through a pipe, while the data are saved into the file maxout.gnuplot_pipes. A single gnuplot process is kept open and subsequent plot commands will be sent to the same process, replacing previous plots, unless the gnuplot pipe is closed (for example, with the function `gnuplot_close()`). When this format is used, the function `gnuplot_replot` can be used to modify a plot that has already displayed on the screen (see `gnuplot_replot`).

This format should only be used to plot to the screen; for plotting to files it is better to use the gnuplot format.

1.4.3 mgnuplot

mgnuplot is a Tk-based wrapper around gnuplot. It is included in the Maxima distribution. mgnuplot offers a rudimentary GUI for gnuplot, but has fewer overall features than the plain gnuplot interface. mgnuplot requires an external gnuplot installation and, in Unix systems, the Tcl/Tk system.

1.4.4 Xmaxima

Xmaxima is a Tcl/Tk graphical interface for Maxima that can also be used to display plots created when Maxima is run from the console or from other graphical interfaces. To use this format, the Xmaxima program, which is distributed together with Maxima, should be installed. If Maxima is being run from Xmaxima itself, this format will make the plot functions send the data and commands through the same socket used for the communication between Maxima and Xmaxima. When used from the console or from other interface, the commands and data will be saved in the file `maxout.Xmaxima`, and the Xmaxima program will be launched with the name of the location of that file as argument.

In previous versions this format used to be called Openmath. For backward compatibility, "Openmath" will still be accepted as a synonym for "Xmaxima."

1.5 Comparison of wxMaxima's plotting routines (to appear)

Chapter 2

The bode package

Bode (boh-dee) plots give information about the frequency and phase responses of such (time-invariant) systems as filters, as used in signal processing. The plots utilize the transfer function of the filter, which is the ratio of the Laplace transform of the output function of the system to that of the input function. As such, it represents the effect of the system on the input (note, however, that it exists in the so-called "s-domain"). The bode gain plot describes the system's frequency response, and the bode phase plot describes how the system shifts the phase of the input.

`bode_gain(H, range, ...plot options)` **Function**

See also: bode_phase

`bode_gain` draws Bode gain plots. You must load the `bode` package before using this function.

Example: Various bode gain plots.

```
(%i1) load("bode")$
      H1(s) := 100 * (1 + s) / ((s + 10) * (s + 100))$
      bode_gain(H1(s), [w, 1/1000, 1000])$
      H2(s) := 1 / (1 + s/omega0)$
      bode_gain(H2(s), [w, 1/1000, 1000]), omega0 = 10$
      H3(s) := 1 / (1 + s/omega0)^2$
      bode_gain(H3(s), [w, 1/1000, 1000]), omega0 = 10$
      H4(s) := 1 + s/omega0$
      bode_gain(H4(s), [w, 1/1000, 1000]), omega0 = 10$
      H5(s) := 1/s$
      bode_gain(H5(s), [w, 1/1000, 1000])$
      H6(s) := 1/((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$
      bode_gain(H6(s), [w, 1/1000, 1000]), omega0 = 10, zeta = 1/10$
      H7(s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$
      bode_gain(H7(s), [w, 1/1000, 1000]), omega0 = 10, zeta = 1/10$
      H8(s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$
      bode_gain(H8(s), [w, 1/1000, 1000])$
```

`bode_phase(H, range, ...plot options)` **Function**

See also: bode_gain, bode_phase_unwrap

`bode_phase` draws Bode phase plots. You must load the `bode` package before using this function.

Example: Various bode phase plots.

```
(%i1) load("bode")$
      H1(s) := 100 * (1 + s) / ((s + 10) * (s + 100))$
      bode_phase(H1(s), [w, 1/1000, 1000])$
      H2(s) := 1 / (1 + s/omega0)$
      bode_phase(H2(s), [w, 1/1000, 1000]), omega0 = 10$
      H3(s) := 1 / (1 + s/omega0)^2$
      bode_phase(H3(s), [w, 1/1000, 1000]), omega0 = 10$
      H4(s) := 1 + s/omega0$
      bode_phase(H4(s), [w, 1/1000, 1000]), omega0 = 10$
      H5(s) := 1 / s$
      bode_phase(H5(s), [w, 1/1000, 1000])$
      H6(s) := 1 / ((s/omega0)^2 + 2 * zeta * (s/omega0) + 1)$
      bode_phase(H6(s), [w, 1/1000, 1000]), omega0 = 10, zeta = 1/10$
      H7(s) := (s/omega0)^2 + 2 * zeta * (s/omega0) + 1$
      bode_phase(H7(s), [w, 1/1000, 1000]), omega0 = 10, zeta = 1/10$
```

bode_phase_unwrap **Graphics option**

Default value: `false`

Allowable values: `false`, `true`

See also: bode_phase

Example: The effect of the `bode_phase_unwrap` option.

```
(%i1) H8(s) := 0.5 / (0.0001 * s^3 + 0.002 * s^2 + 0.01 * s)$
      bode_phase(H8(s), [w, 1/1000, 1000])$
      block([bode_phase_unwrap : false],
        bode_phase(H8(s), [w, 1/1000, 1000]));
      block([bode_phase_unwrap : true],
        bode_phase(H8(s), [w, 1/1000, 1000]));
```

Chapter 3

The descriptive package

The descriptive package is a collection of functions for descriptive statistics. It includes functions for basic statistical graphics, such as histograms.

3.1 Functions for statistical graphs

histogram(list) **Function**
histogram(list, options,...) **Function**
histogram(one_column_matrix) **Function**
histogram(one_column_matrix, options,...) **Function**
histogram(one_row_matrix) **Function**
histogram(one_row_matrix, options,...) **Function**

Options: nclasses and all options used by draw

See also: continuous_freq, discrete_freq

This function plots an histogram from a continuous sample. Sample data must be stored in a list of numbers or a one dimensional matrix.

scatterplot(list) **Function**
scatterplot(list, options) **Function**
scatterplot(matrix) **Function**
scatterplot(matrix, options) **Function**

Options: nclasses and all options used by draw

See also: histogram, points

Plots scatter diagrams both for univariate (list) and multivariate (matrix) samples.

Example: Univariate scatter diagram from a simulated Gaussian sample.

```
(%i1) load(descriptive)$
      load(distrib)$
      scatterplot(random_normal(0,1, 200),
        xaxis      = true,
        point_size = 2)$
```

Example: Two-dimensional scatter plot.

```
(%i1) load(descriptive)$
      s2 : read_matrix(file_search("wind.data"))$
      scatterplot(submatrix(s2, 1,2,3),
```

```
  title      = "Data from stations #4 and #5",
  point_type = diamant,
  point_size = 2,
  color      = blue)$
```

Example: Three dimensional scatter plot.

```
(%i1) load(descriptive)$
      s2 : read_matrix(file_search("wind.data"))$
      scatterplot(submatrix(s2, 1,2))$
```

Example: Five dimensional scatter plot, with histograms having five classes each.

```
(%i1) load(descriptive)$
      s2 : read_matrix(file_search("wind.data"))$
      scatterplot(s2,
        nclasses     = 5,
        fill_color   = blue,
        fill_density = 0.3,
        xtics        = 5)$
```

barsplot(data1, data2, options) **Function**

Options: bars_colors, box_width, groups_gap, relative_frequencies, ordering, sample, and all those used by draw

See also: bars, histogram, piechart

Plots bars diagrams for discrete statistical variables, both for one or multiple samples. Data can be a list of outcomes representing one sample, or a matrix of m rows and n columns, representing n samples of size m each.

Example: Univariate sample in matrix form (absolute frequencies.)

```
(%i1) load(descriptive)$
      m : read_matrix(file_search("biomed.data"))$
      barsplot(col(m,2),
        title        = "Ages",
        xlabel       = "years",
        box_width    = 1/2,
        fill_density = 3/4)$
```

Example: Two samples of different sizes, with relative frequencies and user declared colors.

```
(%i1) load(descriptive)$
      l1:makelist(random(10), k,1,50)$
      l2:makelist(random(10), k,1,100)$
      barsplot(l1,l2,
        box_width            = 1,
        fill_density         = 1,
        bars_colors          = [black, grey],
        relative_frequencies = true,
        sample_keys          = ["A", "B"])$
```

Example: Four non-numeric samples of equal size.

```
(%i1) load(descriptive)$
      barsplot(makelist([Yes, No, Maybe][random(3)+1], k,1,50),
        makelist([Yes, No, Maybe][random(3)+1], k,1,50),
        makelist([Yes, No, Maybe][random(3)+1], k,1,50),
        makelist([Yes, No, Maybe][random(3)+1], k,1,50),
        title        = "Opinion poll with four groups",
        ylabel       = "# of individuals",
        groups_gap   = 3,
        fill_density = 0.5,
        ordering     = ordergreatp)$
```

piechart(list) **Function**
piechart(list, options) **Function**
piechart(one_column_matrix) **Function**
piechart(one_column_matrix, options) **Function**
piechart(one_row_matrix) **Function**
piechart(one_row_matrix, options) **Function**

Options: pie_center, pie_radius, and all those used by draw
See also: barsplot

Similar to barsplot, but plots sectors instead of rectangles.

Example:

```
(%i1) load(descriptive)$
      s1 : read_list(file_search("pidigits.data"))$
      piechart(s1,
        xrange      = [-1.1, 1.3],
        yrange      = [-1.1, 1.1],
        axis_top    = false,
        axis_right  = false,
        axis_left   = false,
        axis_bottom = false,
        xtics       = none,
        ytics       = none,
        title       = "Frequencies of the digits of pi")$
```

boxplot(data) **Function**
boxplot(data, options) **Function**

Options: box_width and all those used by draw
See also:

This function plots box-and-whisker diagrams. Argument data can be a list, which is not of great interest, since these diagrams are mainly used for comparing different samples, or a matrix, so it is possible to compare two or more components of a multivariate statistical variable. But it is also allowed data to be a list of samples with possible different sample sizes, in fact this is the only function in package descriptive that admits this type of data structure.

Example: Box-and-whisker diagram from a multivariate sample.

```
(%i1) load(descriptive)$
      s2 : read_matrix(file_search("wind.data"))$
      boxplot(s2,
        box_width  = 0.2,
```

```
  title      = "Windspeed in knots",
  xlabel     = "Stations",
  color      = red,
  line_width = 2)$
```

Example: Box-and-whisker diagram from three samples of different sizes.

```
(%i1) load(descriptive)$
      A :
        [[ 6,  4,  6,  2,  4,  8,  6,  4,  6,  4, 3, 2],
        [ 8, 10,  7,  9, 12,  8, 10],
        [16, 13, 17, 12, 11, 18, 13, 18, 14, 12]]$
      boxplot(A)$
```

3.2 Graphics options for statistical graphics

`bars_colors` **Graphics option**

Default value: `[]`

Applies to: barsplot

A list of colors for multiple samples. Defaults to the empty list []. When there are more samples than specified colors, the extra necesary colors are chosen at random. See color to learn more about them.

`box_width` **Graphics option**

Default value: `3/4`

Applies to: barsplot

Relative width of rectangles. This value must be in the range [0,1].

`groups_gap` **Graphics option**

Default value: `1`

Applies to: barsplot

A positive integer representing the gap between two consecutive groups of bars.

`nclasses` **Graphics option**

Default value: `10`

Applies to: `histogram`

See also: `bars, barsplot, discrete_freq and continuous_freq`

Number of classes of the histogram.

Example: A simple histogram with eight classes.

```
(%i1) load(descriptive)$
      s1 : read_list(file_search("pidigits.data"))$
      histogram(s1,
        nclasses     = 8,
        title        = "pi digits",
        xlabel       = "digits",
        ylabel       = "Absolute frequency",
        fill_color   = grey,
        fill_density = 0.6)$
```

`ordering` **Graphics option**

Default value: `orderlessp`
Applies to: barsplot
Allowable values: `ordergreatp, orderlessp`
Indicates how statistical outcomes should be ordered on the x-axis.

`pie_center` **Graphics option**
Default value: `[0,0]`
Location of the center of a pie chart.

`pie_radius` **Graphics option**
Default value: `1`
Radius of a pie chart.

`relative_frequencies`false **Applies to:** barsplot
If `false`, absolute frequencies are used; if `true`, tics on the y-axis are relative frequencies.

`sample_keys`[] **Applies to:** barsplot
A list with the strings to be used in the legend. If the list is not empty, the number of strings must equal the number of samples.

Chapter 4

The draw package

4.1 Introduction to the draw package

draw is a library of Lisp functions for creating graphics in Maxima, and therefore in wxMaxima. It is rich with features and very flexible.

Use of the draw package requires gnuplot 4.2 or higher, which may or may not be installed on your system. Also, you must load the draw package before using it in any give Maxima session.

Note: As of 31 December 2010, wxMaxima support for Macintosh™ operating systems is experimental. In particular, gnuplotcannot yet render objects created for terminal aquaterm in the Aqua graphics display program that Macintosh™ computers use.

draw's preprocessing involves creating a *scene* out of *graphics objects*. Examples of graphics objects include parametric plots, implicit plots, and explicit plots. Graphics objects are combined with *graphics options* to make a scene. One or more scenes can be plotted simultaneously in the same graphics window or in separate graphics windows, or one after the other in an animated gif file. draw also allows for saving plots in various graphic and other formats, such as jpg, png, eps, and pdf.

Some graphics options are *global*, meaning they affect every scene in the plot or every graphics object in the scene. Others are not. Non-global options must precede a given graphics object to have an effect on it. It does not matter where in the draw command global options appear. Their effect will be the same whether they come before or after the graphics object. I have found it useful to put all the global options for a given plot first, followed by the non-global options and the graphics objects they are to affect. This sort of organization can save you much trouble over time.

The draw command proper sends its preprocessed plot to gnuplot, which opens a separate window to display the plot. The variant wxdraw instructs gnuplotto display the plot within your wxMaxima file. You have to tell these commands pretty much everything there is to tell about the graphic you want to construct, including whether you want to plot in two dimensions or three (or both). If you use the draw2d or wxdraw2d command, draw will assume you want 2D graphics objects, and make use of default settings for 2D plots; likewise for the draw3d and wxdraw3d commands. This can save you some typing. Most of the examples in this document use the draw2d and draw3d variants. If you want to plot both 2D graphics objects and 3D graphics objects using one command, you'll have to use the draw variant.

Note: As of 7 November 2011, in Windows™platforms, the gnuplotwindow generated for a given plot must be closed before draw can plot any subsequent scene. This appears to be due to the lack of pipes.

Here is a link to more elaborate examples for this package than those given herein:[1]
http://www.telefonica.net/web2/biomates/maxima/gpdraw

4.2 The draw and wxdraw commands and their variants

draw(option(s), gr2d,...,gr3d,...) **Function**

Options: columns, data_file_name, delay, file_name, eps_height, eps_width, gnuplot_file_name, pdf_height, pdf_width, pic_height, pic_width, terminal

See also: draw2d, draw3d, gr2d, gr3d, wxdraw2d, wxdraw3d

Package draw plots a collection of scenes, which are constructed from graphics objects, graphics options, and other functions and variables by gr2d or gr3d. Either gr2d or gr3d or both may be used in any given call to draw. The scenes are plotted in a separate window than the one in which your wxMaxima session is running. By default, draw puts the scenes in one column. Functions draw2d and draw3d are shortcuts to be used when only one scene is required, in two or three dimensions, respectively.

Example: An ellipse and a triangle.

```
(%i1) load(draw)$
      scene1: gr2d(title = "Ellipse",
                   nticks   = 30,
                   parametric(2*cos(t),5*sin(t),t, 0,2*%pi))$
      scene2: gr2d(title = "Triangle",
                   polygon([4,5,7], [6,4,2]))$
      draw(scene1, scene2, columns = 2)$
```

Example: These two draw commands are equivalent.

```
(%i1) load(draw)$
      draw(gr3d(explicit(x^2+y^2, x,-1,1, y,-1,1)))$
      draw3d(explicit(x^2+y^2, x,-1,1, y,-1,1))$
```

Example: An animated gif, stored in a file called zzz. wxMaxima will not display this file. To see it, you'll have to open it in some other program.

```
(%i1) load(draw)$
      draw(delay  = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2, x,-1,1)),
        gr2d(explicit(x^3, x,-1,1)),
        gr2d(explicit(x^4, x,-1,1)))$
```

draw2d (option(s), graphic_object(s),...) **Function**

See also: draw, draw3d, gr2d, wxdraw, wxdraw2d, wxdraw3d

draw2d is a short cut for draw(gr2d(option(s),...,graphic_object(s),...)). It can be used to plot a single scene in 2D.

draw3d(option(s), graphic_object(s),...) **Function**

See also: draw, draw2d, gr3d, wxdraw, wxdraw2d, wxdraw3d

[1] This link worked as recently as 19 October 2011.

draw3d is a short cut for draw(gr3d(option(s),...,graphic_object(s),...)). It can be used to plot a single scene in 3D.

wxdraw(option(s), gr2d,...,gr3d,...) **Function**

Options: columns, data_file_name, delay, file_name, eps_height, eps_width, gnuplot_file_name, pdf_height, pdf_width, pic_height, pic_width, terminal

See also: draw, draw2d, draw3d, gr2d, gr3d, wxdraw2d, wxdraw3d

Function wxdraw behaves exactly like draw, except wxdraw places its plots in your wxMaxima session file.

wxdraw2d(option(s), graphic_object(s),...) **Function**

See also: draw, draw2d, draw3d, gr2d, wxdraw, wxdraw3d

wxdraw2d is a shortcut for wxdraw(gr2d(option(s),...,graphic_object(s),...)). It can be used to plot a scene in 2D.

wxdraw3d(option(s), graphic_object(s),...) **Function**

See also: draw, draw2d, draw3d, gr3d, wxdraw, wxdraw2d

wxdraw3d is a shortcut for wxdraw(gr3d(option(s),...,graphic_object(s),...)). It can be used to plot a scene in 3D.

4.3 Scenes

Scenes are the basic "things" that draw and its variants can plot. If you use the draw command or the wxdraw command, you will also have to give an explicit description of the scene(s) you want plotted. When you use the draw2d, draw3d, wxdraw2d, or wxdraw3d variants, draw will assume you want 2D or 3D scenes (as appropriate) and use the relevant defaults.

gr2d(graphics option(s),...,graphics object(s),...) **Scene constructor**

See also: axis_bottom, axis_left, axis_right, axis_top, eps_height, eps_width, file_name, grid, logx, logy, pic_height, pic_width, terminal, title, user_preamble, xaxis, xaxis_color, xaxis_type, xaxis_width, xlabel, xrange, xtics, xtics_axis, xtics_rotate, xy_file, yaxis, yaxis_color, yaxis_type, yaxis_width, ylabel, yrange, ytics, ytics_axis, ytics_rotate

gr2d builds a complete description of a 2D scene, for use by the draw command. Arguments to gr2d are graphics options and graphics objects, or lists of graphics options and graphics objects. The arguments are interpreted sequentially, in the sense that non-global graphics options affect all the graphics objects that follow them, until the options are changed. Global graphics options affect appearance of the entire scene.

The following graphics objects can be used in 2D scenes: bars, ellipse, explicit, geomap, image, implicit, label, parametric, points, polar, polygon, quadrilateral, rectangle, triangle, and vector.

gr3d(graphics option(s),...,graphics object(s),...) **Scene constructor**

See also: axis_3d, axis_bottom, axis_left, axis_right, axis_top, colorbox, contour, contour_levels, enhanced3d, eps_height, eps_width, file_name, grid, logx, logy, logz, palette, pic-height, pic_width, rot_horizontal, rot_vertical, surface_hide, terminal, title, user_preamble, xaxis, xaxis_color, xaxis_type, xaxis_width, xlabel, xrange, xtics, xtics_axis, xtics_rotate, xy_file, xu_grid, yaxis, yaxis_color, yaxis_type, yaxis_width, ylabel, yrange, ytics, ytics_axis, ytics_rotate, yv_grid, zaxis, zaxis_color, zaxis_type, zaxis_width, zlabel, zrange, ztics, ztics_axis, ztics_rotate

gr3d builds a complete description of a 3D scene, for use by the draw command. The arguments to gr3d are graphics options and graphics objects. The arguments are interpreted sequentially, in the sense that non-global graphics options affect all the graphics objects that follow them, until the options are changed. Global graphics options affect appearance of the entire scene.

The following graphics objects can be used in 3D scenes: cylindrical, elevation_grid, explicit, geomap, implicit, label, parametric, parametric_surface, points, quadrilateral, spherical, triangle, tube, vector.

4.4 Graphics objects

Graphics objects are the basic ingredients of scenes. Some, like explicit and parametric can be used to make either 2D or 3D plots. Others, such as polar, can only be used to make 2D plots. Still others, like parametric_surface, can only be used to make 3D plots.

4.4.1 Graphics objects used in both 2D and 3D plots

`explicit(fcn, var,minval,maxval)` **Graphics object**

Options: adapt_depth, color, fill_color, filled_func, key, line_type, line_width, nticks

See also: filled_function

`explicit(fcn, var1,minval1,maxval1, var2,minval2, maxval2)` **Graphics object**

Options: color, enhanced3d, key, line_type, line_width, xu_grid, yv_grid

explicit draws functions that are defined explicitly.

2D: explicit(fcn, var,minval,maxval) plots function fcn, which must be given explicitly in terms of variable var. (This means fcn has the form f(var), where f is a function of var alone.) explicit plots fcn for values of var ranging from minval to maxval.

Example: 2D explicit objects, with and without fill.

```
(%i1) load(draw)$
      draw2d(line_width = 3,
        explicit(x^2, x,-3,3))$
      draw2d(fill_color = red,
        filled_func     = true,
        explicit(x^2, x,-3,3))$
```

3D: explicit(fcn, var_1,minval_1,maxval_1, var_2,minval_2,maxval_2) plots function fcn, which must be given explicitly in terms of variables var_1 and var_2. explicit plots fcn for all combinations of values of var_1 and var2 ranging from minval_1 to maxval_1 and from minval_2 to maxval_2, respectively. The effect is to plot fcn over a rectangle.

Example: A 3D explicit object.

```
(%i1) load(draw)$
      draw3d(surface_hide = true,
        key               = "Plane",
        explicit(x+y, x,-5,5, y,-5,5),
        key               = "Gauss",
        color             = "\#a02c00",
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3))$
```

`implicit(exprssn, x,xmin,xmax, y,ymin,ymax)` **Graphics object**

Options: color, ip_grid, ip_grid_in, key, line_width, line_type

`implicit(fcn, x,xmin,xmax, y,ymin,ymax, z,zmin,zmax)` **Graphics object**

Options: color, key, line_type, line_width, x_voxel, y_voxel, z_voxel

Draws in 2D and 3D functions that are defined implicitly.

2D: `implicit(exprssn, x,xmin,xmax, y,ymin,ymax)` plots curve `exprssn`, defined implicitly in terms of variables `x` and `y`. `x` takes values from `xmin` to `xmax`, and variable `y` takes values from `ymin` to `ymax`. Note that `exprssn` can take one of two forms:

- $f(x, y)$, which `draw` assumes is equal to 0, and
- $f(x, y) = g(x, y)$.

Example: Using both forms of `expressn`.

```
(%i1) load(draw)$
      draw2d(title    = "Two implicit functions",
        grid          = true,
        line_type     = solid,
        key           = "y^2 = x^3-2*x+1",
        implicit(y^2-x^3+2*x-1, x,-4,4, y,-4,4),
        line_type     = dots,
        key           = "x^3+y^3 = 3*x*y^2-x-1",
        implicit(x^3+y^3 = 3*x*y^2-x-1, x,-4,4, y,-4,4))$
```

3D: `implicit(exprssn, x,xmin,xmax, y,ymin,ymax, z,zmin,zmax)` plots the implicit surface defined by `exprssn`, with variable `x` taking values from `xmin` to `xmax`, variable `y` taking values from `ymin` to `ymax` and variable `z` taking values from `zmin` to `zmax`. Note that `exprssn` can take one of two forms:

- $F(x, y, z)$, which `draw` assumes is equal to 0, and
- $F(x, y, z) = G(x, y, z)$.

Example: Using both forms of `exprssn`.

```
(%i1) load(draw)$
      draw3d(surface_hide = true,
        implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5) = 0.015,
            x,-1,1, y,-1.2,2.3, z,-1,1))$
      draw3d(surface_hide = true,
        implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5) - 0.015,
            x,-1,1, y,-1.2,2.3, z,-1,1))$
```

`label([string,x,y],...)` **Graphics object**

`label([string,x,y,z],...)` **Graphics object**

Options: color, label_alignment, label_orientation

All these options apply to both the 2D and the 3D forms of this command.

Note: Colored labels work only with gnuplot 4.3. This is a known bug in package `draw`.

Writes labels in 2D and 3D.

Example: In 2D, `label([string, x,y])` writes the string at point `[x,y]`.

```
(%i1) load(draw)$
      draw2d(yrange = [0.1,1.4],
```

```
        color       = red,
        label(["Label in red", 0,0.3]),
        color       = "\#0000ff",
        label(["Label in blue", 0,0.6]),
        color       = light-blue,
        label(["Label in light-blue", 0,0.9], ["Another light-blue", 0,1.2]))$
```

Example: In 3D, `label([string, x,y,z])` writes the string at point `[x,y,z]`.

```
(%i1) load(draw)$
      draw3d(explicit(exp(sin(x)+cos(x^2)), x,-3,3, y,-3,3),
        color = red,
        label(["UP 1", -2,0,3], ["UP 2", 1.5,0,4]),
        color = blue,
        label(["DOWN 1", 2,0,-3]))$
```

`parametric(xfun, yfun, par,parmin,parmax)` **Graphics object**
`parametric(xfun, yfun, zfun, par,parmin,parmax)` **Graphics object**

Options: `color`, `enhanced3d`, `key`, `line_type`, `line_width`, `nticks`

Draws parameterized curves in 2D and 3D.

2D: `parametric(xfun, yfun, par,parmin,parmax)` plots parametric function `[xfun,yfun]`, with parameter `par` taking values from `parmin` to `parmax`.

Example:

```
(%i1) load(draw)$
      draw2d(explicit(exp(x), x,-1,3),
        color = red,
        parametric(2*cos(%theta), %theta^2, %theta,0,2*%pi))$
```

3D: `parametric(xfun, yfun, zfun, par,parmin,parmax)` plots parametric curve `[xfun, yfun,zfun]`, with parameter `par` taking values from `parmin` to `parmax`.

```
(%i1) load(draw)$
      draw3d(explicit(exp(sin(x)+cos(x^2)), x,-3,3, y,-3,3),
        color        = royalblue,
        parametric(cos(5*u)^2, sin(7*u), u-2, u,0,2),
        color        = turquoise,
        line_width   = 2,
        parametric(t^2, sin(t), 2+t, t,0,2),
        surface_hide = true,
        title        = "Surface & curves")$
```

`points([[x1,y1], [x2,y2],...])` **Graphics object**
`points([x1,x2,...], [y1,y2,...])` **Graphics object**
`points([y1,y2,...])` **Graphics object**
`points([[x1,y1,z1], [x2,y2,z2],...])` **Graphics object**
`points([x1,x2,...], [y1,y2,...], [z1,z2,...])` **Graphics object**
`points(matrix)` **Graphics object**
`points(1d_y_array)` **Graphics object**
`points(1d_x_array, 1d_y_array)` **Graphics object**
`points(1d_x_array, 1d_y_array, 1d_z_array)` **Graphics object**

`points(2d_xy_array)` **Graphics object**
`points(2d_xyz_array)` **Graphics object**

Options: color, key, line_type, line_width, point_joined, point_size, point_type, and (in 3D only), enhanced3d

`points` draws individual points in 2D and 3D.

2D: `points([[x1,y1], [x2,y2],...,[xn,yn]])` or `points([x1,x2,...,xn], [y1,y2,...,yn], [z1,z2,...,zn])` plots points `[x1,y1,z1], [x2,y2,z2],...,[xn,yn,zn]`. If abscissae (horizontal coördinates) are not given, they are set to consecutive positive integers, so that `points([y1,y2,...,yn])` draws points `[1,y1], [2,y2], [3,yn]`. If `matrix` is a two-column or two-row matrix, `points(matrix)` draws the associated points. If matrix is a one-column or one-row matrix, abscissae are assigned automatically. If `1d_y_array` is a 1D lisp array of numbers, `points(1d_y_array)` again uses consecutive positive integers as abscissae. `points(1d_x_array, 1d_y_array)` plots points with their coördinates taken from the two arrays. If `2d_xy_array` is a 2D array with two columns or with two rows, `points(2d_xy_array)` plots the corresponding points on the plane.

Example: Two types of arguments for points: a list of pairs and two lists of separate coördinates.

```
(%i1) load(draw)$
      draw2d(key      = "Small points",
        points(makelist([random(20),random(50)], k,1,10)),
        point_type    = circle,
        point_size    = 3,
        points_joined = true,
        key           = "Large points",
        points(makelist(k, k,1,20),makelist(random(30), k,1,20)),
        point_type    = filled_down_triangle,
        key           = "Automatic abscissae",
        color         = red,
        points([2,12,8]))$
```

Example: Drawing impulses.

```
(%i1) load(draw)$
      draw2d(points_joined = impulses,
        line_width         = 2,
        color              = red,
        points(makelist([random(20),random(50)], k,1,10)))$
```

Example: An array with ordinates.

```
(%i1) load(draw)$
      a: make_array (flonum, 100)$
      for i:0 thru 99 do a[i]: random(1.0)$
      draw2d(points(a))$
```

Example: Two arrays with separate coördinates.

```
(%i1) load(draw)$
      x: make_array (flonum, 100)$
      y: make_array (fixnum, 100)$
      for i:0 thru 99 do (x[i]: float(i/100), y[i]: random(10))$
      draw2d(points(x,y))$
```

Example: A two-column 2D array.

```
(%i1) load(draw)$
      xy: make_array(flonum, 100, 2)$
      for i:0 thru 99 do(xy[i, 0]: float(i/100), xy[i, 1]: random(10))$
      draw2d(points(xy))$
```

Example: Drawing an array filled by function `read_array`.

```
(%i1) load(draw)$
      a: make_array(flonum,100)$
      read_array(file_search("pidigits.data"),a)$
      draw2d(yrange = [-1,10],
        points(a))$
```

3D: `points([[x1,y1,z1], [x2,y2,z2],...])` or `points([x1,x2,...], [y1,y2,...], [z1,z2,...])` plots points `[x1,y1,z1], [x2,y2,z2]`, etc. If `matrix` is a three-column or three-row matrix, `points(matrix)` draws the associated points. When arguments are lisp arrays, `points(1d_x_array, 1d_y_array, 1d_z_array)` takes coördinates from the three 1D arrays. If `2d_xyz_array` is a 2D array with three columns or with three rows, `points(2d_xyz_array)` plots the corresponding points.

Example: A 3D example.

```
(%i1) load(draw)$
      load(numericalio)$
      s2 : read_matrix(file_search("wind.data"))$
      draw3d(title = "Daily average wind speeds",
        point_size = 2,
        points(args(submatrix (s2, 4, 5))))$
```

Example: Two 3D examples in one plot.

```
(%i1) load(draw)$
      load(numericalio)$
      s2 : read_matrix(file_search("wind.data"))$
      draw3d(title = "Daily average wind speeds. Two data sets",
        point_size = 2,
        key        = "Sample from stations 1, 2 and 3",
        points(args(submatrix(s2,4,5))),
        point_type = 4,
        key        = "Sample from stations 1, 4 and 5",
        points(args(submatrix(s2,2,3))))$
```

Example: 1D arrays.

```
(%i1) load(draw)$
      x: make_array(fixnum, 10)$
      y: make_array(fixnum, 10)$
      z: make_array(fixnum, 10)$
      for i:0 thru 9 do (x[i]: random(10),
                         y[i]: random(10),
                         z[i]: random(10))$
      draw3d(points(x,y,z))$
```

Example: A colored 2D array.

```
(%i1) load(draw)$
      xyz: make_array(fixnum, 10, 3)$
      for i:0 thru 9 do (xyz[i, 0]: random(10),
                         xyz[i, 1]: random(10),
                         xyz[i, 2]: random(10))$
      draw3d(enhanced3d = true,
        points_joined   = true,
        points(xyz))$
```

`quadrilateral(point 1, point 2, point 3, point 4)` **Graphics object**

Options: border, color, fill_color, key, line_type, line_width, transform, transparent, xaxis_secondary, yaxis_secondary, and in 3D, enhanced3d

`quadrilateral` draws a quadrilateral in 2D or 3D.

Example: A quadrilateral in 2D.

```
(%i1) load(draw)$
      draw2d(quadrilateral([1,1], [2,2], [3,-1], [2,-2]))$
```

Example: A quadrilateral in 3D. Note that it is not a *plane* quadrilateral.

```
(%i1) load(draw)$
      draw3d(quadrilateral([1,1,0], [2,2,1], [3,-1,2], [2,-2,4]))$
```

`triangle(point1, point2, point3)` **Graphics object**

Options: border, color, fill_color, key, line_type, line_width, transform, transparent, xaxis_secondary, yaxis_secondary

`triangle(point1, point2, point3)` **Graphics object**

Options: color, enhanced3d, key, line_type, line_width, transform

`triangle` draws triangles in 2D and 3D.

Example: A triangle in 2D.

```
(%i1) load(draw)$
      draw2d(triangle([1,1], [2,2], [3,-1]))$
```

Example: A triangle in 3D.

```
(%i1) load(draw)$
      draw3d(triangle([1,1,0], [2,2,1], [3,-1,2]))$
```

`vector([x,y], [dx,dy])` **Graphics object**

`vector([x,y,z], [dx,dy,dz])` **Graphics object**

Options: color, head_angle head_both, head_length, head_type, key, line_type, line_width

All these options apply to both the 2D and the 3D forms of this command.

Draws vectors in 2D and 3D.

2D: `vector([x,y], [dx,dy])` plots vector `[dx,dy]` with tail at the point `(x,y)` and head at the point `(x+dx,y+dy)`

Example: 2D vectors with different options for their heads.

```
(%i1) load(draw)$
      draw2d(xrange = [0,12],
        yrange      = [0,10],
        head_length = 1,
        vector([0,1], [5,5]),
        head_type   = 'empty,   /* default type */
        vector([3,1], [5,5]),
        head_both   = true,
        head_type   = 'nofilled,
        line_type   = dots,
        vector([6,1], [5,5]))$
```

3D: `vector([x,y,z], [dx,dy,dz])` plots vector `[dx,dy,dz]` with tail at the point `(x,y,z)` and head at the point `(x+dx,y+dy,z+dz)`.

Example: Some vectors in 3D.

```
(%i1) load(draw)$
      draw3d(color = cyan,
        vector([0,0,0], [1,1,1]/sqrt(3)),
        vector([0,0,0], [1,-1,0]/sqrt(2)),
        vector([0,0,0], [1,1,-2]/sqrt(6)))$
```

4.4.2 Graphic objects used in 2D plots only

`bars([x1,h1,w1], [x2,h2,w2],...)` **Graphics object**

Options: fill_color, fill_density, key, line_width

Draws vertical bars in 2D. `bars([x1,h1,w1], [x2,h2,w2],...)` draws bars centered at values `x1, x2,...` with heights `h1, h2,...` and widths `w1, w2,...`.

```
(%i1) load(draw)$
      draw2d(key     = "Group A",
        fill_color   = blue,
        fill_density = 0.2,
        bars([0.8,5,0.4], [1.8,7,0.4], [2.8,-4,0.4]),
        key= "Group B",
        fill_color   = red,
        fill_density = 0.6,
        line_width   = 4,
        bars([1.2,4,0.4], [2.2,-2,0.4], [3.2,5,0.4]),
        xaxis        = true);
```

`ellipse(xc, yc, a,b, polarang,Δpolarang)` **Graphics object**

Options: border, color, fill_color, key, line_type, line_width, nticks, transparent

`ellipse(xc, yc, a,b, polarang,Δploarang)` plots an elliptical arc centered at `[xc, yc]`, with horizontal and vertical semi-axes `a` and `b`, respectively, starting at polar angle `polarang` and ending at polar angle `ang`+`Δpolarang`. `Δpolarang` is allowed to be negative.

Example:

```
(%i1) load(draw)$
      draw2d(transparent = false,
        fill_color       = red,
```

```
color            = gray30,
transparent      = false,
line_width       = 5,
ellipse(0, 6, 3,2, 270,-270),
/* center (x,y), a, b, start and end in degrees */
transparent      = true,
color            = blue,
line_width       = 3,
ellipse(2.5, 6, 2,3, 30,-90),
xrange           = [-3,6],
yrange           = [2,9])$
```

errors([x1,x2,...,xn], [y1,y2,...,yn]) **Graphics object**

Options: error_type, fill_density, color, line_width, key, line_type, points_joined, xaxis_secondary, yaxis_secondary

See also: error_type

errors draws points with error bars. The error bars may be horizontal, vertical, or both, according to the value of option error_type. If error_type = x, arguments to errors must be of the form [x, y, xdelta] or [x, y, xlow, xhigh]. If error_type = y, arguments must be of the form [x, y, ydelta] or [x, y, ylow, yhigh]. If error_type = xy or if error_type = boxes, arguments to errors must be of the form [x, y, xdelta, ydelta] or [x, y, xlow, xhigh, ylow, yhigh].

Note: Option fill_density is only relevant when error_type = boxes.

Example: Horizontal error bars. Note that two of them cannot be seen, due to the software's automatic choice of yrange.

```
(%i1) load(draw)$
      draw2d(error_type = x,
        errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```

Example: Horizontal error bars with yrange adjusted to make all the bars visible. Note that yrange is such that two of the bars are cut off.

```
(%i1) load(draw)$
      draw2d(yrange = [1, 7],
        error_type  = x,
        errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```

Example: Horizontal error bars with xrange and yrange adjusted to make all the bars visible. Note that yrange is such that two of the bars are cut off.

```
(%i1) load(draw)$
      draw2d(xrange = [-1, 20],
        yrange      = [1, 7],
        error_type  = x,
        errors([[1,2,1], [3,5,3], [10,3,1], [17,6,2]]))$
```

Example: Vertical error bars.

```
(%i1) load(draw)$
      draw2d(error_type = y,
        errors([[1,2,1,2], [3,5,3,7], [10,3,2,4], [17,6,1/2,2]]))$
```

Example: Vertical and horizontal error bars.

```
(%i1) load(draw)$
      draw2d(error_type = xy,
        errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]))$
```

Example: Error boxes. Note that yrange is not automatically adjusted to accommodate the full heights of all the error boxes.

```
(%i1) load(draw)$
      draw2d(error_type = boxes,
        errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]))$
```

Example: Error boxes, with yrange and yrange adjusted to accommodate the error boxes. Note that the error boxes are not transparent, making it easy to see where they overlap.

```
(%i1) load(draw)$
      draw2d(xrange = [-1, 20],
        yrange      = [0, 10],
        error_type  = boxes,
        errors([[1,2,1,2], [3,5,2,1], [10,3,1,1], [17,6,1/2,2]]))$
```

image(im, x0,y0, width,height) — Graphics object

See also: colorbox, make_level_picture, make_RGB_picture, palette, read_xpm

image renders images in 2D. image(im,x0,y0,width,height) plots image im in the rectangular region from vertex (x0,y0) to (x0+width,y0+height) on the real plane. Argument im must be a matrix of real numbers, a matrix of vectors of length three, or a picture object. If im is a matrix of real numbers or a levels picture object, pixel values are interpreted according to graphics option palette. If im is a matrix of vectors of length three or an rgb picture object, they are interpreted as red, green and blue color components.

Example: Changing the colors of an image using option palette.

```
(%i1) load(draw)$
      im: apply('matrix,
            makelist(makelist(random(200), i,1,30), i,1,30))$
      palette        = color, /* default */
        draw2d(image(im,0,0,30,30))$
      draw2d(palette = gray,
        image(im, 0,0, 30,30))$
      draw2d(palette = [15,20,-4],
        colorbox     = false,
        image(im, 0,0, 30,30))$
```

Example: A matrix of RGB triples.

```
(%i1) load(draw)$
      im: apply('matrix,
            makelist(
              makelist(
                [random(300),random(300),random(300)],
              i,1,30),
            i,1,30))$
      draw2d(image(im, 0,0, 30,30))$
```

Example: A `level picture` object is built by hand and then rendered.

```
(%i1) load(draw)$
      im: make\_level\_picture([45,87,2,134,204,16],3,2);
(%o2) picture(level, 3, 2, Lisp array [6])

      /* default color palette */
      draw2d(image(im, 0,0, 30,30))$
      /* gray palette */
      draw2d(palette = gray, image(im, 0,0, 30,30))$
```

Example: An xpm file is read and rendered.

```
(%i1) load(draw)$
      im: read_xpm("myfile.xpm")$
      draw2d(image(im, 0,0, 10,7))$
```

See http://www.telefonica.net/web2/biomates/maxima/gpdraw/image for more elaborate examples.[2]

`polar(radius, ang,minang,maxang)` **Graphics object**

Options: color, key, line_type, line_width, nticks

`polar(radius, ang,minang,maxang)` plots function `radius(ang)` defined in polar coördinates, with variable `ang` taking values from `minang` to `maxang`.

Example:

```
(%i1) load(draw)$
      draw2d(user_preamble = "set grid polar",
        nticks     = 200,
        xrange     = [-5,5],
        yrange     = [-5,5],
        color      = blue,
        line_width = 3,
        title      = "Hyperbolic Spiral",
        polar(10/theta, theta,1,10*%pi))$
```

`polygon([[x1,y1], [x2,y2],...])` **Graphics object**
`polygon([x1,x2,...], [y1,y2,...])` **Graphics object**

Options: border, color, fill_color, key, line_type, line_width, transparent

`polygon([[x1,y1], [x2,y2],...])` or `polygon([x1,x2,...], [y1,y2,...])` plots on the plane a polygon with vertices [x1,y1], [x2,y2], etc.

Example: A pair of triangles.

```
(%i1) load(draw)$
      draw2d(color  = "\#e245f0",
        line_width  = 8,
        polygon([[3,2], [7,2], [5,5]]),
        border      = false,
        fill_color  = yellow,
        polygon([[5,2], [9,2], [7,5]]))$
```

[2] This link worked as recently as 2012-02-02.

`rectangle([x1,y1], [x2,y2])` **Graphics object**

Options: border, color, fill_color, key, line_type, line_width, transparent

`rectangle([x1,y1], [x2,y2])` draws a rectangle with opposite vertices `[x1,y1]` and `[x2,y2]`.

Example: Two rectangles.

```
(%i1) load(draw)$
      draw2d(fill\_color = red,
        line_width       = 6,
        line_type        = dots,
        transparent      = false,
        fill_color       = blue,
        rectangle([-2,-2], [8,-1]), /* opposite vertices */
        transparent      = true,
        line_type        = solid,
        line_width       = 1,
        rectangle([9,4], [2,-1.5]),
        xrange           = [-3,10],
        yrange           = [-3,4.5])$
```

`region(expr, var1,minval1,maxval1, var2,minval2,maxval2)` **Graphics object**

Options: `fill_color, key, x_voxel, and y_voxel`

`region` plots a region on the plane defined by inequalities. `expr` is an expression formed by inequalities and boolean operators `and`, `or`, and `not`. The region is bounded by the rectangle defined by `[minval1, maxval1]` and `[minval2, maxval2]`.

Example: A region in the plane.

```
(%i1) load(draw)$
      draw2d(x_voxel = 30,
        y_voxel      = 30,
        region(x^2+y^2<1 and x^2+y^2 > 1/2, x,-1.5,1.5, y, -1.5, 1.5))$
```

4.4.3 Graphic objects used in 3D plots only

`cylindrical(radius, z,minz,maxz, polang,minpolang,maxpolang)` **Graphics object**

Options: color, key, line_type, xu_grid, yv_grid

Draws 3D functions defined in cylindrical coördinates. The functions are of the form $radius = f(\theta,\ z)$, where θ is the polar angle (or, azimuth). The variable `z` taking values from `minz` to `maxz` and the polar angle `polang` taking values from `minpolang` to `maxpolang`.

Example:

```
(%i1) load(draw)$
      draw3d(cylindrical(1, z,-2,2, %theta,0,2*%pi))$
```

`elevation_grid(mat, x0,y0, width,height)` **Graphics object**

Options: color, enhanced3d, key, line_type, line_width

See also: mesh

Draws matrix `mat` in 3D space. z-values are taken from `mat`, the abscissae range from `x0` to `x0+width` and ordinates from `y0` to `y0+height`. Element `mat(1,1)` is projected on point

(x0,y0+height), mat(1,n) on (x0+width,y0+height), mat(m,1) on (x0,y0), and mat(m,n) on (x0+width,y0). The points are joined by line segments.

Note: In older versions of wxMaxima, elevation_grid was called mesh.

Example: Random elevations.

```
(%i1) load(draw)$
      m: apply(matrix,
           makelist(makelist(random(10.0), k,1,30), i,1,20))$
      draw3d(xlabel  = "x",
        ylabel       = "y",
        surface_hide = true,
        elevation_grid(m,0,0,3,2))$
```

mesh(row1, row2,...,rown) — Graphics object

Options: color, enhanced3d, key, line_type, line_width, transform

The rows that mesh takes as arguments are lists of points in 3D, each of which is a list of the form [x_i, y_i, j_i]. All the rows have to have the same number of points in them. Taken together, the rows of points define a surface in 3 dimensions. In some sense, mesh is a generalization of the elevation_grid object.[3]

Example: A simple mesh plot.

```
(%i1) load(draw)$
      draw3d(mesh([[1,1,3], [7,3,1], [12,-2,4], [15,0,5]],
        [[2,7,8], [4,3,1], [10,5,8], [12,7,1]],
        [[-2,11,10], [6,9,5], [6,15,1], [20,15,2]]))$
```

Example: Plotting a triangle in 3D.

```
(%i1) load(draw)$
      draw3d(line_width = 2,
        mesh([[1,0,0],[0,1,0]], [[0,0,1],[0,0,1]]))$
```

Example: Two quadrilaterals.

```
(%i1) load(draw)$
      draw3d(surface_hide = true,
        line_width        = 3,
        color             = red,
        mesh([[0,0,0], [0,1,0]], [[2,0,2], [2,2,2]]),
```

[3] **Note:** In older versions of wxMaxima, mesh was used as follows:

mesh(mat, x0,y0, width,height) — **Graphic object**

Options: color, enhanced3d, key, line_type, line_width, transform

mesh plots the data in matrix mat in 3D space. The coördinates for the plotted points are constructed as follows: The x-coördinates range from x0 to x0 + width and the y-coördinates from y0 to y0 + height. The z-coördinates are taken from mat. **Note:** This graphics object ignores enhanced3d values other than true and false.

Example: Using mesh in older versions of wxMaxima.

```
(%i1) load(draw)$
(%i2) m :  apply('matrix,
makelist(makelist(random(10.0), k,1,30), i,1,20))$
(%i3) draw3d(color = blue,
mesh(m, 0,0, 3,2),
xlabel = "x",
ylabel = "y",
surface_hide = true)$
```

```
          color             = blue,
          mesh([[0,0,2], [0,1,2]], [[2,0,4], [2,2,4]]))$
```

parametric_surface(xfun, yfun, zfun, par1,par1min,par1max, par2,par2min,par2max) **Graphics object**

Options: color, enhanced3d, key, line_type, line_width, xu_grid, yv_grid

parametric_surface plots parametric surface [xfun,yfun,zfun], with parameter par1 taking values from par1min to par1max and parameter par2 taking values from par2min to par2max.

Example:

```
(%i1) load(draw)$
      draw3d(title     = "Sea shell",
        surface_hide   = true,
        rot_vertical   = 100,
        rot_horizontal = 20,
        xu_grid        = 100,
        yv_grid        = 25,
        parametric_surface(0.5*u*cos(u)*(cos(v)+1),
          0.5*u*sin(u)*(cos(v)+1),
          u*sin(v)-((u+3)/8*%pi)^2-20,
          u,0,13*%pi, v,-%pi,%pi))$
```

spherical(radius, polang,minpolang,maxpolang, colat,mincolat,maxcolat) **Graphics object**

Options: color, key, line_type, xu_grid, yv_grid

spherical(radius, polang,minpolang,maxpolang, colat,mincolat,maxcolat) plots function radius(polang, colat) defined in spherical coördinates, with polar angle polang taking values from minpolang to maxpolang and colatitude colat taking values from mincolat to maxcolat. **Note: The package** vect **redefines the function** spherical. Loading vect renders spherical unusable as a graphics object until draw is loaded again.

Example: The unit sphere in spherical coördinates.

```
(%i1) load(draw)$
      draw3d(spherical(1, %theta,0,2*%pi, phi,0,%pi))$
```

tube(xfun,yfun,zfun,rfun, p,pmin,pmax) **Graphics object**

Options: color, enhanced3d, key, line_type, line_width, tube_extremes, xu_grid, yv_grid

Draws a tube in 3D (with possibly varying diameter) around the parametric curve given by [xfun,yfun,zfun]. Circles of radius rfun are placed with their centers on the curve and perpendicular to it.

Example: A basic tube.

```
(%i1) load(draw)$
      draw3d(enhanced3d = true,
        xu_grid         = 50,
        tube(cos(a), a, 0, cos(a/10)^2, a, 0, 4*%pi))$
```

4.5 Options for draw and its variants

4.5.1 Options used with both 2D and 3D graphics objects

background_color **Graphics option**

Default value: white

See also: color

Sets the background color for terminals gif, png, jpg, and gif.

color **Graphics option**

Default value: blue

See also: fill_color

color specifies the color for plotting borders of polygons, lines, points, and labels. Available color names are: white, black, gray0, grey0, gray10, grey10, gray20, grey20, gray30, grey30, gray40, grey40, gray50, grey50, gray60, grey60, gray70, grey70, gray80, grey80, gray90, grey90, gray100, grey100, gray, grey, light-gray, light-grey, dark-gray, dark-grey, red, light-red, dark-red, yellow, light-yellow, dark-yellow, green, light-green, dark-green, spring-green, forest-green, sea-green, blue, light-blue, dark-blue, midnight-blue, navy, medium-blue, royalblue, skyblue, cyan, light-cyan, dark-cyan, magenta, light-magenta, dark-magenta, turquoise, light-turquoise, dark-turquoise, pink, light-pink, dark-pink, coral, light-coral, orange-red, salmon, light-salmon, dark-salmon, aquamarine, khaki, dark-khaki, goldenrod, light-goldenrod, dark-goldenrod, gold, beige, brown, orange, dark-orange, violet, dark-violet, plum, and purple.

Colors can be given as names or in hexadecimal RGB code, in the form "#rrggbb".

Example: Colored plots.

```
(%i1) load(draw)$
      draw2d(explicit(x^2,x,-1,1), /* default is blue */
        color = red,
        explicit(0.5 + x^2,x,-1,1),
        color = black,
        explicit(1 + x^2,x,-1,1),
        color = light-blue,
        explicit(1.5 + x^2,x,-1,1),
        color = "\#23ab0f",
        label(["This is a label",0,1.2]))$
```

Notes: In older versions of wxMaxima, (1) the default color was black and (2) any color name with a hyphen in it had to be offset by double quotes (color = "light-blue" and not color = light-blue). As of version 5.24.0, the double quotes for named colors are optional, but colors given in hexadecimal RGB code still need the double quotes.

colorbox **Global graphics option**

Default value: true

See also: palette

If colorbox is true, colored 3D graphics objects are accompanied in the graphics window by a color scale. The same is true of 2D image objects (but no other kind of 2D object).

Example: A 2D image object.

```
(%i1) load(draw)$
      im: apply('matrix,
        makelist(makelist(random(200),i,1,30),i,1,30))$
```

```
draw2d(image(im,0,0,30,30))$
draw2d(colorbox = false,
  image(im,0,0,30,30))$
```

Example: Color scale and a 3D colored object.

```
(%i1) load(draw)$
      draw3d(colorbox = "Magnitude",
        enhanced3d    = true,
        explicit(x^2+y^2,x,-1,1,y,-1,1))$
```

head_angle — Graphics option

Default value: 45

Applies to: vector graphics objects

See also: head_both, head_length, head_type

head_angle controls the shape of the heads of vectors by giving the angle, in degrees, between the edges of the triangles that comprise the heads, the segment to which they're attached.

Example: Various head_angles.

```
(%i1) load(draw)$
      draw2d(xrange = [0,10],
        yrange      = [0,9],
        head_length = 0.7,
        head_angle  = 10,
        vector([1,1],[0,6]),
        head_angle  = 20,
        vector([2,1],[0,6]),
        head_angle  = 30,
        vector([3,1],[0,6]),
        head_angle  = 40,
        vector([4,1],[0,6]),
        head_angle  = 60,
        vector([5,1],[0,6]),
        head_angle  = 90,
        vector([6,1],[0,6]),
        head_angle  = 120,
        vector([7,1],[0,6]),
        head_angle  = 160,
        vector([8,1],[0,6]),
        head_angle  = 180,
        vector([9,1],[0,6]))$
```

head_both — Graphics option

Default value: false

Applies to: vector graphics objects

See also: head_angle, head_length, head_type

If head_both is true, vectors are plotted with arrow heads at both ends. If false, only one arrow head is plotted, at the terminal point of the vector (the point traditionally called the "head" of the vector).

Example: A vector with and without both heads.

```
(%i1) load(draw)$
      draw2d(xrange = [0,8],
        yrange      = [0,8],
        head_length = 0.7,
        vector([1,1],[6,0]),
        head_both  = true,
        vector([1,7],[6,0]))$
```

head_length **Graphics option**

Default value: 2

Applies to: vector graphics objects

See also: head_both, head_angle, head_type

head_length indicates, in the same units as on the x-axis, the length of arrow heads.

Example: Various head_lengths.

```
(%i1) load(draw)$
      draw2d(xrange = [0,12],
        yrange      = [0,8],
        vector([0,1],[5,5]),
        head_length = 1,
        vector([2,1],[5,5]),
        head_length = 0.5,
        vector([4,1],[5,5]),
        head_length = 0.25,
        vector([6,1],[5,5]))$
```

head_type **Graphics option**

Default value: filled

Applies to: vector graphics objects

See also: head_angle, head_both, head_length

head_type is used to specify how arrow heads are plotted. Possible values are: filled (closed and filled arrow heads), empty (closed but not filled arrow heads), and nofilled (open arrow heads).

Example: Various head_types.

```
(%i1) load(draw)$
      draw2d(xrange = [0,12],
        yrange      = [0,10],
        head_length = 1,
        vector([0,1],[5,5]), /* default type */
        head_type   = 'empty,
        vector([3,1],[5,5]),
        head_type   = 'nofilled,
        vector([6,1],[5,5]))$
```

label_alignment **Graphics option**

Default value: center

Applies to: label graphics objects

See also: color, label_orientation

label_alignment is used to specify where to write labels with respect to the given coördinates. Possible values are: center, left, and right.

Example: All three label alignments.

```
(%i1) load(draw)$
      draw2d(xrange      = [0,10],
        yrange           = [0,10],
        points_joined    = true,
        points([[5,0],[5,10]]),
        color            = blue,
        label(["Centered alignment (default)",5,2]),
        label_alignment = 'left,
        label(["Left alignment",5,5]),
        label_alignment = 'right,
        label(["Right alignment",5,8]))$
```

label_orientation — Graphics option

Default value: horizontal

Applies to: label graphics objects

See also: color, label_alignment

label_orientation is used to specify orientation of labels. Possible values are: horizontal and vertical.

Example: Both label orientations. (In this example, a dummy point is needed, because draw cannot draw a scene without something to draw.)

```
(%i1) load(draw)$
      draw2d(xrange       = [0,10],
        yrange            = [0,10],
        point_size        = 0,
        points([[5,5]]),
        color             = navy,
        label(["Horizontal orientation (default)",5,2]),
        label_orientation = 'vertical,
        color             = "\#654321",
        label(["Vertical orientation",1,5]))$
```

line_type — Graphics option

Default value: solid

Applies to: In 2D: ellipse, explicit, implicit, parametric, points, polar, polygon, rectangle, vector; in 3D: explicit, parametric, parametric_surface, points

See also: line_width

line_type indicates how lines are displayed. Its possible values are solid and dots.

Example: Showing both options for line_type.

```
(%i1) load(draw)$
      draw2d(line_type = dots,
        explicit(1 + x^2,x,-1,1),
        line_type      = solid, /* default */
        explicit(2 + x^2,x,-1,1))$
```

line_width — Graphics option

Default value: 1

Applies to: In 2D: ellipse, explicit, implicit, parametric, points, polar, polygon, rectangle, vector; In 3d: parametric, points

See also: line_type

line_width is the width of plotted lines. Its value must be a positive number.

Example: Three different line_widths.

```
(%i1) load(draw)$
      draw2d(explicit(x^2, x,-1,1), /* default width */
        line_width = 5.5,
        explicit(1 + x^2, x,-1,1),
        line_width = 10,
        explicit(2 + x^2, x,-1,1))$
```

nticks — Graphics option

Default value: 29

See also: adapt_depth

Applies to: In 2D, objects ellipse, explicit, parametric, polar; in 3D, only parametric objects.

draw uses an adaptive plotting routine. It begins by taking the interval over which the desired graphics object is to be plotted and subdivides it according to the value of option nticks. When it detects rapid changes in the shape of the object being plotted, it splits (or, subdivides) the subinterval(s) in question and therefore plots more points.

Example: The effect of the nticks option.

```
(%i1) load(draw)$
      draw2d(transparent = true,
        ellipse(0,0, 4,2, 0,180),
        nticks           = 5,
        ellipse(0,0, 4,2, 180,180))$
```

palette — Global graphics option

Default value: color

See also: colorbox

When draw plots certain graphics objects, it creates a matrix of real numbers[4] that represent colors or shades of gray, to be applied at the points of the image. palette indicates how to map the numbers in the matrix onto color components. There are two ways of defining this map.

First, palette can be a vector of length three with components ranging from −36 to +36; each value is an index for a formula mapping the levels onto red, green and blue colors, respectively:

```
 0: 0              1: 0.5            2: 1
 3: x              4: x^2            5: x^3
 6: x^4            7: sqrt(x)        8: sqrt(sqrt(x))
 9: sin(90x)      10: cos(90x)      11: |x-0.5|
12: (2x-1)^2      13: sin(180x)     14: |cos(180x)|
15: sin(360x)     16: cos(360x)     17: |sin(360x)|
18: |cos(360x)|   19: |sin(720x)|  20: |cos(720x)|
```

[4] Floating point numbers, actually, of course.

```
21: 3x               22: 3x-1          23: 3x-2
24: |3x-1|           25: |3x-2|        26: (3x-1)/2
27: (3x-2)/2         28: |(3x-1)/2|    29: |(3x-2)/2|
30: x/0.32-0.78125   31: 2*x-0.84      32: 4x; 1; -2x+1.84; x/0.08-11.5
33: |2*x - 0.5|      34: 2*x           35: 2*x - 0.5
36: 2*x - 1
```

If an entry in the vector `palette` is negative, the corresponding negative color component will be used.

Note: `palette = gray` and `palette = color` are short cuts for `palette = [3,3,3]` and `palette = [7,5,15]`, respectively.

Example: Compare the results of plotting the same `image` graphics object using various vectors of length 3 as the values of `palette`.

```
(%i1) load(draw)$
      im: apply('matrix,
          makelist(makelist(random(200), i,1,30), i,1,30))$
      draw2d(image(im, 0,0, 30,30))$    /* palette = color, default */
      draw2d(palette = gray,
        image(im, 0,0, 30,30))$
      draw2d(palette = [15,20,-4],
        colorbox     = false,
        image(im, 0,0, 30,30))$
```

Example: `palette` and `enhanced3d` together.

```
(%i1) load(draw)$
      draw3d(enhanced3d = [z-x+2*y, x,y,z],
        palette         = [32, -8, 17],
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3))$
```

Second, `palette` can be a user-defined lookup table. In this case, the format for building a lookup table of length n is `palette=[color_1, color_2,...,color_n]`, where `color_i` is a well formed color (see option color), such that `color_1` is assigned to the lowest gray level and `color_n` to the highest. The rest of colors are interpolated.

Example: Setting `palette` equal to a user-defined lookup table.

```
(%i1) load(draw)$
      draw3d(palette = [red, blue, yellow],
        enhanced3d   = x,
        explicit(x^2+y^2, x,-1,1, y,-1,1)) $
```

points_joined — Graphics option

Default value: `false`

Applies to: `points` graphics object

When `points_joined` is `true`, points are joined by lines; when `false`, isolated points are drawn. A third possible value for this graphics option is `impulses`, in which case, vertical segments are drawn from points to the x-axis (2D) or to the x, y-plane (3D).

Example: Points joined by different types of lines.

```
(%i1) load(draw)$
      draw2d(xrange   = [0,10],
```

```
        yrange        = [0,4],
        point_size    = 3,
        point_type    = up_triangle,
        color         = blue,
        points([[1,1], [5,1], [9,1]]),
        points_joined = true,
        point_type    = square,
        line_type     = dots,
        points([[1,2], [5,2], [9,2]]),
        point_type    = circle,
        color         = red,
        line_width    = 7,
        points([[1,3], [5,3], [9,3]]))$
```

point_size — Graphics option

Default value: 1

Applies to: points graphics objects, in both 2D and 3D

point_size sets the size for plotted points. It must be a non-negative number. **Note:** point_size has no effect when graphics option point_type is set to dot.

Example: 10 small points and 20 large points.

```
(%i1) load(draw)$
      draw2d(points(makelist([random(20),random(50)], k,1,10)),
        point_size = 5,
        points(makelist(k, k,1,20), makelist(random(30), k,1,20)))$
```

point_type — Graphics option

Default value: 1

Applies to: points graphics objects, in both 2D and 3D

point_type determines the displayed shape of isolated points. The value of this option can be any of the following point styles. Either the name of the point style or the corresponding integer can be used.

```
-1: none                  0: dot                   1: plus
 2: multiply              3: asterisk              4: square
 5: filled_square         6: circle                7: filled_circle
 8: up_triangle           9: filled_up_triangle   10: down_triangle
11: filled_down_triangle  12: diamant             13: filled_diamant
```

Example: Several different points styles.

```
(%i1) load(draw)$
      draw2d(xrange = [0,10],
        yrange      = [0,10],
        point_size  = 3,
        point_type  = diamant,
        points([[1,1], [5,1], [9,1]]),
        point_type  = filled_down_triangle,
        points([[1,2], [5,2], [9,2]]),
        point_type  = asterisk,
        points([[1,3], [5,3], [9,3]]),
```

```
  point_type  = filled_diamant,
  points([[1,4], [5,4], [9,4]]),
  point_type  = 5,
  points([[1,5], [5,5], [9,5]]),
  point_type  = 6,
  points([[1,6], [5,6], [9,6]]),
  point_type  = filled_circle,
  points([[1,7], [5,7], [9,7]]),
  point_type  = 8,
  points([[1,8], [5,8], [9,8]]),
  point_type  = filled_diamant,
  points([[1,9], [5,9], [9,9]]))$
```

transform — Graphics option

Default value: none

Effects a transformation of 2D or 3D space. If transform is none, the space is not transformed and graphics objects are drawn as defined. When a transformation is desired, a list must be assigned to option transform. In the case of a 2D scene, the list takes the form [f1(x,y), f2(x,y), x,y]. In the case of a 3D scene, the list is of the form [f1(x,y,z), f2(x,y,z), f3(x,y,z),x,y,z]. The names of the variables defined in the lists may be different from those used in the definitions of the graphics objects.

Example: Rotation in 2D.

```
(%i1) load(draw)$
      th : %pi / 4$
      draw2d(color        = "\#e245f0",
        proportional_axes = xy,
        line_width        = 8,
        triangle([3,2], [7,2], [5,5]),
        border            = false,
        fill_color        = yellow,
        transform         = [cos(th)*x - sin(th)*y,
          sin(th)*x + cos(th)*y, x,y],
        triangle([3,2], [7,2], [5,5]))$
```

Example: Translation in 3D.

```
(%i1) load(draw)$
      draw3d(color = "\#a02c00",
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3),
        transform  = [x+10,y+10,z+10, x,y,z],
        color      = blue,
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3))$
```

unit_vectors — Graphics option

Default value: false

This option affects only vector graphics objects.

If unit_vectors is true, vectors are plotted with norm 1. (This is handy for plotting vector fields.) If unit_vectors is false, vectors are plotted with their actual norms.

Example: A vector and a unit_vector.

```
(%i1) load(draw)$
      draw2d(xrange  = [-1,6],
        yrange       = [-1,6],
        head_length  = 0.1,
        vector([0,0], [5,2]),
        unit_vectors = true,
        color        = red,
        vector([0,3], [5,2]))$
```

4.5.2 Options used with 2D graphics objects only

adapt_depth **Graphics option**

Default value: 10

See also: nticks

Applies to: 2D explicit plots

draw uses an adaptive plotting routine. It begins by taking the interval over which the desired graphics object is to be plotted and subdivides it according to the value of option nticks. When it detects rapid changes in the shape of the object being plotted, it splits (or, subdivides) the subinterval(s) in question and therefore plots more points. adapt_depth is the maximum number of splittings used by the adaptive plotting routine. The value of adapt_depth should be set higher for wilder plots.

border **Graphics option**

Default value: true

If border is true, borders of polygons are painted according to line_type and line_width. This option affects the following 2D graphics objects: ellipse, polygon, and rectangle.

Example: A figure with border and a figure without.

```
(%i1) load(draw)$
      draw2d(color = brown,
        line_width = 8,
        polygon([[3,2], [7,2], [5,5]]),
        border     = false,
        fill_color = blue,
        polygon([[5,2], [9,2], [7,5]]))$
```

error_type **Graphics option**

Default value: y

See also: errors

If error_type is set equal to x, y, or xy, graphics object errors will draw points with horizontal, vertical, or both, error bars. When error_type = boxes, boxes will be drawn instead of crosses.

fill_color **Graphics option**

Default value: red

See also: color

fill_color specifies the color for filling polygons and 2D explicit functions. See color to learn how colors are specified.

fill_density **Graphics option**

Default value: 0
See also: bars
fill_density is a number between 0 and 1 that specifies the intensity of the fill_color in bars objects. See bars for examples.

filled_func — Graphics option

Default value: false
See also: fill_color, explicit
Option filled_func controls how regions bounded by functions should be filled. When filled_func is true, the region bounded by the function defined with object explicit and the bottom of the graphics window is filled with the color fill_color. To fill a region bounded by the graphs of two functions, put one of the functions in an object of type explicit and set filled_func equal to the other. By default, explicit functions are not filled.

Example: Region bounded above by an explicit object and below by the bottom of the graphics window.

```
(%i1) load(draw)$
      draw2d(fill_color = red,
        filled_func     = true,
        explicit(sin(x),x,0,10))$
```

Example: Region bounded by an explicit object and the function defined by option filled_func. Note that the variable in filled_func must be the same as that used in explicit.

```
(%i1) load(draw)$
      draw2d(fill_color = grey,
        filled_func     = sin(x),
        explicit(-sin(x), x,0,%pi))$
```

ip_grid — Graphics option

Default value: [50, 50]
Applies to: implicit graphics objects in 2D plots
ip_grid tells draw how many points to plot in each implicit object. The default of [50, 50] indicates that $51 \times 51 = 2601$ points will be used throughout the plotting region for plotting the requested object.[5]

ip_grid_in — Graphics option

Default value: [5, 5]
Applies to: implicit graphics objects in 2D plots
ip_grid_in sets the grid for the second sampling in implicit plots. If draw decides it needs to plot points more densely in some area of an implicit plot, it checks ip_grid_in to see how many additional points to plot.

transparent — Graphics option

Default value: false
Applies to: ellipse, polygon, rectangle
If transparent is false, interior regions of ellipses, polygons, and rectangles are filled according to fill_color. If transparent is true, the interior regions are not filled.

Example: The effect of the transparent option.

[5] It may actually be $50 \times 50 = 2500$ points, but I don't think so, based on experiments I've done.

```
(%i1) load(draw)$
      draw2d(polygon([[3,2], [7,2], [5,5]]),
        transparent = true,
        polygon([[5,2], [9,2], [7,5]]))$
```

4.5.3 Options used with 3D graphics objects only

axis_3d **Global graphics option**

Default value: true

See also: axis_bottom, axis_left, axis_right, axis_top

If axis_3d is true, the x, y and z-axis are shown in 3D scenes.

Example: Plots with and without the 3D axes.

```
(%i1) load(draw)$
      draw3d(axis_3d = false,
        explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
      draw3d(axis_3d = true,
        explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
```

cbrange **Global graphics option**

Default value: auto

See also: enhanced3d, colorbox, cbtics

If cbrange is auto and both options colorbox and enhanced3d are not false, the range for the values which are colored in a 3D graphics object is computed automatically. (Any values outside the color range are assigned color of the nearest extreme value.) When enhanced3d or colorbox is false, option cbrange has no effect. If the user wants a specific interval for the colored values, it must be given as a wxMaxima list, as in cbrange=[-2, 3].

Example: The effect of the enhanced3d option.

```
(%i1) load(draw)$
      draw3d(enhanced3d = true,
        cbrange         = [-3,10],
        explicit(x^2+y^2, x,-2,2, y,-2,2));
```

cbtics **Global graphics option**

Default value: auto

See also: enhanced3d, colorbox, cbrange, xtics

This graphics option controls the way tic marks are drawn on the colorbox, when options colorbox and enhanced3d are both true. When either of these options is false, option cbtics has no effect. See xtics for a complete description.

Example: cbtics in use with the enhanced3d option.

```
(%i1) load(draw)$
      draw3d(enhanced3d = true,
        cbtics          = {["High",10], ["Medium",05], ["Low",0]},
        cbrange         = [0,10],
        explicit(x^2+y^2, x,-2,2, y,-2,2))$
```

contour **Global graphics option**

Default value: none

Allowable values: base, both, map, none, surface

See also: contour_levels

Contour lines can be included in certain kinds of 3D plots. Option contour enables the user to select where the contour lines are plotted. The possible values of contour are:

- none: no contour lines are plotted.
- base: contour lines are projected on the x, y-plane.
- surface: contour lines are plotted on the surface.
- both: contour lines are plotted both on the x, y-plane and on the surface.
- map: contour lines are projected on the x, y-plane, and the view point is set as if the viewer were looking down the z-axis.

Example: Contours both on the surface and on the "floor" of the plot.

```
(%i1) load(draw)$
      draw3d(contour_levels = 15,
        contour             = both,
        surface_hide        = true,
        explicit(20*exp(-x^2-y^2)-10, x,0,2, y,-3,3))$
```

contour_levels **Global graphics option**

Default value: 5

See also: contour

This graphics option controls the way contours are drawn. contour_levels can be set to a positive integer number, a list of three numbers or an arbitrary set of numbers:

- When option contour_levels is bound to the positive integer n, n contour lines will be drawn at equal intervals. By default, five equally spaced contours are plotted.
- When option contour_levels is bound to a list of length three of the form [lowest, s, highest], contour lines are plotted from lowest to highest in steps of s.
- When option contour_levels is bound to a set of numbers of the form {n1, n2,...,n}, contour lines are plotted at levels n1, n2,...,n

Example: Ten equally spaced contour lines. The actual number of levels can be adjusted to give simple labels.

```
(%i1) load(draw)$
      draw3d(color     = green,
        explicit(20*exp(-x^2-y^2)-10, x,0,2, y,-3,3),
        contour_levels = 10,
        contour        = both,
        surface_hide   = true)$
```

Example: Contour lines from -8 to 8 in steps of 4.

```
(%i1) load(draw)$
      draw3d(color     = green,
        explicit(20*exp(-x^2-y^2)-10, x,0,2, y,-3,3),
        contour_levels = [-8,4,8],
        contour        = both,
        surface_hide   = true)$
```

Example: Contour lines at levels -7, -6, 0.8 and 5.

```
(%i1) load(draw)$
      draw3d(color     = green,
        explicit(20*exp(-x^2-y^2)-10, x,0,2, y,-3,3),
        contour_levels = {-7, -6, 0.8, 5},
        contour        = both,
        surface_hide   = true)$
```

enhanced3d **Graphics option**

Default value: none

See also: color, palette, surface_hide

If enhanced3d is none, surfaces in 3D plots are not colored. In order to get a colored surface, a list must be assigned to option enhanced3d. The first element of the list is an expression and the rest are the names of the variables or parameters used in that expression. A list such as [f(x,y,z), x, y, z] means that point [x,y,z] of the surface is assigned number f(x,y,z), which will be colored according to the current palette. In this way, the height of the surface above the x, y-plane is indicated by color. For those 3D graphics objects defined in terms of parameters, the color number can be defined in terms of the parameters, as well. With objects parametric and tube, use [f(u), u]. With object parametric_surface, use [f(u,v), u, v].

While all 3D objects admit the model based on absolute coördinates [f(x,y,z), x, y, z], there are exactly two of them that accept models defined in terms of the [x,y] coördinates, as in [f(x,y), x, y]. These two are elevation_grid and explicit. Object points also accepts the model [f(k), k] is also valid, k being an ordering parameter. The names of the variables defined in the lists may be different from those used in the definitions of the graphics objects. However, if an expression is given to enhanced3d, its variables must be the same used in the definition of the surface.

When enhanced3d is set to something different from none, options color and surface_hide are ignored.

Note: In order to maintain backward compatibility, enhanced3d = false is defined to be enhanced3d = none, and enhanced3d = true is defined to be enhanced3d = [z, x, y, z].

Example: An elevation_grid object with coloring defined by the [f(x,y,z), x, y, z] model.

```
(%i1) load(draw)$
      draw3d(enhanced3d = [x-z/10, x,y,z],
        palette         = gray,
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3))$
```

Example: An elevation_grid object with coloring defined by the [f(x,y), x, y] model. THis example illustrates the fact that the names of the variables defined in the lists may be different from those used in the definitions of the graphics objects; r corresponds to x, and s to y.

```
(%i1) load(draw)$
      draw3d(enhanced3d = [sin(r*s), r,s],
        explicit(20*exp(-x^2-y^2)-10, x,-3,3, y,-3,3))$
```

Example: A parametric object with coloring defined by the [f(x,y,z), x, y, z] model.

```
(%i1) load(draw)$
      draw3d(nticks = 100,
        line_width  = 2,
        enhanced3d  = [if y>= 0 then 1 else 0, x,y,z],
        parametric(sin(u)^2,cos(u),u, u,0,4*%pi))$
```

Example: A parametric object with coloring defined by the [f(u), u] model. In this example, (u-1)^2 is a shortcut for [(u-1)^2,u].

```
(%i1) load(draw)$
      draw3d(nticks = 60,
        line_width  = 3,
        enhanced3d  = (u-1)^2,
        parametric(cos(5*u)^2,sin(7*u),u-2, u,0,2))$
```

Example: An elevation_grid object with coloring defined by the [f(x,y), x, y] model.

```
(%i1) load(draw)$
      m: apply(matrix,
               makelist(makelist(cos(i^2/80-k/30), k,1,30), i,1,20))$
      draw3d(xlabel = "x",
        ylabel      = "y",
        enhanced3d  = [cos(x*y*10), x,y],
        elevation_grid(m, -1,-1, 2,2))$
```

Example: A tube object with coloring defined by the [f(x,y,z), x, y, z] model.

```
(%i1) load(draw)$
      draw3d(enhanced3d = [cos(x-y), x,y,z],
        palette         = gray,
        xu_grid         = 50,
        tube(cos(a),a,0,1, a,0,4*%pi))$
```

Example: A tube object with coloring defined by the [f(u), u] model. Here, enhanced3d = -a is a shortcut for enhanced3d = [-foo,foo].

```
(%i1) load(draw)$
      draw3d(tube_extremes = [open, closed],
        palette            = [26,15,-2],
        enhanced3d         = -a,
        tube(a,a,a^2,1, a,-2, 2))$
```

Example: implicit and points objects with coloring defined by the [f(x,y,z), x, y, z] model.

```
(%i1) load(draw)$
      draw3d(enhanced3d = [x-y,x,y,z],
        implicit((x^2+y^2+z^2-1)*(x^2+(y-1.5)^2+z^2-0.5)-0.015,
            x,-1,1, y,-1.2,2.3, z,-1,1))$
      m: makelist([random(1.0),random(1.0),random(1.0)], k,1,2000)$
      draw3d(point_type = filled_circle,
        point_size      = 2,
        enhanced3d      = [u+v-w, u,v,w],
        points(m))$
```

Example: Use of the model [f(k), k] is also valid, k being an ordering parameter.

```
(%i1) load(draw)$
      m:makelist([random(1.0), random(1.0), random(1.0)], k,1,5)$
      draw3d(enhanced3d = [sin(j), j],
        point_size      = 3,
        point_type      = filled_circle,
        points_joined   = true,
        points(m))$
```

logcb **Global graphics option**

Default value: false

See also: enhanced3d, colorbox, cbrange

If logcb, enhanced3d, and colorbox are all true, the tics in the colorbox will be drawn in the logarithmic scale. When enhanced3d or colorbox is false, option logcb has no effect.

Example: A colorbox with tics drawn in the logarithmic scale.

```
(%i1) load(draw)$
      draw3d (enhanced3d = true,
        color            = green,
        logcb            = true,
        logz             = true,
        palette          = [-15,24,-9],
        explicit(exp(x^2-y^2), x,-2,2, y,-2,2))$
```

rot_horizontal **Global graphics option**

Default value: 30

Superceded by: view

See also: rot_vertical, view

rot_horizontal is the angle (in degrees) of horizontal rotation (around the z-axis) to set the view point in 3D scenes. The angle is must be in the interval [0, 180].

Example: A horizontally rotated plot.

```
(%i1) load(draw)$
      draw3d(rot_horizontal    = 45,
        explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
```

rot_vertical **Global graphics option**

Default value: 60

Superceded by: view

See also: rot_horizontal, view

rot_vertical is the angle (in degrees) of vertical rotation (around the x-axis) to set the view point in 3D scenes. The angle is must be in the interval [0, 180]. **Note:** rot_vertical is deprecated. Use view, instead.

Example: A vertically rotated plot.

```
(%i1) load(draw)$
      draw3d(rot_vertical = 170,
        explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
```

surface_hide **Global graphics option**

Default value: false

If surface_hide is true, 3D surfaces are plotted as though they were opaque, so that any part of a surface that is behind some part (from the viewer's perspective) is hidden.

Example: The effect of the surface_hide option.

```
(%i1) load(draw)$
      draw(columns        = 2,
        gr3d(explicit(exp(sin(x)+cos(x^2)), x,-3,3, y,-3,3)),
        gr3d(surface_hide = true,
          explicit(exp(sin(x)+cos(x^2)), x,-3,3, y,-3,3)))$
```

tube_extremes **Graphics option**

Default value: [open, open]

A list with two possible elements, either of which may be open or closed, indicating whether the extremes of a tube graphics object remain open or must be closed. By default, both extremes are left open.

Example: A tube with an open end and a closed end/

```
(%i1) load(draw)$
      draw3d(tube_extremes = [open, closed],
        tube(0,0,a,1, a,0,8))$
```

view **Global graphics option**

Default value: [60, 30]

Supercedes: rot_horizontal and rot_vertical

view is a pair of angles, measured in degrees, indicating the direction from which one views a 3D scene. The first angle is the vertical rotation around the x-axis, in the range [0, 180]. The second one is the horizontal rotation around the z-axis, in the range [0, 360].

Example: Setting the view angle.

```
(%i1) load(draw)$
      draw3d(view = [170, 360],
        explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
```

x_voxel **Graphics option**

Default value: 10

y_voxel **Graphics option**

Default value: 10

z_voxel **Graphics option**

Default value: 10

The following applies to y_voxel and z_voxel just as it does to x_voxel.

x_voxel is the number of voxels in the x-direction to be used by the 3D implicit graphics object.

xu_grid **Graphics option**

Default value: 30

yv_grid **Graphics option**

Default value: 30

Applies to: explicit and parametric_surface graphics objects in 3D

The following applies to yv_grid just as it does to xu_grid.

When draw plots explicit or parametric_surface graphics objects in 3D, it subdivides the xrange into a number of subintervals determined by xu_grid.

Example: An explicit object with a grid sparse in the x-direction but dense in the y-direction.

```
(%i1) load(draw)$
      draw3d(xu_grid = 10,
        yv_grid      = 50,
        explicit(x^2+y^2, x,-3,3, y,-3,3))$
```

Example: A parametric_surface object with a grid sparse in the x-direction but dense in the y-direction.

```
(%i1) load(draw)$
      draw3d(xu_grid = 10,
        yv_grid      = 50,
        parametric_surface(u,v,u^2 - v^2, u,-3,3, v,-3,3))$
```

xyplane **Global graphics option**

Default value: false

xyplane tells draw where to put the x, y-plane in 3D scenes. When xyplane is false, the x, y-plane is placed automatically; when it is a real number, the x, y-plane intersects the z-axis at this level.

Example: Relocating the x, y-plane.

```
(%i1) load(draw)$
      draw3d(xyplane = %e-2,
        explicit(x^2+y^2, x,-1,1, y,-1,1))$
```

4.5.4 Options for coördinate axes

axis_bottom, axis_left, axis_top, axis_right **Graphics option**

Default value: true

See also: axis_3d

If axis_bottom is true, the bottom axis is shown in 2D scenes. Likewise for axis_left, axis_top, and axis_right.

Example: A plot without the bottom axis.

```
(%i1) load(draw)$
      draw2d(axis_bottom = false,
        explicit(x^3, x,-1,1))$
```

logx **Global graphics option**

Default value: false

logy **Global graphics option**

Default value: false

logz **Global graphics option**

Default value: false

The following applies to logy and logz just as it does to logx.

If logx is true, the x-axis will be drawn in the logarithmic scale.

Example: A plot using the logarithmic scale on the horizontal axis.

```
(%i1) load(draw)$
      draw2d(logx = true,
        explicit(log(x), x,0.01,5))$
```

proportional_axes **Global graphics option**

Default value: none

When proportional_axes is equal to xy or xyz, a 2D or 3D (respectively) scene will be drawn with axes proportional to their relative lengths. The effect is to make the scales on the coördinate axes agree, thus "squaring up" the plot.

Notes: (1) proportional_axes is overridden by xrange, yrange, and zrange. (2) Option proportional_axes requires gnuplot version 4.2.6 or later.

Example: Compare these plots to see the effect of proportional_axes.

```
(%i1) load(draw)$
(%i2) draw2d(ellipse(0,0, 1,2, 0,360))$
(%i3) draw2d(proportional_axes = xy,
        ellipse(0,0, 1,2, 0,360))$
```

Example: proportional_axes in a multiplot.

```
(%i1) load(draw)$
(%i2) draw(terminal           = wxt,
        gr2d(proportional_axes = xy,
          explicit(x^2, x,0,1)),
        gr2d(explicit(x^2, x,0,1),
          xrange               = [0,1],
          yrange               = [0,2],
          proportional_axes=xy),
        gr2d(explicit(x^2, x,0,1)))$
```

xaxis **Global graphics option**

Default value: false

yaxis **Global graphics option**

Default value: false

zaxis **Global graphics option**

Default value: false

See also: xaxis_color, xaxis_type, xaxis_width

The following applies to yaxis and zaxis just as it does to xaxis.

If xaxis is true, the x-axis is drawn.

Example: A plot with the x-axis included.

```
(%i1) load(draw)$
      draw2d(xaxis  = true,
        xaxis_color = blue,
        explicit(x^3, x,-1,1))$
```

Example: A plot with the y-axis included.

```
(%i1) load(draw)$
      draw2d(yaxis  = true,
        yaxis_color = blue,
        explicit(x^3, x,-1,1))$
```

Example: A plot with the z-axis included.

```
(%i1) load(draw)$
      draw3d(zaxis  = true,
        zaxis_type  = solid,
        zaxis_color = blue.
        explicit(x^2+y^2, x,-1,1, y,-1,1))$
```

xaxis_color **Global graphics option**

Default value: black

yaxis_color **Global graphics option**

Default value: black

zaxis_color **Global graphics option**

Default value: black

See also: color, xaxis, xaxis_type, xaxis_width

The following applies to yaxis_color and zaxis_color just as it does to xaxis_color.

xaxis_color specifies the color for the x-axis. See color to know how colors are defined.

Example: A plot with a blue x-axis.

```
(%i1) load(draw)$
      draw2d(xaxis  = true,
        xaxis_color = blue,
        explicit(x^3, x,-1,1))$
```

Example: A plot with a blue y-axis.

```
(%i1) load(draw)$
      draw2d(yaxis  = true,
        yaxis_color = blue,
        explicit(x^3, x,-1,1))$
```

Example: A plot with a blue z-axis.

```
(%i1) load(draw)$
      draw3d(zaxis  = true,
        zaxis_type  = solid,
        zaxis_color = blue.
        explicit(x^2+y^2, x,-1,1, y,-1,1))$
```

xaxis_secondary **Graphics option**

Default value: false

yaxis_secondary **Graphics option**

Default value: false

Applies to: 2D plots

See also: xrange_secondary, xtics_rotate_secondary, xtics_secondary, xtics_secondary_axis, yrange_secondary, ytics_rotate_secondary, ytics_secondary, ytics_secondary_axis

The following applies to xaxis_secondary just as it does to yaxis_secondary.

If xaxis_secondary is true, function values can be plotted with respect to a second x-axis, which will be drawn on top of the scene, and likewise for yaxis_secondary.

Example: Using the secondary x-axis.

```
(%i1) load(draw)$
      draw2d(key         = "x+1",
        explicit(x+1, x,1,2),
        xtics_secondary = true,
        xaxis_secondary = true,
        key             = "x^2",
        color           = red,
        explicit(x^2, x,-1,1))$
```

Example: Using the secondary y-axis.

```
(%i1) load(draw)$
      draw2d(key         = "sin(x)",
        explicit(sin(x),x,0,10),
        yaxis_secondary = true,
        ytics_secondary = true,
        color           = black,
        key             = "100*sin(x+0.1)+2",
        explicit(100*sin(x+0.1)+2, x,0,10))$
```

xaxis_type **Global graphics option**

Default value: dots

yaxis_type **Global graphics option**

Default value: dots

zaxis_type **Global graphics option**

Default value: dots

See also: xaxis, xaxis_color, xaxis_width

The following applies to yaxis_type and zaxis_type just as it does to xaxis_type.

xaxis_type indicates how the x-axis is displayed; possible values are solid and dots.

Example: A solid x-axis.

```
(%i1) load(draw)$
      draw2d(explicit(x^3, x,-1,1),
        xaxis      = true,
        xaxis_type = solid)$
```

Example: A solid y-axis.

```
(%i1) load(draw)$
      draw2d(explicit(x^3, x,-1,1),
        yaxis      = true,
        yaxis_type = solid)$
```

Example: A solid z-axis.

```
(%i1) load(draw)$
      draw3d(explicit(x^2+y^2, x,-1,1, y,-1,1),
        zaxis      = true,
        zaxis_type = solid)$
```

xaxis_width **Global graphics option**

Default value: 1

yaxis_width **Global graphics option**

Default value: 1

zaxis_width **Global graphics option**

Default value: 1

See also: xaxis, xaxis_color, xaxis_type

The following applies to yaxis_width and zaxis_width just as it does to xaxis_width.

xaxis_width is the width of the x-axis. Its value must be a positive number.

Example: Changing the width of the x-axis.

```
(%i1) load(draw)$
      draw2d(explicit(x^3, x,-1,1),
        xaxis       = true,
        xaxis_width = 3)$
```

Example: Changing the width of the y-axis.

```
(%i1) load(draw)$
      draw2d(explicit(x^3, x,-1,1),
        yaxis       = true,
        yaxis_width = 3)$
```

Example: Changing the width of the z-axis.

```
(%i1) load(draw)$
      draw3d(explicit(x^2+y^2, x,-1,1, y,-1,1),
        zaxis       = true,
        zaxis_width = 3)$
```

xlabel **Global graphics option**

Default value: ""

ylabel **Global graphics option**

Default value: ""

zlabel **Global graphics option**

Default value: ""

The following applies to ylabel and zlabel just as it does to xlabel.

Option xlabel, a string, is the label for the x-axis. Because its default value is the empty string. The default is that no label is written.

Example: Labeling the x- and y-axes.

```
(%i1) load(draw)$
      draw2d(xlabel = "Time",
        ylabel      = "Population",
        explicit(exp(u), u,-2,2))$
```

xrange_secondary **Global graphics option**

Default value: auto

yrange_secondary **Global graphics option**

Default value: auto

The following applies to xrange_secondary just as it does to yrange_secondary.

If xaxis_secondary is true and xrange_secondary is auto, the range for the second x-axis is computed automatically. If the user wants a specific interval for the second x-axis, it must be given as a wxMaxima list, as in xrange_secondary = [-2, 3].

Example: Setting the secondary yrange.

```
(%i1) load(draw)$
      draw2d(
        explicit(sin(x), x,0,10),
        yaxis_secondary  = true,
        ytics_secondary  = true,
        yrange           = [-3, 3],
        yrange_secondary = [-20, 20],
        color            = blue,
        explicit(100*sin(x+0.1)+2, x,0,10))$
```

xtics **Global graphics option**

Default value: auto

ytics **Global graphics option**

Default value: auto

ztics **Global graphics option**

Default value: auto

The following applies to ytics and ztics just as it does to xtics.

This graphics option controls the way tic marks are drawn on the x-axis, y-axis, or z-axis, respectively, according to these rules:

- When option xtics is bound to symbol auto, tic marks are drawn automatically.
- When option xtics is bound to symbol none, tic marks are not drawn.
- When option xtics is bound to a positive number, this is the distance between two consecutive tic marks.
- When option xtics is bound to a list of length three of the form [start, incr, end], tic marks are plotted from start to end at intervals of length incr.
- When option xtics is bound to a set of numbers of the form n_1, n_2,..., tic marks are plotted at values n_1, n_2,...
- When option xtics is bound to a set of pairs of the form ["label_1", n_1], ["label_2", n_2],..., tic marks at locations n_1, n_2,... are labeled with "label_1", "label_2",... respectively.

Example: Disabling tic marks.

```
(%i1) load(draw)$
      draw2d(xtics = 'none,
        explicit(x^3, x,-1,1))$
```

Example: Tic marks every 1/4 unit.

```
(%i1) load(draw)$
      draw2d(xtics = 1/4,
        explicit(x^3, x,-1,1))$
```

Example: Tic marks from -3/4 to 3/4 in steps of 1/8.

```
(%i1) load(draw)$
      draw2d(xtics = [-3/4,1/8,3/4],
        explicit(x^3, x,-1,1))$
```

Example: Tic marks at points -1/2, -1/4 and 3/4.

```
(%i1) load(draw)$
      draw2d(xtics = {-1/2,-1/4,3/4},
        explicit(x^3, x,-1,1))$
```

Example: Labeled tic marks.

```
(%i1) load(draw)$
      draw2d(xtics = {["High",0.75], ["Medium",0], ["Low",-0.75]},
        explicit(x^3, x,-1,1))$
```

xtics_axis **Global graphics option**

Default value: false

ytics_axis **Global graphics option**

Default value: false

ztics_axis **Global graphics option**

Default value: false

The following applies to ytics_axis and ztics_axis just as it does to xtics_axis.

If xtics_axis is true, tic marks and their labels are plotted along the x-axis. Otherwise, tics are plotted on the border.

xtics_rotate **Global graphics option**

Default value: auto

ytics_rotate **Global graphics option**

Default value: auto

ztics_rotate **Global graphics option**

Default value: auto

The following applies to ytics_rotate and ztics_rotate just as it does to xtics_rotate.

If tics_rotate is true, tic marks on the x-axis are rotated 90 degrees.

xtics_rotate_secondary **Global graphics option**

Default value: false

ytics_rotate_secondary **Global graphics option**

Default value: false

The following applies to xtics_rotate_secondary just as it does to ytics_rotate_secondary.

If xtics_rotate_secondary is true, tic marks on the secondary x-axis are rotated 90 degrees.

xtics_secondary **Graphics option**

Default value: auto

ytics_secondary **Graphics option**

Default value: auto

The following applies to xtics_secondary just as it does to ytics_secondary.

This graphic option controls the way tic marks are drawn on the secondary x-axis. See xtics for a complete description.

xtics_secondary_axis **Global graphics option**

Default value: false

ytics_secondary_axis **Global graphics option**

Default value: false

The following applies to xtics_secondary_axis just as it does to ytics_secondary_axis.

If xtics_secondary_axis is true, tic marks and their labels are plotted along the secondary x-axis. It xtics_secondary_axis is false, tic marks are plotted on the border.

4.5.5 The terminal and its options

The terminal isn't so much an object as an option. Even so, it helps to think of the terminal as a "thing." terminal is the option that tells draw and related commands what type of device or file format to use when creating scenes and plotting them. screen is the default value of terminal. It tells draw to put the graphic it creates on the screen in a gnuplot window. Other values of terminal tell draw to put the graphic in a file having a given format. For example, terminal = 'eps tells draw to put the graphic in a file in the encapsulated PostScript™ format, with extension ".eps".

dimensions **Global graphics option**

Default value: [600,500]

Can be used as an argument of function draw.

Supercedes: eps_height, eps_width, pdf_height, and pdf_width

dimensions sets the dimensions of the output terminal. Its value is a list of the form [width, height]. The meaning of the numbers width and height depends on the terminal you are working with:

- The terminals animated_gif, aquaterm, gif, jpg, png, screen, svg, and wxt all expect width and height to be positive integers representing the number of points in each direction. If width and height are not integers, they are rounded.
- For terminals eps, eps_color, pdf, and pdfcairo, both numbers represent hundredths of cm, which means that, by default, pictures in these formats are 6cmin width and 5cmin height.

Example: Option dimensions applied to file output and to the wxt terminal.

```
(%i1) load(draw)$
      draw2d(dimensions = [300,300],
        terminal        = 'png,
        explicit(x^4, x,-1,1))$
      draw2d(dimensions = [300,300],
        terminal        = 'wxt,
        explicit(x^4, x,-1,1))$
```

Example: Option dimensions applied to the eps terminal to produce an eps file with A4 portrait dimensions.

```
(%i1) load(draw)$
      A4portrait: 100*[21, 29.7]$
      draw3d(dimensions = A4portrait,
        terminal        = 'eps,
        explicit(x^2-y^2, x,-2,2, y,-2,2))$
```

eps_height **Global graphics option**

Default value: 8

Superceded by: dimensions

Can be used as an argument of function draw.

See also: dimensions, eps_width, file_name, terminal

This is the height (in cm) of the Postscript file generated by terminals eps and eps_color.

Example: Setting the height and width of the eps terminal.

```
(%i1) load(draw)$
      draw2d(terminal = 'eps,
        eps_width     = 3,
        eps_height    = 3,
        explicit(x^2, x,-1,1))$
```

eps_width **Global graphics option**

Default value: 12

Superceded by: dimensions

Can be used as an argument of function draw.

See also: eps_height, file_name, terminal

This is the width (in cm) of the Postscript file generated by terminals eps and eps_color. See the example under eps_height.

file_bgcolor **Graphics option**

Default value: "#ffffff"

Sets the background color for terminals png, jpg and gif. Colors must be given in hexadecimal RGB code. Color names are not allowed. The default background color "#ffffff" is white.

pdf_height **Global graphics option**

Default value: 29.7

Superceded by: dimensions

Can be used as an argument of function draw.

See also: file_name, pdf_width, terminal

This is the height (in cm) of the PDF document generated by terminals pdf and pdfcairo. The default height of 29.7cm is the standard height for A4 paper, in the portrait orientation.

Example: Setting the height and width of the pdf terminal.

```
(%i1) load(draw)$
      draw2d(terminal = 'pdf,
        pdf_width     = 3.0,
        pdf_height    = 3.0,
        explicit(x^2, x,-1,1))$
```

pdf_width **Global graphics option**

Default value: 21.0

Superceded by: dimensions

Can be used as an argument of function draw.

See also: file_name, pdf_height, terminal

This is the width (measured in cm) of the PDF document generated by terminals pdf and pdfcairo. The default width of 21.0cm is the standard width of A4 paper, in the portrait orientation. See the example under pdf-height.

pic_height **Global graphics option**

Default value: 640

Can be used as an argument of function draw.

See also: file_name, pic_width, terminal

This is the height of the bitmap file generated by terminals png and jpg.

Example: Setting the height and width of the png terminal.

```
(%i1) load(draw)$
      draw2d(terminal = 'png,
        pic_width     = 300,
        pic_height    = 300,
        explicit(x^2, x,-1,1))$
```

pic_width **Global graphics option**

Default value: 640

Can be used as an argument of function draw.

See also: file_name, pic_height, terminal

This is the width of the bitmap file generated by terminals png and jpg. See the example under pic_height.

terminal **Global graphics option**

Default value: screen

Can be used as an argument of function draw.

See also: delay, dimensions, file_name

Selects the terminal to be used by gnuplot. The possible values are animated_gif, aquaterm, eps, eps_color, gif, jpg, pdf, pdfcairo, png, pngcairo, screen, svg, and wxt. In older versions of wxMaxima, x11 is also an option.

Notes:

1. Terminals aquaterm, screen, and wxt can be also defined as lists with two elements: (1) The name of the terminal itself, and (2) A non negative whole number. In this form, multiple windows can be opened at the same time, each with its corresponding number. **This feature does not work in Windows™ platforms**.
2. Terminal aquaterm is used by Macintosh™ computers.
3. Terminal pdf requires gnuplot to be compiled with the option --enable-pdf, and libpdf must be installed before compiling.[6] The pdf library is available from http://www.pdflib.com/en/download/pdflib-family/pdflib-lite/.
4. Terminal pdfcairo requires gnuplot version 4.3 or newer.
5. Terminal wxt requires wxWidgets.[7]
6. Terminal x11 is used on Unix-like machines.

Example: Various terminal types.

```
(%i1) load(draw)$
      /* screen terminal (default) */
      draw2d(explicit(x^2,x,-1,1))$
      /* png file */
      draw2d(terminal = 'png,      /*Note the tic mark.*/
        pic_width     = 300,
        explicit(x^2, x,-1,1))$
      /* jpg file */
      draw2d(terminal = 'jpg,      /*Note the tic mark.*/
        pic_width     = 300,
```

[6] As of 14 November 2011, gnuplot is in version 4.4.3.

[7] wxMaxima comes with wxWidgets, other implementations of Maxima might not.

```
  pic_height    = 300,
  explicit(x^2, x,-1,1))$
/* eps file */
draw2d(terminal = 'eps,      /*Note the tic mark.*/
  file_name     = "myfile",
  explicit(x^2, x,-1,1))$
/* wxwidgets window */
draw2d(terminal = 'wxt,       /*Note the tic mark.*/
  explicit(x^2, x,-1,1))$
```

Example: A pdf terminal. (This is the syntax given in the wxMaxima help files. For some reason, the pdf file it produces is a 0-byte file. I haven't had a chance to fix this yet.)

```
(%i1) load(draw)$
      /* pdf file */
      draw2d(terminal = 'pdf,      /*Note the tic mark.*/
      file_name       = "mypdf",
      dimensions      = 100*[12.0,8.0],
      explicit(x^2, x,-1,1))$
```

Example: Terminals in multiple screens.

```
(%i1) load(draw)$
      draw2d(explicit(x^5, x,-2,2), terminal=[screen, 3])$
      draw2d(explicit(x^2, x,-2,2), terminal=[screen, 0])$
```

Example: An animated gif file.

```
(%i1) load(draw)$
      draw(delay  = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2, x,-1,1)),
        gr2d(explicit(x^3, x,-1,1)),
        gr2d(explicit(x^4, x,-1,1)));
```

4.5.6 Options for the graphics window

`columns` **Global graphics option**

Default value: `1`

Can be used as an argument of function `draw`.

When `draw` plots more than one scene, `columns` is the number of columns in which it arranges the scenes, in the graphics window.

Example: Two columns in the graphics window.

```
(%i1) load(draw)$
      scene1: gr2d(title = "Ellipse",
                nticks   = 30,
                parametric(2*cos(t),5*sin(t), t,0,2*%pi))$
      scene2: gr2d(title ="Triangle",
                polygon([4,5,7], [6,4,2]))$
      draw(scene1, scene2, columns = 2)$
```

font **Global graphics option**

Default value: ""

See also: font_size, terminal

This option can be used to set the font face to be used by the terminal. Only one font face and size can be used throughout the plot. The default value of font is the empty string, which tells draw to use the default font.

gnuplot does not handle fonts by itself. It delegates this task to the support libraries of the different terminals, and each one has its own way of handling fonts. A brief summary follows.

aqua: Default is "Times-Roman"

gif (also jpeg, png): The gif, jpeg and png terminals use the libgd library, which uses the font path stored in the environment variable GDFONTPATH; in this case, it is only necessary to set option font to the font's name.

Example: Setting the font face by using the font name.

```
(%i1) load(draw)$
      draw2d(terminal = 'gif,
        font          = "FreeSerifBoldItalic",
        font_size     = 20,
        color         = red,
        label(["FreeSerifBoldItalic font, size 20", 1,1]))$
```

Example: It is also possible to give the absolute pathname of the font file. (This example uses a Linux pathname.)

```
(%i1) load(draw)$
      path: "/usr/share/fonts/truetype/freefont/" $
      file: "FreeSerifBoldItalic.ttf" $
      draw2d(terminal = 'gif,
        font          = concat(path, file),
        font_size     = 20,
        color         = red,
        label(["FreeSerifBoldItalic font, size 20", 1,1]))$
```

jpeg: See gif.

pdf: See Postscript.

pdfcairo: See wxt.

png: See gif.

Postscript (also pdf): You can set the font face for Postscript output by using the font name. The standard Postscript fonts are: "Times-Roman", "Times-Italic", "Times-Bold", "Times-BoldItalic", "Helvetica", "Helvetica-Oblique", "Helvetica-Bold", "Helvetic-BoldOblique", "Courier", "Courier-Oblique", "Courier-Bold", and "Courier-BoldOblique".

Example: Setting a Postscript font face.

```
(%i1) load(draw)$
      draw2d(terminal = 'eps,
        font          = "Courier-Oblique",
        font_size     = 15,
        label(["Courier-Oblique font, size 15", 1,1]))$
```

windows: The windows terminal does not support changing of fonts from inside the plot. Instead, once the plot has been generated, you can change the font by right-clicking on the menu

of the graph window.

wxt (also pdfcairo): This terminal uses the pango library, which finds fonts via the fontconfig utility.

x11: This terminal uses the normal x11 font server mechanism.

Example: Setting the font in the x11 terminal.

```
(%i1) load(draw)$
      draw2d(font = "Arial",
        font_size = 20,
        label(["Arial font, size 20", 1,1]))$
```

Note: See the gnuplot documentation for more information about terminals and fonts.

font_size — Global graphics option

Default value: 10

See also: font

Option font_size sets the size of the font to be used by the terminal. Only one font face and size can be used throughout the plot. font_size is active only when option font is not equal to the empty string. **Note:** The windows terminal does not support setting the font size inside the draw command or its variants. To change the font, right-click on the menu in the graphics window, after the plot has been made.

grid — Global graphics option

Default value: false

If grid is true, a grid will be drawn on the x, y-plane.

Example: A plot with a grid.

```
(%i1) load(draw)$
      draw2d(grid = true,
        explicit(exp(u), u,-2,2))$
```

Example: A grid in a polar plot.

```
(%i1) load(draw)$
      draw2d(grid = true,
        nticks    = 200
        polar(3-2*cos(%theta), %theta,0,2*%pi))$
```

key — Graphics option

Default value: ""

Applies to: In 2D ellipse, explicit, implicit, parametric, points, polar, polygon, rectangle, vector; in 3D explicit, parametric, parametric_surface, points

If key is not the empty string for at least one graphics object in the plot, then a legend is created for the graphics window, and each graphics object that has a key is listed in the legend. key is empty by default, for every graphics object in the graphics window, so that the default is that no legend is created.[8]

Example: Two key statements, two keys in the legend.

[8] The wxMaxima help files go on to say, "Moreover, unlike e.g., the color option, the value of key applies only to the first graphics object that appears after the key = statement" and invite the reader to "[C]ompare the legends in the gnuplot windows of the two following examples". However, on my Linux machine, in the second example, both curves are labeled "Sinus" in the legend, which seems contrary to the wording of the statement just quoted.

```
(%i1) load(draw)$
      draw2d(key = "Sinus",
        explicit(sin(x), x,0,10),
        color    = red,
        key      = "Cosinus",
        explicit(cos(x), x,0,10))$
```

Example: One key statement, one key in the legend.

```
(%i1) load(draw)$
      draw2d(explicit(sin(x), x,0,10),
        key = "Cosinus",
        color    = red,
        explicit(cos(x), x,0,10))$
```

Example: Be careful: If the key statement appears before the first function, that key is applied to both functions:

```
(%i1) load(draw)$
      draw2d(key = "Sinus",
        explicit(sin(x), x,0,10),
        color    = red,
        explicit(cos(x), x,0,10))$
```

title **Global graphics option**

Default value: ""

Option title is a string, and must be punctuated with quotation marks. It's the main title for the scene. Its default value is the empty string, so by default, no title is written.

Example: Adding a title to a plot.

```
(%i1) load(draw)$
      draw2d(title = "Exponential function",
        explicit(exp(u), u,-2,2))$
```

xrange **Global graphics option**

Default value: auto

yrange **Global graphics option**

Default value: auto

zrange **Global graphics option**

Default value: auto

The following applies to yrange and zrange just as it does to xrange.

If xrange is auto, the range for the x-coördinate is computed automatically. If the user wants a specific interval for x, it must be given as a wxMaxima list, as in xrange=[-2, 3].

Example: Three plots of the same function, with different xrange and yrange settings. The objects are plotted over the interval [-1,1] for comparison.

```
(%i1) load(draw)$
      draw2d(xrange = [-3,5],
        explicit(x^2, x,-1,1))$
```

```
(%i1) load(draw)$
      draw2d(xrange = [-3,3], yrange = [-2,3],
        explicit(x^2, x,-1,1))$
```

```
(%i1) load(draw)$
      draw3d(xrange = [-3,3], yrange = [-3,3], zrange = [-2,5],
        explicit(x^2+y^2, x,-1,1, y,-1,1))$
```

4.5.7 Handling multiple options

`set_draw_defaults(graphics option,...,graphics object,...)` **Function**

`set_draw_defaults` sets the default options according to the user's choices. This function is useful for plotting a sequence of graphics with common graphics options. To restore the default options to wxMaxima's preset values, call this function without arguments.

Example: Changing the default graphics options and resetting them to wxMaxima's defaults.

```
(%i1) load(draw)$
      set_draw_defaults(xrange = [-10,10],
        yrange                 = [-2, 2],
        color                  = blue,
        grid                   = true)$
      /* plot with user defaults */
      draw2d(explicit(((1+x)**2/(1+x*x))-1, x,-10,10))$
      set_draw_defaults()$
(%i5) /* plot with standard defaults */
      draw2d(explicit(((1+x)**2/(1+x*x))-1, x,-10,10))$
```

Note: Function `set_draw_defaults` only changes the default graphics options for the duration of the current session. When you begin a new session, wxMaxima will revert the defaults to their pre-set values.

`user_preamble` **Global graphics option**

Default value: `""`

This option is used to fine-tune gnuplot's behavior by writing settings to be sent before the `draw` command in which it appears. In some cases, certain options can only be set by using the `user_preamble`. The value of this option must be a string or a list of strings (one string per line).

Example: : The `dumb terminal` is not supported by package `draw`, but it is possible to set it by making use of option `user_preamble`.

```
(%i1) load(draw)$
      draw2d(user_preamble = "set terminal dumb",
        explicit(exp(x)-1, x,-1,1),
        parametric(cos(u),sin(u), u,0,2*%pi))$
```

4.6 Additional topics

4.6.1 Multiple plots

Example: An animated gif file. **See also:** `delay`, `file_name`, `pic_height`, `pic_width`

```
(%i1) load(draw)$
      draw(delay  = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2, x,-1,1)),
        gr2d(explicit(x^3, x,-1,1)),
```

```
  gr2d(explicit(x^4, x,-1,1))$
```

Example: Multiple graphics windows in a single gnuplot window.

```
(%i1) load(draw)$
      draw(gr2d(explicit(x^2, x,0,1)),
        gr2d(xrange = [0,1],
          yrange    = [0,2],
          explicit(x^2, x,0,1)),
        gr2d(explicit(x^2, x,0,1)));
```

Example: Simultaneous multiple gnuplot windows.

```
(%i1) load(draw)$
      draw2d(explicit(x^5, x,-2,2),
        terminal = [screen, 3])$
      draw2d(explicit(x^2, x,-2,2),
        terminal = [screen, 0])$
```

`multiplot_mode(term)` **Function**

When `multiplot-mode` is enabled, each call to `draw` sends a new plot to the same window (in `terminal` `term`), without erasing the previous ones.[9] Accepted arguments for this function are `aquaterm`, `none`, `screen`, and `wxt`. To disable the `multiplot mode`, use the command `multiplot_mode(none)`.

Notes: (1) When multiplot mode is enabled, global option `terminal` is blocked. You have to disable this working mode before changing to another `terminal`. (2) **This function does not work in Windows™ platforms**.

Example: Multiplot of some power functions.

```
(%i1) load(draw)$
      set_draw_defaults(xrange = [-1,1],
        yrange                 = [-1,1],
        grid                   = true,
        title                  = "Step by step plot")$
      multiplot_mode(screen)$
      draw2d(color = blue, explicit(x^2, x,-1,1))$
      draw2d(color = red, explicit(x^3, x,-1,1))$
      draw2d(color = brown, explicit(x^4, x,-1,1))$
      multiplot_mode(none)$
```

`delay` **Global graphics option**

Default value: `5`

This option affects only `animated gif images`.

Can be used as an argument of function `draw`.

See also: `pic_height`, `pic_width`, `terminal`

Option `delay` is the delay (in hundredths of seconds) between frames in animated gif files. It therefore controls the speed of animated gif's.

Example: Delay set to one second.

[9] If memory serves, in older versions, the user had to interact with wxMaxima to make each plot appear (after the first one). Version 5.24.0 does not seem to require user interaction, at least, not on my system.

```
(%i1) load(draw)$
      draw(delay  = 100,
        file_name = "zzz",
        terminal  = 'animated_gif,
        gr2d(explicit(x^2, x,-1,1)),
        gr2d(explicit(x^3, x,-1,1)),
        gr2d(explicit(x^4, x,-1,1)));
```

4.6.2 File handling: input, output, and exporting graphics made by `draw`

wxMaxima can read files for use by `draw`, and can write files that `draw` creates. You can use this ability to export graphics for use in other programs or documents. See `terminal` for more information.

`data_file_name` **Global graphics option**

Default value: `"data.gnuplot"`

Can be used as an argument of function `draw`.

This is the name of the file where `draw` puts the numeric data needed by gnuplot to build the requested plot. `draw` assumes that the value of `data_file_name` is the pathname of your file, given relative to the current working directory, unless an absolute pathname is used.

See the example under `gnuplot_file_name`.

`draw_file(graphics option,...,graphics object,...)` **Function**

Options: `eps_height`, `eps_width`, `file_name`, `file_bgcolor`, `terminal`, `pic_height`, `pic_width`

Saves the current plot into a file, called `maxima_out.EXT`, where `EXT` is the default extension for the terminal type. On *nix systems, `maxima_out.EXT` is saved in the user's home folder.

Example: Saving a graphic in an .eps file.

```
(%i1) load(draw)$
      /* screen plot */
      draw(gr3d(explicit(x^2+y^2, x,-1,1, y,-1,1)))$
      /* same plot in eps format */
      draw_file(terminal = eps,
        eps_width        = 5,
        eps_height       = 5)$
```

`file_name` **Global graphics option**

Default value: `"maxima_out"`

Can be used as an argument of function `draw`.

See also: `pic_height`, `pic_width`, `terminal`

This is the name of the file where `terminal`s `png`, `jpg`, `eps`, `eps_color`, `pdf` and `pdfcairo` will save the graphic. `draw` assumes that the value of `file_name` is the path name of your file, given relative to the current working directory, unless an absolute pathname is used.

Example: Saving a graphic in a .png file.

```
(%i1) load(draw)$
      draw2d(file_name = "myfile",
        terminal       = 'png,
        explicit(x^2, x,-1,1))$
```

gnuplot_file_name **Global graphics option**

Default value: "maxout.gnuplot"

Can be used as an argument of function draw.

See also: data_file_name

This is the name of the file with the necessary commands to be processed by gnuplot. draw assumes that the value of gnuplot_file_name is the path name of your file, given relative to the current working directory, unless an absolute pathname is used.

Example: Naming the data and gnuplot command files.

```
(%i1) load(draw)$
      draw2d(file_name    = "my_file",
        gnuplot_file_name = "my_commands_for_gnuplot",
        data_file_name    = "my_data_for_gnuplot",
        terminal          = 'png,
        explicit(x^2, x,-1,1))$
```

xy_file **Graphics option**

Default value: ""

After draw plots its scenes, you can save the plot coördinates to a file, by selecting the plot with the mouse and hitting the x key. xy_file is the name of the file where the plot coördinates are saved. Its default value is the empty string, so by default, no coördinates are saved. draw assumes that the value of data_file_name is the path name of your file, given relative to the current working directory, unless an absolute pathname is used.

Chapter 5

The drawdf package

(Note to programmers and system administrators: drawdf is built upon the draw package, which requires gnuplot 4.2.)

drawdf is a third-party package for wxMaxima that draws direction fields of systems of two first-order ordinary differential equations (ODE's). It can also plot the direction field of a single first-order ODE. The main function provided by the drawdf package is the drawdf function and its wxMaxima variant wxdrawdf. Since this is an additional package, in order to use it you must first load it with load(drawdf).

To plot the direction field of a single ODE, the ODE must be written in the form $\frac{dy}{dx} = F(x, y)$, and the function F is used as an argument of drawdf. If the independent and dependent variables are not x, and y, as in the equation above, then those two variables must be named explicitly in a list given as an argument to the drawdf command. (See the examples.)

To plot the direction field of a system of two autonomous ODE's, they must be written in the form $\frac{dx}{dt} = F(x, y)$, $\frac{dy}{dt} = G(x, y)$, and the list [F(x,y), G(x,y)] must be passed to drawdf as an argument. Note that the first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. Those two variables do not have to be x and y, but if they are not, then the second argument given to drawdf must be another list naming the two variables, first the one on the horizontal axis and then the one on the vertical axis. (See the examples.)

If only one ODE is given, drawdf will assume $x = t$ and $F(x, y) = 1$, transforming the non-autonomous equation into a system of two autonomous equations.

5.1 The drawdf and wxdrawdf functions

drawdf(dy/dx, options_and_objects) **Function**
drawdf(dv/du, [u,v], options_and_objects) **Function**
drawdf(dv/du, [u,umin,umax], [v,vmin,vmax], options_and_objects) **Function**
drawdf([dx/dt,dy/dt], options_and_objects) **Function**
drawdf([du/dt,dv/dt], [u,v], options_and_objects) **Function**
drawdf([du/dt,dv/dt], [u,umin,umax], [v,vmin,vmax] options_and_objects) **Function**

See also: draw, draw2d

drawdf draws a 2D direction field with optional solution curves and other graphics using the draw package. The first argument specifies the derivative(s), and must be either an expression or a

list of two expressions. dy/dx, dx/dt and dy/dt are expressions that depend on x and y. dv/du, du/dt and dv/dt are expressions that depend on u and v. If the independent and dependent variables are not x and y, then their names must be specified immediately following the derivative(s), either as a list of two names [u,v], or as two lists of the form [u,umin,umax] and [v,vmin,vmax].

The remaining arguments are graphics options, graphics objects, or lists containing graphics options and objects, nested to arbitrary depth. All graphics objects and options supported by draw2d and gr_2d from the draw package are also supported by drawdf. Additional graphics objects and options are supported by drawdf; these are given below.

The arguments are interpreted sequentially: Graphics options affect graphics objects that follow them (but global graphics options affect all the relevant graphics objects). Graphics objects are drawn on the canvas in order specified. Any object will hide any previously drawn objects that it overlaps.

drawdf passes all unrecognized parameters to draw2d or gr_2d, allowing you to combine the full power of the draw package with drawdf. Also, drawdf accepts nested lists of graphics options and objects, allowing convenient use of makelist and other function calls to generate graphics.

Example: A basic direction field using a single ODE.

```
(%i1) load(drawdf)$
      drawdf(exp(-x)+y)$
```

Example: The same direction field, set up as an autonomous system.

```
(%i1) load(drawdf)$
      drawdf(exp(-t)+y, [t,y])$
```

Example: Another autonomous system, but with the x- and y-ranges altered.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5], [x,1,5], [y,-2,2])$
```

Example: A direction field with keyed and colored solution curves and a parametric graphics object.

```
(%i1) load(drawdf)$
      drawdf(x^2+y^2, [x,-2,2], [y,-2,2],
        field_color = gray,
        key         = "soln 1",
        color       = black,
        soln_at(0,0),
        key         ="soln 2",
        color       = red,
        soln_at(0,1),
        key         = "isocline",
        color       = green,
        line_width  = 2,
        nticks      = 100,
        parametric(cos(t),sin(t), t,0,2*%pi))$
```

Example: Using makelist to plot several solution curves.

```
(%i1) load(drawdf)$
      colors : ['red,'blue,'purple,'orange, 'green]$
```

```
drawdf([x-x*y/2, (x*y - 3*y)/4], [x,2.5,3.5], [y,1.5,2.5],
  field_color   = gray,
  makelist([key = concat("soln",k),
    color       = colors[k],
    soln_at(3, 2 + k/20)],
    k,1,5))$
```

wxdrawdf(dy/dx, options_and_objects) **Function**
wxdrawdf(dv/du, [u,v], options_and_objects) **Function**
wxdrawdf(dv/du, [u,umin,umax], [v,vmin,vmax], options_and_objects) **Function**
wxdrawdf([dx/dt,dy/dt], options_and_objects) **Function**
wxdrawdf([du/dt,dv/dt], [u,v], options_and_objects) **Function**
wxdrawdf([du/dt,dv/dt], [u,umin,umax], [v,vmin,vmax], options_and_objects) **Function**

See also: drawdf

wxdrawdf behaves exactly like drawdf, except it draws its plots in the wxMaxima notebook you're working in, instead of drawing in the gnuplot window.

5.2 Graphics objects for drawdf and wxdrawdf

drafwdf and wxdrawdf support all of the graphics objects used by draw and its variants. They also support the graphics objects given in this section.

point_at(x,y) **Graphics object**
points_at([x_1,y_1], [x_2,y_2],..., [x_n,y_n]) **Graphics object**

point_at draws the point (x, y). Note that the point is not given in list format. points_at draws points having coördinates [x_1,y_1], [x_2,y_2],..., [x_n,y_n]. Note that the points are lists.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        tstep        = 0.05,
        soln_arrows  = true,
        point_size   = 1.5,
        point_at(0,0),
        points_at([2*%pi,0], [-2*%pi,0]))$
```

saddle_at(x,y) **Graphics object**
saddles_at([x_1,y_1], [x_2,y_2],..., [x_n,y_n]) **Graphics object**

saddle_at and saddles_at attempt to automatically linearize the equation at each saddle and to plot a numerical solution corresponding to each eigenvector, including the separatrices. Note that saddles_at requires the points to be given as lists [x_k,y_k], but saddle_at does not.

Example: The direction field for a model of a damped pendulum, with one saddle showing. Note that small time steps are used to keep the x- and y-steps small.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        tstep        = 0.05,
        soln_arrows  = true,
```

```
  field_degree = 'solns,
  saddle_at(%pi,0))$
```

Example: The same model, with two saddles plotted.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        tstep        = 0.05,
        soln_arrows  = true,
        point_size   = 0.5,
        points_at([0,0], [2*%pi,0], [-2*%pi,0]),
        field_degree = 'solns,
        saddles_at([%pi,0], [-%pi,0]))$
```

soln_at **Graphics object**
solns_at **Graphics object**

See also: trajectory_at

soln_at and solns_at draw solution curves passing through the specified points, using a slightly enhanced 4th-order Runge-Kutta numerical integrator. Integration will stop automatically if the solution moves too far away from the plotted region, or if the derivative becomes complex or infinite, or if division by zero occurs.

Example: Both soln_at and solns_at in use. Note that for soln_at, the point is not given as a list, but for solns_at, points are required to be given as lists.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        solns_at([0,0.1], [0,-0.1]),
        color = blue,               /* default is red */
        soln_at(0,0))$
```

Example: drawdf accepts most of the parameters supported by plotdf.[1]

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,y],
        [t,-5,10],
        [y,-4,9],
        [trajectory_at, 0,0])$
```

5.3 Options for drawdf and wxdrawdf

drawdf and wxdrawdf support all of the graphics options used by draw and its variants. They also support the options found in this section.

direction **Graphics option**

Default value: both

Allowable values: both, forward, reverse

[1] The wxMaxima documentation says this is for backward compatibility. Make of this what you will.

When drawdf plots a solution curve through a given point, its default behavior is to plot the curve for both positive and negative values of t. This is the behavior corresponding to the default value both of option direction. If direction is forward, then the solution curve is plotted only for positive values of t. If direction is negative, the solution curve is plotted only for negative values of t.

Example: Plotting the forward halves of solution curves.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        direction   = forward,
        soln_arrows = true,
        solns_at([0,0.1], [0,-0.1], [0,0]))$
```

duration — Graphics option

Default value: 10

Applies to: soln_at, solns_at

Allowable values: Any floating-point number

duration controls the lengths of solution curves by specifying the time duration of numerical integration. If duration is negative or 0, no solution curve is plotted. Integration will also stop automatically if the solution moves too far away from the plotted region, or if the derivative becomes complex or infinite, or if division by zero occurs.

Example: Three different values of duration.

```
(%i1) load(drawdf)$
      drawdf(2*cos(x)-1+y,
        [x,-12, 10],
        [y,-100,100],
        duration = 1,
        soln_at(0,0.3),
        color    = blue,
        duration = 5,
        soln_at(0,-0.3),
        color    = dark-green,
        duration = 10,
        soln_at(0,-0.6))$
```

Example: A model of a predator-prey system. duration has been increased to allow the system to evolve over a longer period of time.

```
(%i1) load(drawdf)$
      drawdf([x*(1-x-y), y*(3/4-y-x/2)],
        [x,0,1.1],
        [y,0,1],
        field_degree = 2,
        duration     = 40,
        soln_arrows  = true,
        point_at(1/2,1/2),
        solns_at([0.1,0.2], [0.2,0.1], [1,0.8], [0.8,1], [0.1,0.1],
          [0.6,0.05], [0.05,0.4], [1,0.01], [0.01,0.75]))$
```

field_arrows **Graphics option**

Default value: true

Allowable values: false, true

field_arrows governs whether the arrows of the direction field are drawn with arrow heads.

Example: Turning off the arrowheads of the direction field.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        field_arrows = false,
        color        = blue,
        soln_at(0,0))$
```

field_color **Graphics option**

Default value: black

Allowable values: Any color, whether named or in hexadecimal RGB format

See also: color, soln_arrows

field_color sets the color in which the arrows of the direction field are plotted.

Example: Changing the color of the direction field arrows to forest-green.

```
(%i1) drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        field_color = forest-green,
        soln_at(0,0.1))$
```

Example: Purple direction field arrows.

```
(%i1) drawdf(2*cos(t)-1+y, [t,-5,10], [y,-4,9],
        field_color  = "#800060",
        soln_at(0,0.1))$
```

field_degree **Graphics option**

Default value: 1

Allowable values: 1, 2, 'solns

field_degree governs the nature of the arrows of the direction field in the following way: If field_degree = 1, which is the default value, then the arrows are line segments. If field_degree = 2, the arrows are quadratic splines, based on the first and second derivtives at each grid point. If field_degree = 'solns, then the arrows are actually short solution curves, computed by the Runge-Kutta numerical integrator.

Example: The default behavior, using field_degree = 1.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        field_degree = 1,
        field_grid   = [20,15],
        solns_at([0,0.1], [0,-0.1]),
        color        = blue,
        soln_at(0,0))$
```

Example: field_degree = 2.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        field_degree = 2,
        field_grid   = [20,15],
        solns_at([0,0.1], [0,-0.1]),
        color        = blue,
        soln_at(0,0))$
```

Example: field_degree = 'solns causes the field to be composed of many small solution curves computed by a 4^{th}-order Runge-Kutta algorithm, with better results in this case.

```
(%i1) load(drawdf)$
      drawdf([x*(1-x-y), y*(3/4-y-x/2)],
        [x,0,1.1],
        [y,0,1],
        field_degree = 'solns,
        duration     = 40,
        soln_arrows  = true,
        point_at(1/2,1/2),
        solns_at([0.1,0.2], [0.2,0.1], [1,0.8], [0.8,1], [0.1,0.1],
          [0.6,0.05], [0.05,0.4], [1,0.01],  [0.01,0.75]))$
```

field_duration — Graphics option

Default value: (undetermined, at present)

Allowable values: Any floating-point number

See also: field_degree

When option field_degree is 'solns, drawdf uses short bits of solution curves as the arrows in the direction field. field_duration partially determines how long those solution curves are. If field_duration is 0 or negative, no direction field is drawn. drawdf also stops generating any given direction field arrow if its solution curve is long enough to potentially interfere with neighboring direction field arrows. Within these two constraints, the value of field_duration governs the lengths of the direction field arrows. Depending on the system of ODE's, useful values of field_duration can be small, on the order of 0.01 or 0.1, say. This option has no effect unless field_degree = 'solns.

Example: Compare the two plots to see the effect of field_duration

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        tstep          = 0.05,
        field_degree   = 'solns,
        field_duration = 0.05)$
      drawdf([y,-9*sin(x)-y/5],
        tstep          = 0.05,
        field_degree   = 'solns,
        field_duration = 0.2)$
```

field_grid — Graphics option

Default value: [30, 22]
field_grid = [cols,rows] specifies the number of columns and rows in the grid.

Example: A coarser field grid than the default.

```
(%i1) load(drawdf)$
      drawdf([x*(1-x-y), y*(3/4-y-x/2)],
        [x,0,1.1],
        [y,0,1],
        field_grid   = [15,15],
        soln_arrows  = true,
        solns_at([1,0.8], [0.8,1]))$
```

field_nsteps — Graphics option

Default value: (undetermined, at present)
Allowable values: Any floating-point number
See also: field_degree, field_duration

When option field_degree is 'solns, drawdf uses short bits of solution curves as the arrows in the direction field. The number of time steps used by the numerical integrator for each arrow is field_nsteps. For each arrow, integration stops automatically if the arrow becomes long enough to potentially interfere with any neighboring arrow(s). If field_nsteps is negative or 0, no direction field is plotted.

Example: Differing numbers of time steps for the direction field arrows.

```
(%i1) load(drawdf)$
      drawdf([x*(1-x-y), y*(3/4-y-x/2)],
        [x,0,1.1],
        [y,0,1],
        field_degree = 'solns,
        field_nsteps = 1,
      drawdf([x*(1-x-y), y*(3/4-y-x/2)],
        [x,0,1.1],
        [y,0,1],
        field_degree = 'solns,
        field_nsteps = 5)$
```

field_tstep — Graphics option

Default value: 0.1
Allowable values: Any floating-point number
See also: field_degree

When option field_degree is 'solns, drawdf uses short bits of solution curves as the arrows in the direction field. field_tstep is used by the numerical integrator as the length of time between plotted points of any given arrow. Increasing field_tstep can result in more accurate direction field arrows and a smoother plot.

nsteps — Graphics option

Default value: (undetermined, at present)
Allowable values: Any floating-point number

drawdf uses a numerical integrator to plot solution curves. nsteps is the number of time steps used in doing so. Integration will stop automatically if the curve extends too far beyond the plot window, or if any derivative becomes complex or infinite, or if division by zero occurs. If nsteps is 0 or negative, no solution curve is plotted.

Example: Differing numbers of time steps.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        field_grid = [20,15],
        nsteps     = 10,
        soln_at(0,0),
        nsteps     = 50,
        color      = blue,
        soln_at(0,0.3))$
```

show_field — Graphics option

Default value: `true`

This option controls whether the direction field is plotted or not.

Example: Suppressing the direction field by using `show_field = false`.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        tstep       = 0.05,
        show_field  = false,
        soln_arrows = true,
        saddles_at([3*%pi,0], [-3*%pi,0], [%pi,0], [-%pi,0]))$
```

soln_arrows — Graphics option

Default value: `false`

`soln_arrows = true` adds arrows to the solution curves, and (by default) removes them from the direction field. It also changes the default colors to emphasize the solution curves. **Example:** Compare the two examples to see the effect of `soln_arrows`.

```
(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        solns_at([0,0.1], [0,-0.1], [0,0]))$

(%i1) load(drawdf)$
      drawdf(2*cos(t)-1+y,
        [t,-5,10],
        [y,-4,9],
        soln_arrows = true,
        solns_at([0,0.1], [0,-0.1], [0,0]))$
```

tstep — Graphics option

Default value: `0.1`

`tstep` controls the mesh size for the time variable, as used by the numerical integrator. Decreasing `tstep` can improve the accuracy of the solution and make for a smoother plot.

Example: The pendulum again, but with the default `tstep` value of 0.1.

```
(%i1) load(drawdf)$
      drawdf([y,-9*sin(x)-y/5],
        soln_arrows  = true,
        point_size   = 0.5,
        points_at([0,0], [2*%pi,0], [-2*%pi,0]),
        field_degree = 'solns,
        saddles_at([%pi,0], [-%pi,0]))$
```

Chapter 6

The dynamics package

The dynamics package is used for graphing discrete dynamical systems and fractals. It uses plot2d for graphing, and all its functions but the julia and mandelbrot functions accept all options accepted by plot2d. It also includes an implementation of the 4^{th} order Runge-Kutta method for solving systems of differential equations. To use the functions in this package you must first load it with load(dynamics).

If your system is slow, you'll have to reduce the number of iterations in some of the following examples. And if the dots appear too small in your monitor, you might want to try a different style, such as [style, [points,0.8]].

6.1 Functions for dynamics

chaosgame([[x_1,y_1],...,[x_m,y_m]], [x_0, y_0], b, n, options) **Function**

Options: All those used by plot2d

Implements the so-called "chaos game:" The initial point [x_0, y_0] is plotted and then one of the m points [x_1, y_1],ldots,[x_m, y_m] is selected at random. The next point plotted will be on the segment from the previous point plotted to the point chosen randomly, at a distance from the random point which will be b times that segment's length. The procedure is repeated n times.

Example: Sierpinsky's triangle, via chaosgame:

```
(%i1) load(dynamics)$
      chaosgame([[0, 0], [1, 0], [0.5, sqrt(3)/2]], [0.1, 0.1], 1/2,
        30000, [style, dots]);
```

evolution(F, y_0, n, options) **Function**

Options: All those used by plot2d

Draws $n+1$ points in a two-dimensional graph, where the horizontal coördinates of the points are the integers $0, 1, 2, \ldots, n$, and the vertical coördinates are the corresponding values $y(n)$ of the sequence defined by the recurrence relation $y(n+1) = F\big(y(n)\big)$, with initial value $y(0)$ equal to y_0. F must be an expression that depends only on one variable (in the example, it depend on y, but any other variable can be used), y_0 must be a real number and n must be a positive integer.

evolution_2d([F,G], [u,v], [u_0,y_0], n, options) **Function**

Options: All those used by plot2d

Shows, in a two-dimensional plot, the first $n+1$ points in the sequence of points defined by the two-dimensional discrete dynamical system with recurrence relations

$$u(n+1) = F\big(u(n), v(n)\big) \qquad v(n+1) = G\big(u(n), v(n)\big),$$

with initial values u_0 and v_0. F and G must be two expressions that depend only on two variables, u and v, which must be named explicitly in a list.

ifs([r1,...,rm], [A_1,...,A_m], [[x_1,y_1],...,[x_m,y_m]], [x_0,y_0], n, options) **Function**

Options: All those used by plot2d

Implements the iterated function system method of drawing a fractal. This method is similar to the method described in the function chaosgame, with this difference: Instead of shrinking the segment from the current point to the randomly chosen point, the two components of that segment will be multiplied by the 2×2 matrix A_i that corresponds to the point chosen randomly.

The random choice of one of the m attractors can be made with a non-uniform probability distribution defined by the weights r_1,...,r_m. Those weights are given in cumulative form; for instance if there are 3 points with probabilities 0.2, 0.5 and 0.3, the weights r_1, r_2 and r_3 could be 2, 7 and 10.

Example: Barnsley's fern, obtained with an iterated function system:

```
(%i1) load(dynamics)$
      a1: matrix([0.85,0.04], [-0.04,0.85])$
      a2: matrix([0.2,-0.26], [0.23,0.22])$
      a3: matrix([-0.15,0.28], [0.26,0.24])$
      a4: matrix([0,0], [0,0.16])$
      p1: [0,1.6]$
      p2: [0,1.6]$
      p3: [0,0.44]$
      p4: [0,0]$
      w:  [85,92,99,100]$
      ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);
```

julia(x, y, options) **Function**

Options: center, color, filename, hue, huerange, levels, radius, saturation, size, value

Creates a file containing a graphical representation of the Julia set for the complex number $x + iy$. The parameters x and y must be real. The file is created in the current directory or in the user's directory, using the XPM graphics format. The program may take several seconds to run. When it is finished, a message will be printed with the name of the file created.

Each point that does not belong to the Julia set is assigned a color, as follows: The sequence of iterates beginning with the given point is computed until the first time an iterate lies outside the circle of radius 2 (centered at the origin), and the number of iterations required is determined. The color used in plotting the given point is determined by this number of iterations. However, if the number of iterations exceeds the value of levels, the iteration will stop. If the final iterate is inside the convergence circle, the point will be painted with the color defined by the option color.

All the colors used for the points that do not belong to the Julia set will have the same saturation and value, but with different hue angles, which will be distributed uniformly from hue to (hue + huerange).

The list of accepted options is given in Section 6.2 below.

Example: The Julia set for the number (-0.55 + i 0.6). The graph will be saved in the file dynamics10.xpm and will show the region from -0.3 to 0.3 in the x direction, and from 0.3 to 0.9 in the y direction. 36 colors will be used, starting with blue and ending with yellow.

```
(%i1) load(dynamics)$
      julia(-0.55, 0.6,
        [levels, 36],
        [center, 0,0.6],
        [radius, 0.3],
        [hue, 240],
        [huerange, -180],
        [filename, "dynamics10"])$
```

`mandelbrot(options)` **Function**

Options: center, color, filename, hue, huerange, levels, radius, saturation, size, value

Creates a graphics file with the representation of the Mandelbrot set. The file is created in the current directory or in the user's directory, using the XPM graphics format. The program may take several seconds to run. When it is finished, a message will be printed with the name of the file created.

Each point that does not belong to the Mandelbrot set is assigned a color, as follows: The sequence of iterates beginning with the given point is computed until an iterate lies outside the circle of radius 2 (centered at the origin), and the number of iterations required is determined. The color used in plotting the given point is determined by this number of iterations. The maximum number of iterations is set with the option levels. After this number of iterations, if the sequence is still inside the convergence circle, the point will be painted with the color defined by the option color.

All the colors used for the points that do not belong to the Mandelbrot set will have the same saturation and value, but with different hue angles, which will be distributed uniformly from hue to (hue + huerange).

In the syntax given above, options is an optional sequence of options. The list of accepted options is given in Section 6.2.

Example: The Mandelbrot set, with 12 colors, stored in a file named dynamics9.xpm.

```
(%i1) load(dynamics)$
      mandelbrot([filename,"dynamics9"])$
```

`orbits(F, y_0, n_1, n_2, [x, x_0,x_final, xstep], options)` **Function**

Options: pixels and all options used by plot2d

Draws the orbits diagram for a family of one-dimensional discrete dynamical systems, with one parameter x. (This kind of diagram is used to study the bifurcations of a one-dimensional discrete dynamical system.)

F, which is a function of y, defines a sequence with a starting value of y_0, as in the case of the function evolution. However, in this case that F will also depend on a parameter x taking values in the interval from x_0 to x_final, with increments of xstep. Each value used for the parameter x is shown on the horizontal axis. The vertical axis will show the n_2 values of the sequence y(n_1+1),...,y(n_1+n_2+1) obtained after letting the sequence evolve n_1 iterations.

Example: Orbits diagram for the quadratic map $x(n+1) = a + x(n)^2$, with parameter a.

```
(%i1) load(dynamics)$
      orbits(x^2+a, 0, 50, 200, [a, -2, 0.25], [style, dots])$
```

Example: To enlarge the region around the lower bifurcation near $x = -1.25$ use:

```
(%i1) load(dynamics)$
      orbits(x^2+a, 0, 100, 400,
        [a,-1,-1.53],
        [x,-1.6,-0.8],
        [nticks, 400],
        [style, dots])$
```

`rk(ODE, var, initial, domain_list)` **Function**

`rk([ODE_1,...,ODE_m], [v_1,...,v_m], [init_1,...,init_m], domain_list)` **Function**

The command `rk(ODE, var, initial, domain_list)` solves numerically a first-order ordinary differential. The command `rk([ODE_1,...,ODE_m], [v_1,...,v_m], [init_1,...,init_m], domain_list)` second form solves numerically a system of m such equations. Both commands use the 4^{th} order Runge-Kutta method. `var` is the dependent variable. `ODE` must be an expression that depends only on the independent and dependent variables and defines the derivative of the dependent variable with respect to the independent variable.

The independent variable is specified within `domain_list`, which must be a list of the form `[variable_name, initial_value, final_value, step_size]`. For instance, the `domain_list` `[t, 0, 10, 0.1]` declares `t` to be the independent variable, that the first and last values of `t` to be used are 0 and 10, respectively, and that the step size is 0.1.

The number m of dependent variables must be the same as the number of equations to be solved. The dependent variables are called `v_1,v_2,...,v_m`, in the syntax given above. The initial values for those variables will be `init_1,init_2,...,init_m`. There will still be just one independent variable defined by domain, as in the previous case. `ODE_1,ODE_2,...,ODE_m` are the expressions that define the derivatives of each dependent variable in terms of the independent variable and any of the dependent variables. It is important to give the derivatives `ODE_1,ODE_2...,ODE_m` in the list in exactly the same order used for the dependent variables; for instance, the third element in the list will be interpreted as the derivative of the third dependent variable.

The program will try to integrate the equations from the initial value of the independent variable to the final value, using constant increments. If at some step the absolute value of one of the dependent variables is too large, the integration will be interrupted. The result will be a list with as many elements as the number of iterations made. Each element in the list `results` is itself another list with $m + 1$ elements: the value of the independent variable, followed by the corresponding values of the m dependent variables.

Example: Numerical solution of $dx/dt = t - x^2$ with initial value $x_0 = 1$, in the interval of t from 0 to 8 in increments of 0.1. The results will be saved in a list called `results`.

```
(%i1) results: rk(t-x^2,x,1,[t,0,8,0.1])$
```

Example: Numerical solution of the system

$$dx/dt = 4 - x^2 - 4y^2 \qquad dy/dt = y^2 - x^2 + 1,$$

for t between 0 and 4, with initial values $x_0 = -1.25$ and $y_0 = 0.75$. The solution will be saved in a list called `sol`.

```
(%i1) sol: rk([4-x^2-4*y^2,y^2-x^2+1],
              [x,y],
              [-1.25,0.75],
              [t,0,4,0.02])$
```

staircase(F, y_0, m, options) **Function**

Options: All those used by plot2d

Draws a staircase diagram for the sequence defined by the recurrence relation $y(n+1) = F\big(y(n)\big)$.

The interpretation and allowed values of the input parameters are the same as for the function evolution. A staircase diagram consists of a plot of the function $F(y)$, together with the line $G(y) = y$. A vertical segment is drawn from the point (y_0, y_0) on that line until the point where it intersects the graph of the function F. From that point a horizontal segment is drawn until it reaches the point (y_1, y_1) on the line, and the procedure is repeated m times until the point (y_n, y_n) is reached.

Example: Graphical representation and staircase diagram for the sequence $\{2,\ \cos(2),\ \cos\big(\cos(2)\big),\ \ldots\}$

```
(%i1) load(dynamics)$
      evolution(cos(y), 2, 11)$
(%o2)

(%i3) staircase(cos(y), 1, 11, [y, 0, 1.2])$
(%o4)
```

6.2 Options for dynamics

Each option is a list of two or more items. The first item is the name of the option, and the remainder comprises the arguments for the option. Most of the functions in this package accept all the options that plot2d accepts. Additional options now follow.

[center,x,y] dynamics **option**

Default setting: [center,0,0]

Allowable values: Any pair of floating-point numbers

Applies to: julia, mandelbrot

This option is a list of two real parameters, which give the coördinates, on the complex plane, of the point in the center of the region shown. The default value of 0 for both coördinates defines the origin.

[color,hue,saturation,value] dynamics **option**

Default setting: [color,0,0,0]

Allowable values: julia, mandelbrot

Applies to: julia, mandelbrot

This option is a list of three parameters that define the hue, saturation, and value for the color used to represent the points of the set to be plotted. The default setting of 0,0,0 corresponds to black. For an explanation of the range of allowed values, see hue, saturation and value.

[filename,path_name] dynamics **option**

Default setting: [filename, Either julia or mandelbrot]

Allowable values: Any valid pathname

Applies to: julia, mandelbrot

filename defines the path name of the file where the resulting graph will be saved. The extension .xpm will be added to that name. If the file already exists, it will be replaced by the file generated by the function. The default values are julia for the Julia set, and mandelbrot for the Mandelbrot set.

[hue,n] dynamics option

Default setting: [hue,300]
Allowable values:
Applies to: julia, mandelbrot
See also: huerange

hue sets the hue, in degrees, of the first color used for those points that do not belong to the plotted set. Its default value of 300 degrees corresponds to magenta; the hue angles for other standard colors are 0 for red, 45 for orange, 60 for yellow, 120 for green, 180 for cyan and 240 for blue.

[huerange,n] dynamics option

Default setting: [huerange,360]
Allowable values:
Applies to: julia, mandelbrot

This option defines the range of hue angles used for points not belonging to the set. The default value of 360 makes the full range of hues available. Values greater than 360 will mean repeated ranges of the hue, and negative values can be used to make the hue angle decrease as the number of iterations increases.

[levels,n] dynamics option

Default setting: [levels,12]
Allowable values:
Applies to: julia, mandelbrot

levels defines the maximum number of iterations, which is also equal to the number of colors used for points not belonging to the set. Larger values of levels mean much longer processing times.

[pixels,n] dynamics option

Default setting: [pixels,]
Allowable values:
Applies to: orbits

This option defines the maximum number of different points that will be represented in the vertical direction.

[radius,r] dynamics option

Default setting: [radius,2]
Allowable values:
Applies to: julia, mandelbrot

radius sets the radius of the biggest circle inside the square region that will be displayed.

[saturation,n] dynamics option

Default setting: [saturation,0.46]
Allowable values: Any floating point number between 0 and 1, inclusive
Applies to: julia, mandelbrot

saturation sets the value of the saturation used for colors of points not belonging to the plotted set.

[size,side_length] dynamics option

Default setting: [size,400]

[size,width,height] dynamics option

Default setting: [size,400,400]

Allowable values:
Applies to: julia, mandelbrot

If only one argument is given, size causes the graphic in question to have height and width equal to side_length, in pixels. If two arguments are given, they will define the width and height, respectively, of the graphic. Note that if the two values are not equal, the aspect ratio of the graphic will not be 1:1, so the graphic will appear distorted.

[value,n] **dynamics option**

Default setting: [value,0.96]
Allowable values:
Applies to: julia, mandelbrot

This options sets the value of the colors used for points not belonging to the plotted set. Higher values of hue result in brighter colors.

Chapter 7

The `finance` package

The `finance` package offers exactly one graphics function: `graph_flow`. Its purpose is to plot the time flow of money, but it can be used to plot any sequence.

`graph_flow([y_1,y_2,...y_n])` **Function**

`graph_flow` Plots the money flow in a time line. Positive values are in blue and above the horizontal axis; negative ones are in red and below the horizontal axis. `[y_1,y_2,...y_n]` is a list of flow values. `graph_flow` admits no options.

Example: A simple financial flow.

```
(%i1) load(finance)$
      graph_flow([-5000,-3000,800,1300,1500,2000])$
```

Chapter 8

The fractals package

The fractals package supplies data that can be used with the image graphics object of the draw package, or with the plot2d or plot3d commands, to create pictures of iterated function systems (e.g., trees, ferns, Sierpinski gaskets), Mandelbrot sets, Julia sets, Koch snowflakes, and Peano maps (e.g., the Sierpinski and Hilbert maps).

Fractals of the "iterated function system" type are generated by iterative applications of contractive affine transformations. A list consisting of several contractive affine transformations is defined, and a starting point chosen. We randomly select a transformation from the list, apply it to the point, and plot the new point. This process is repeated, with the transformation used in each iteration being chosen randomly from the list in a recursive way. The probability of the choice of a transformation must be related to the contraction ratio. You can change the transformations to define new fractals.

Peano maps yields continuous space-filling curves. The so-called *replacement algorithm* is used: An initial curve consisting of several segments is chosen. At each iteration, each segment of the curve is replaced with an appropriately oriented, scaled-down version of the initial curve. If the initial curve is chosen carefully, then after infinitely many iterations, the resulting curve can fill a two-dimensional region.

fractals creates Koch snowflakes using the replacement algorithm, except that it starts from a polygon instead of a segmented curve. The result of the algorithm, after infinitely many iterations, is a curve with fractal dimension between 1 and 2.

8.1 Graphics objects for fractals

`fernfale(n)` **Graphics object**

frenfale creates a fern fractal using four predefined contractive maps, a predefined a scale factor, and predefined translations. The probabilities of choosing among the contractive maps is related in a specific way to the scale factor. Argument n is the number of iterations. To be able to see the fern well at all, n needs to be 10000 or greater.[1] Output from this function, even with the $ used to suppress output, is very verbose.

Example: A fern fractal.

```
(%i1) load(fractals)$
      plot2d([discrete,fernfale(10000)], [style,dots])$
```

`hilbertmap(iterations)` **Graphics object**

[1] This command runs quickly, so using 10,000 iterations is not a problem.

hilbert_map plots the points given by the Hilbert map. Argument iterations is the number of recursive applications of the Koch transformation. **Warning:** The number of points drawn grows exponentially with the number of iterations. iterations needs to be small, say, no larger than 6. Output from this function, even with the $ used to suppress output, is very verbose.

Example: The Hilbert map.

```
(%i1) load(fractals)$
      plot2d([discrete,hilbertmap(5)])$
```

julia_set(xrange, yrange) — Graphics object

See also: julia_parameter

julia_set makes Julia sets using the transformation julia_parameter+z^2. This program is time-consuming because it requires many computations. Also, the denser the grid, the longer the command requires.

Example: A Julia set.

```
(%i1) load(fractals)$
      plot3d(julia_set, [x, -2, 1], [y, -1.5, 1.5],
        [gnuplot_preamble, "set view map"],
        [gnuplot_pm3d, true],
        [grid, 150, 150])$
```

julia_sin(xrange, yrange) — Graphics object

See also: julia_parameter, julia_set

julia_sin makes Julia sets using the transformation julia_parameter*sin(z). See the source code for more details. This program is time consuming because it requires many computations, in particular, the computation of the sines of many numbers. Also, the denser the grid, the longer the command requires.

Example: A Julia set.

```
(%i1) load(fractals)$
      julia_parameter:1+.1*%i$
      plot3d(julia_sin, [x, -2, 2], [y, -3, 3],
        [gnuplot_preamble, "set view map"],
        [gnuplot_pm3d, true],
        [grid, 150, 150])$
```

mandelbrot_set(xrange, yrange) — Graphics object

mandelbrot_set draws the Mandelbrot set in the complex plane. This program is time-consuming because it requires many computations. Also, the denser the grid, the longer the command requires.

Example: The Mandelbrot set.

```
(%i1) load(fractals)$
      plot3d(mandelbrot_set, [x, -2.5, 1], [y, -1.5, 1.5],
        [gnuplot_preamble, "set view map"],
        [gnuplot_pm3d, true],
        [grid, 150, 150])$
```

sierpinskiale(n) **Graphics object**

sierpinskiale creates the triangular Sierpinski gasket using three predefined contractive maps, a scale factor (contraction constant) of 0.5, and predefined translations. Argument n is the number of iterations. To be able to see the gasket well at all, n needs to be 10000 or greater.[2]

Example: The triangular Sierpinski gasket.

```
(%i1) load(fractals)$
      plot2d([discrete,sierpinskiale(10000)], [style,dots])$
```

sierpinskimap(iterations) **Graphics object**

sierpinksi_map plots the points given by the Sierpinski map. Argument iterations is the number of recursive applications of the Koch transformation. **Warning:** The number of points drawn grows exponentially with the number of iterations. iterations needs to be small, say, no larger than 6.

Example: The Sierpinski map.

```
(%i1) load(fractals)$
      plot2d([discrete,sierpinskimap(5)])$
```

snowmap([cx_1, cx_2,...,cx_n], iterations) **Graphics object**

snowmap plots the Koch snowflake map over the vertex set of an initial closed polygonal, in the complex plane. Because coördinate geometry is used, the orientation of the polygon matters. [cx_1, cx_2,...,cx_n] is a list of complex numbers, providing the vertex set of the desired polygon. Argument iterations is the number of recursive applications of the Koch transformation. **Warning:** The number of points drawn grows exponentially with the number of iterations. iterations needs to be small, say, no larger than 6.

Example: Some Koch snowflakes. Compare them to see the effect of different initial orientations.

```
(%i1) load(fractals)$
      plot2d([discrete, snowmap([1,exp(%i*%pi*2/3),exp(-%i*%pi*2/3),1],4)])$
      plot2d([discrete, snowmap([1,exp(-%i*%pi*2/3),exp(%i*%pi*2/3),1],4)])$
      plot2d([discrete, snowmap([0,1,1+%i,%i,0],4)])$
      plot2d([discrete, snowmap([0,%i,1+%i,1,0],4)])$
```

treefale(n) **Graphics object**

treefale creates a tree fractal using three predefined contractive maps, a predefined a scale factor, and predefined translations. Argument n is the number of iterations. To be able to see the tree well at all, n needs to be 10000 or greater.[3]

Example: A tree fractal.

```
(%i1) load(fractals)$
      plot2d([discrete,treefale(10000)], [style,dots])$
```

8.2 Graphics options for fractal

julia_parameter **Graphics option**

Default value: %i

[2] This command runs quickly, so using 10,000 iterations is not a problem.
[3] This command runs quickly, so using 10,000 iterations is not a problem.

Allowable values: `Any complex number`
See also: `julia_set`, `julia_sin`
Complex parameter for Julia sets. Its default value is i, the complex unit. Potentially interesting values include $-0.745 + 0.113002i$, $-0.39054 - 0.58679i$, $-0.15652 + 1.03225i$, $-0.194 + 0.6557i$, and $0.011031 - 0.67037i$.

Chapter 9

The graphs package

The "graphs" created by the graphs package are of the edge-and-vertex type, commonly known as "networks." This package is not for graphing things like curves or surfaces.

9.1 Introduction to graphs

The graphs package provides graph and digraph data structure for Maxima. Only simple[1] graphs and digraphs can be used with this package, although digraphs can have a directed edge from u to v and a directed edge from v to u. Internally, graphs are represented by adjacency lists and implemented as a lisp structures. Each vertex is identified by an integer. Edges are represented by lists of length 2, whose elements are vertex ID's. Labels can be assigned to vertices of graphs and digraphs, and weights can be assigned to edges. There is a draw_graph function for drawing graphs. Graphs are drawn using a force-based vertex positioning algorithm. draw_graph can also use graphviz programs available from http://www.graphviz.org, but these programs must be installed separately. draw_graph is based on the Maxima draw package.

To use the graphs package, first load it with load(graphs).

9.2 Graphics objects for graphs

circulant_graph(n, d) **Function**

Returns the circulant graph with parameters n and d. **Example:**

```
(%i1) load(graphs)$
      g : circulant_graph(10, [1,3])$
      print_graph(g)$
Graph on 10 vertices with 20 edges.
Adjacencies:
  9 :  2  6  0  8
  8 :  1  5  9  7
  7 :  0  4  8  6
  6 :  9  3  7  5
  5 :  8  2  6  4
  4 :  7  1  5  3
  3 :  6  0  4  2
  2 :  9  5  3  1
```

[1] A *simple* graph or digraph is one having no loops and no multiple edges.

```
1 :  8  4  2  0
0 :  7  3  9  1
```

clebsch_graph() **Function**
Returns the Clebsch graph.

complement_graph(graph) **Function**
Returns the complement of the graph graph.

complete_bipartite_graph(n, m) **Function**
Returns the complete bipartite graph on $n + m$ vertices. n and m must be positive integers.

complete_graph(n) **Function**
Returns the complete graph on n vertices. n must be a positive integer.

copy_graph(graph) **Function**
Returns a copy of the graph graph.

create_graph(v_list, e_list) **Function**
create_graph(n, e_list) **Function**
create_graph(v_list, e_list, directed) **Function**
Creates a new graph on the set v_list of vertices and with edges e_list. v_list is a list of vertices [v1, v2,..., vn] or a list of vertices together with vertex labels [[v1,l1], [v2,l2],...,[vn,ln]]. Here, n is the number of vertices. Each vertex in v_list must be an integer; no integer in v_list may be repeated. e_list is a list of edges [e1, e2,..., em] or a list of edges together with edge-weights [[e1, w1],..., [em, wm]]. Each edge in e_list is a list consisting of two vertices, as in [1,3]. If directed is not false, a directed graph will be returned.

Example: A cycle on three vertices.

```
(%i1) load(graphs)$
      g : create_graph([1,2,3], [[1,2], [2,3], [1,3]])$
      print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example: A cycle on three vertices, with edge weights.

```
(%i1) load(graphs)$
      g : create_graph([1,2,3], [[[1,2], 1.0], [[2,3], 2.0],
            [[1,3], 3.0]])$
      print_graph(g)$
Graph on 3 vertices with 3 edges.
Adjacencies:
  3 :  1  2
  2 :  3  1
  1 :  3  2
```

Example: A directed graph.

```
(%i1) load(graphs)$
      d : create_graph([1,2,3,4],
            [[1,3], [1,4],
            [2,3], [2,4]],
            'directed = true)$
      print_graph(d)$
Digraph on 4 vertices with 4 arcs.
Adjacencies:
  4 :
  3 :
  2 :  4  3
  1 :  4  3
```

cube_graph(n) **Function**

Returns the n-dimensional cube. n must be a positive integer.

cuboctahedron_graph(n) **Function**

Returns the cuboctahedron graph. n must be a positive integer.

cycle_digraph(n) **Function**

Returns the directed cycle on n vertices. n must be a positive integer.

cycle_graph(n) **Function**

Returns the cycle on n vertices. n must be a positive integer.

dodecahedron_graph() **Function**

Returns the dodecahedron graph.

empty_graph(n) **Function**

Returns the empty graph on n vertices. n must be a positive integer.

flower_snark(n) **Function**

Returns the flower graph on $4n$ vertices. nmust be a positive integer.

Example:

```
(%i1) load(graphs)$
      f5 : flower_snark(5)$
      chromatic_index(f5);
(%o3) 4
```

from_adjacency_matrix(A) **Function**

Returns the graph represented by its adjacency matrix A. The matrix A must be a square matrix.[2]

frucht_graph() **Function**

Returns the Frucht graph.

graph_product(g1, g1) **Function**

Returns the direct product of graphs g1 and g2.

[2] Author's note to self: A 0-1 matrix? Weights?

Example:

```
(%i1) load(graphs)$
      grid : graph_product(path_graph(3), path_graph(4))$
      draw_graph(grid)$
```

graph_union(g1, g1) **Function**
Returns the union (sum) of graphs g1 and g2.

great_rhombicosidodecahedron_graph() **Function**
Returns the great rhombicosidodecahedron graph.

great_rhombicuboctahedron_graph() **Function**
Returns the great rhombicuboctahedron graph.

grid_graph(n, m) **Function**
Returns the n × m grid. m and n must be positive integers.

grotzch_graph() **Function**
Returns the Grotzch graph.

heawood_graph() **Function**
Returns the Heawood graph.

icosahedron_graph() **Function**
Returns the icosahedron graph.

icosidodecahedron_graph() **Function**
Returns the icosidodecahedron graph.

induced_subgraph(V, g) **Function**
Returns the graph induced on the subset V of vertices of the graph g.

Example:

```
(%i1) load(graphs)$
      p : petersen_graph()$
      V : [0,1,2,3,4]$
      g : induced_subgraph(V, p)$
      print_graph(g)$
Graph on 5 vertices with 5 edges.
Adjacencies:
  4 :  3  0
  3 :  2  4
  2 :  1  3
  1 :  0  2
  0 :  1  4
```

line_graph(g) **Function**
Returns the line graph of the graph g.

make_graph(vrt, f) **Function**

make_graph(vrt, f, oriented) **Function**

Creates a graph using a predicate function f. vrt is either a list or set of vertices, or an integer. If vrt is an integer, then vertices of the graph will be integers from 1 to vrt. f is a predicate function. Two vertices a and b will be connected if $f(a, b) =$true. If directed is not false, then the graph will be directed.

Example:

```
(%i1) load(graphs)$
      g : make_graph(powerset({1,2,3,4,5}, 2), disjointp)$
      is_isomorphic(g, petersen_graph());
(%o3) true

(%i4) get_vertex_label(1, g);
(%o4) {1, 2}
```

Example:

```
(%i1) load(graphs)$
      f(i, j) := is(mod(j, i)=0)$
      g : make_graph(20, f, directed=true)$
      out_neighbors(4, g);
(%o4) [8, 12, 16, 20]

(%i5) in_neighbors(18, g);
(%o5) [1, 2, 3, 6, 9]
```

mycielski_graph(g) **Function**

Returns the Mycielskian graph of the graph g.

new_graph() **Function**

Returns the graph with no vertices and no edges.

path_digraph(n) **Function**

Returns the directed path on n vertices. n must be a positive integer.

path_graph(n) **Function**

Returns the path on n vertices. n must be a positive integer.

petersen_graph() **Function**
petersen_graph(n, d) **Function**

Returns the Petersen graph with parameters n and d. The default values for n and d are n = 5 and d = 2.

random_bipartite_graph(m, n, p) **Function**

Returns a random bipartite graph on $m + n$ vertices. Each edge is present with probability p.

random_digraph(n, p) **Function**

Returns a random directed graph on n vertices. Each arc is present with probability p.

random_regular_graph(n) **Function**
random_regular_graph(n, d) **Function**

Returns a random d-regular graph on n vertices. The default value for d is d = 3.

random_graph(n, p) **Function**

Returns a random graph on n vertices. Each edge is present with probability p.

random_graph1(n, m) **Function**

Returns a random graph on n vertices and random m edges.

random_network(n, p, w) **Function**

Returns a random network on n vertices. Each arc is present with probability p and has a weight between 0 and w, inclusive. The function returns the list [network, source, sink].

Example:

```
(%i1) load(graphs)$
      [net, s, t] : random_network(50, 0.2, 10.0);
(%o2) [DIGRAPH, 50, 51]

(%i3) max_flow(net, s, t)$
      first(%);
(%o4) 27.65981397932507
```

random_tournament(n) **Function**

Returns a random tournament on n vertices.

random_tree(n) **Function**

Returns a random tree on n vertices.

small_rhombicosidodecahedron_graph() **Function**

Returns the small rhombicosidodecahedron graph.

small_rhombicuboctahedron_graph() **Function**

Returns the small rhombicuboctahedron graph.

snub_cube_graph() **Function**

Returns the snub cube graph.

snub_dodecahedron_graph() **Function**

Returns the snub dodecahedron graph.

truncated_cube_graph() **Function**

Returns the truncated cube graph.

truncated_dodecahedron_graph() **Function**

Returns the truncated dodecahedron graph.

truncated_icosahedron_graph() **Function**

Returns the truncated icosahedron graph.

truncated_tetrahedron_graph() **Function**

Returns the truncated tetrahedron graph.

tutte_graph() **Function**
Returns the Tutte graph.

underlying_graph(g) **Function**
Returns the underlying graph of the directed graph g.

wheel_graph(n) **Function**
Returns the wheel graph on $n + 1$ vertices.

9.3 Functions for modifying graphs

add_edge(e, gr) **Function**
Adds the edge e to the graph gr.

Example:

```
(%i1) load(graphs)$
      p : path_graph(4)$
      neighbors(0, p);
(%o3) [1]

(%i4) add_edge([0,3], p);
(%o4) done

(%i5) neighbors(0, p);
(%o5) [3, 1]
```

add_edges(e_list, gr) **Function**
Adds all edges in the list e_list to the graph gr.

Example:

```
(%i1) load(graphs)$
      g : empty_graph(3)$
      add_edges([[0,1],[1,2]], g)$
      print_graph(g)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  1
  1 :  2  0
  0 :  1
```

add_vertex(v, gr) **Function**
Adds the vertex v to the graph gr.

Example:

```
(%i1) load(graphs)$
      g : path_graph(2)$
      add_vertex(2, g)$
      print_graph(g)$
Graph on 3 vertices with 1 edges.
Adjacencies:
```

```
  2 :
  1 :  0
  0 :  1
```

add_vertices(v_list, gr) **Function**

Adds all vertices in the list `v_list` to the graph `gr`.

connect_vertices(v_list, u_list, gr) **Function**

Connects all vertices from the list `v_list` with the vertices in the list `u_list` in the graph `gr`. `v_list` and u_list can be single vertices or lists of vertices.

Example:

```
(%i1) load(graphs)$
      g : empty_graph(4)$
      connect_vertices(0, [1,2,3], g)$
      print_graph(g)$
Graph on 4 vertices with 3 edges.
Adjacencies:
  3 :  0
  2 :  0
  1 :  0
  0 :  3  2  1
```

contract_edge(e, gr) **Function**

Contracts the edge `e` in the graph `gr`.

Example:

```
(%i1) load(graphs)$
      g: create_graph(8,
        [[0,3],[1,3],[2,3],[3,4],[4,5],[4,6],[4,7]])$
      print_graph(g)$
Graph on 8 vertices with 7 edges.
Adjacencies:
  7 :  4
  6 :  4
  5 :  4
  4 :  7  6  5  3
  3 :  4  2  1  0
  2 :  3
  1 :  3
  0 :  3

(%i4) contract_edge([3,4], g)$
      print_graph(g)$
Graph on 7 vertices with 6 edges.
Adjacencies:
  7 :  3
  6 :  3
  5 :  3
  3 :  5  6  7  2  1  0
```

```
  2 :  3
  1 :  3
  0 :  3
```

remove_edge(e, gr) **Function**

Removes the edge e from the graph gr.

Example:

```
(%i1) load(graphs)$
      c3 : cycle_graph(3)$
      remove_edge([0,1], c3)$
      print_graph(c3)$
Graph on 3 vertices with 2 edges.
Adjacencies:
  2 :  0  1
  1 :  2
  0 :  2
```

remove_vertex(v, gr) **Function**

Removes the vertex v from the graph gr.

9.4 File handling for graphs

dimacs_export(gr, fl) **Function**

dimacs_export(gr, fl, comment1,...,commentn) **Function**

Exports the graph into the file fl in the DIMACS format. Optional comments will be added to the top of the file.

dimacs_import(fl) **Function**

Returns the graph from file fl in the DIMACS format.

graph6_decode(str) **Function**

Returns the graph encoded in the graph6 format in the string str.

graph6_encode(gr) **Function**

Returns a string which encodes the graph gr in the graph6 format.

graph6_export(gr_list, fl) **Function**

Exports graphs in the list gr_list to the file fl in the graph6 format.

graph6_import(fl) **Function**

Returns a list of graphs from the file fl in the graph6 format.

sparse6_decode(str) **Function**

Returns the graph encoded in the sparse6 format in the string str.

sparse6_encode(gr) **Function**

Returns a string which encodes the graph gr in the sparse6 format.

`sparse6_export(gr_list, fl)` **Function**

Exports graphs in the list `gr_list` to the file `fl` in the sparse6 format.

`sparse6_import(fl)` **Function**

Returns a list of graphs from the file `fl` in the sparse6 format.

9.5 Rendering graphs

`draw_graph(graph)` **Function**
`draw_graph(graph, option_1,...,optionk)` **Function**

Draws the graph using the draw package.

The algorithm used to position vertices is specified by the optional argument program. The default value is `program = spring_embedding`. `draw_graph` can also use the graphviz programs for positioning vertices, but graphviz must be installed separately.

Example:

```
(%i1) load(graphs)$
      g:grid_graph(10,10)$
      m:max_matching(g)$
      draw_graph(g,
        spring_embedding_depth=100,
        show_edges  = m,
        edge_type   = dots,
        vertex_size = 0)$
```

Example:

```
(%i1) load(graphs)$
      g:create_graph(16,
          [[0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
          [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
          [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
          [10,14],[15,14],[13,14]])$
      t:minimum_spanning_tree(g)$
      draw_graph(g,
        show_edges       = edges(t),
        show_edge_width = 4,
        show_edge_color = green,
        vertex_type      = filled_square,
        vertex_size      = 2)$
```

Example:

```
(%i1) load(graphs)$
      g:create_graph(16,
          [[0,1],[1,3],[2,3],[0,2],[3,4],[2,4],
          [5,6],[6,4],[4,7],[6,7],[7,8],[7,10],[7,11],
          [8,10],[11,10],[8,9],[11,12],[9,15],[12,13],
          [10,14],[15,14],[13,14]])$
      mi : max_independent_set(g)$
      draw_graph(g,
        show_vertices     = mi,
```

```
  show_vertex_type = filled_up_triangle,
  show_vertex_size = 2,
  edge_color       = cyan,
  edge_width       = 3,
  show_id          = true,
  text_color       = brown)$
```

Example:

```
(%i1) load(graphs)$
      net : create_graph([0,1,2,3,4,5],
        [[[0,1], 3], [[0,2], 2],
         [[1,3], 1], [[1,4], 3],
         [[2,3], 2], [[2,4], 2],
         [[4,5], 2], [[3,5], 2]],
        directed          = true)$
      draw_graph(net,
        show_weight       = true,
        vertex_size       = 0,
        show_vertices     = [0,5],
        show_vertex_type = filled_square,
        head_length       = 0.2,
        head_angle        = 10,
        edge_color        = "dark-green",
        text_color        = blue)$
```

Example:

```
(%i1) load(graphs)$
      g: petersen_graph(20, 2);
(%o2) GRAPH(40 vertices, 60 edges)

(%i3) draw_graph(g, redraw=true, program=planar_embedding);
(%o3) done
```

Example:

```
(%i1) load(graphs)$
      t: tutte_graph();
(%o2) GRAPH(46 vertices, 69 edges)

(%i3) draw_graph(t, redraw=true, fixed_vertices=[1,2,3,4,5,6,7,8,9]);
(%o3) done
```

9.6 Graphics options for graphs

draw_graph_program **Graphics option**

Default value: spring_embedding

The default value for the program used to position vertices in the draw_graph function.

edge_color **Graphics option**

Default value:

The color used for displaying edges.

edge_coloring **Graphics option**

Default value:

The coloring of edges. The coloring must be specified in the format as returned by the function edge_coloring.

edge_partition **Graphics option**

Default value:

A partition [[e1,e2,...],...,[ek,...,em]] of edges of the graph. The edges of each list in the partition will be drawn using a different color.

edge_type **Graphics option**

Default value:

Defines how edges are displayed. See the line_type option for the draw package.

edge_width **Graphics option**

Default value:

The width of edges.

file_name **Graphics option**

Default value:

The filename of the drawing if terminal is not screen.

fixed_vertices **Graphics option**

Default value:

Specifies a list of vertices which will have positions fixed along a regular polygon. Can be used when program = spring_embedding.

head_angle **Graphics option**

Default value: 15

The angle for the arrows displayed on arcs (in directed graphs).

head_length **Graphics option**

Default value: 0.1

The length for the arrows displayed on arcs (in directed graphs).

label_alignment **Graphics option**

Default value: center

Determines how to align the labels/ids of the vertices. Can be left, center or right.

program **Graphics option**

Default value:

Defines the program used for positioning vertices of the graph. Can be one of the graphviz programs (dot, neato, twopi, circ, fdp), circular, spring_embedding or planar_embedding. planar_embedding is only available for 2-connected planar graphs. When program = spring_embedding, a set of vertices with fixed position can be specified with the fixed_vertices option.

redraw **Graphics option**

Default value: false

If true, vertex positions are recomputed even if the positions have been saved from a previous drawing of the graph.

show_edges **Graphics option**
Default value:
Display edges specified in the list e_list using a different color.

show_edge_color **Graphics option**
Default value:
The color used for displaying edges in the show_edges list.

show_edge_type **Graphics option**
Default value:
Defines how edges in show_edges are displayed. See the line_type option for the draw package.

show_edge_width **Graphics option**
Default value:
The width of edges in show_edges.

show_id **Graphics option**
Default value: false
If true then id's of the vertices are displayed.

show_label **Graphics option**
Default value: false
If true then labels of the vertices are displayed.

show_vertex_type **Graphics option**
Default value:
Defines how vertices specified in show_vertices are displayed. See the point_type option for the draw package for possible values.

show_vertex_size **Graphics option**
Default value:
The size of vertices in show_vertices.

show_vertex_color **Graphics option**
Default value:
The color used for displaying vertices in the show_vertices list.

show_vertices **Graphics option**
Default value: []
Display selected vertices in the using a different color.

show_weight **Graphics option**
Default value: false
If true then weights of the edges are displayed.

spring_embedding_depth **Graphics option**
Default value: 50
The number of iterations in the spring embedding graph drawing algorithm.

terminal **Graphics option**

Default value:

The terminal used for drawing (see the terminal option in the draw package).

vertex_color **Graphics option**

Default value:

The color used for displaying vertices.

vertex_coloring **Graphics option**

Default value:

Specifies coloring of the vertices. The coloring col must be specified in the format as returned by vertex_coloring.

vertex_partition **Graphics option**

Default value: []

A partition [[v1,v2,...],...,[vk,...,vn]] of the vertices of the graph. The vertices of each list in the partition will be drawn in a different color.

vertex_size **Graphics option**

Default value:

The size of vertices.

vertex_type **Graphics option**

Default value: circle

Defines how vertices are displayed. See the point_type option for the draw package for possible values.

vertices_to_cycle(v_list) **Function**

Converts a list v_list of vertices to a list of edges of the cycle defined by v_list.

vertices_to_path(v_list) **Function**

Converts a list v_list of vertices to a list of edges of the path defined by v_list.

vertex_type **Graphics option**

Default value: circle

Defines how vertices are displayed. See the point_type option for the draw package for possible values.

Chapter 10

The implicit_plot package

implicit_plot(expr, x_range, y_range), options **Function**
implicit_plot([expr_1,..., expr_n], x_range, y_range, options) **Function**

To use implicit_plot, you must load it first, via load(implicit_plot).

implicit_plot displays a plot of one or more expressions in implicit form. expr is the expression to be plotted, x_range the range of the horizontal axis and y_range the range of vertical axis (not to be confused with the draw options by the same names). implicit_plot respects the global settings set by the set_plot_option function. Options can also be passed to implicit_plot function as optional arguments.

implicit_plot works by tracking sign changes on the area given by x_range and y_range and can fail for complicated expressions.

Example: An implicit plot.

```
(%i1) load(implicit_plot)$
      implicit_plot(x^2 = y^3 - 3*y + 1, [x, -4, 4], [y, -4, 4],
        [gnuplot_preamble, "set zeroaxis"])$
```

Chapter 11

The picture package

wxMaxima comes with a third-party package called picture, for handling graphics at the pixel level. It appears that documentation for this package is found within the documentation for draw.[1] This seems appropriate, because draw automatically calls picture when needed. Objects and functions associated with picture are discussed in this section.

get_pixel(pic, x,y) **Function**

See also: make_level_picture, make_rgb_picture

Returns the pixel located at the point (x, y) from picture pic. The coördinates x and y range from 0 to width−1 and height−1, respectively. The nature of the pixel depends on the type of picture pic is. See make_level_picture and make_rgb_picture for details.

make_level_picture(data) **Function**

make_level_picture(data, width,height) **Function**

Returns a "levels" picture object. data is a matrix; make_level_picture(data) builds the picture object from this matrix. make_level_picture(data, width,height) builds the object from a list of numbers. In this case, both the width and the height must be given. The returned picture object contains the following four parts:

1. the symbol level,
2. the width of the image,
3. the height of the image, and
4. an integer array with pixel data ranging from 0 to 255.

The matrix data must contain only numbers ranging from 0 to 255. Negative numbers are replaced by 0, and numbers greater than 255 are set to 255.

Example: A level picture from a matrix.

```
(%i1) load(draw)$
      make_level_picture(matrix([3,2,5], [7,-9,3000]));
(%o2) picture(level, 3, 2, {Array: }(3 2 5 7 0 255)})
```

Example: Level picture from numeric list.

[1] At least, in version 5.24.0 of wxMaxima, anyway.

```
(%i1) load(draw)$
      make_level_picture([-2,0,54,%pi],2,2);
(%o2) picture(level, 2, 2, {Array: }(0 0 54 3))
```

make_rgb_picture(redlevel, greenlevel, bluelevel) **Function**

Returns an RGB-colored picture object. All three arguments must be "levels" picture objects, with red, green, and blue levels, respectively. The returned picture object contains the following four parts:

1. the symbol `rgb`,
2. the width of the image,
3. the height of the image, and
4. an integer array of length 3*`width*height`, containing pixel data ranging from 0 to 255.

Each pixel is represented by three consecutive numbers (red, green, blue).

Example: Building an `rgb` picture.

```
(%i1) load(draw)$
      red: make_level_picture(matrix([3,2],[7,260]));
(%o2) picture(level, 2, 2, {Array: }(3 2 7 255)})

(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3) picture(level, 2, 2, {Array: }(54 23 73 0)})

(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4) picture(level, 2, 2, {Array: }(123 82 45 33)})

(%i5) make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2, {Array: }(3 54 123 2 23 82 7 73 45 255 0 33)})
```

negative_picture(pic) **Function**

Returns the negative of the picture `pic`, which may be either a "levels" picture object or an RGB picture object.

picture_equalp(pic1, pic2) **Function**

Returns `true` if `pic1` and `pic2` are equal pictures, and `false` otherwise.

picturep(pic) **Function**

Returns `true` if `pic` is a well-formed image, and `false` otherwise.

read_xpm(le) **Function**

Reads an xpm le and converts it to a picture object.

rgb2level(pic) **Function**

Transforms an RGB picture into a "levels" one by averaging the red, green and blue channels.

take_channel(im, color) **Function**

If argument color is `red`, `green` or `blue`, function `take_channel` returns the corresponding color channel of picture `im`.

Example: Taking the green channel.

```
(%i1) load(draw)$
      red: make_level_picture(matrix([3,2],[7,260]));
(%o2) picture(level, 2, 2, {Array: }(3 2 7 255)})

(%i3) green: make_level_picture(matrix([54,23],[73,-9]));
(%o3) picture(level, 2, 2, {Array: }(54 23 73 0)})

(%i4) blue: make_level_picture(matrix([123,82],[45,32.5698]));
(%o4) picture(level, 2, 2, {Array: }(123 82 45 33)})

(%i5) mypic : make_rgb_picture(red,green,blue);
(%o5) picture(rgb, 2, 2, {Array: }(3 54 123 2 23 82 7 73 45 255 0 33)})

(%i6) take_channel(mypic,green); /* note the apostrophe */done$
(%o6) picture(level, 2, 2, {Array: }(54 23 73 0)})
```

Chapter 12

The plot commands

plot and its variants are external to Maxima itself. These plotting functions calculate a set of points to be plotted and pass them to a plotting package (typically gnuplot), together with a set of commands. This passing of information can be done through a pipe or by calling the plotting package with the name of a file, in which the data for the plot have been saved. The data file is given the name maxout.FORMAT, where FORMAT is the name of the plotting format being used (gnuplot, Xmaxima, mgnuplot, or gnuplot_pipes). (See Section 1.4 on plotting formats).

The maxout.FORMAT file, in the cases when it is used, is created in the directory specified by the system variable `maxima_tempdir`. This variable can be assigned to any valid directory in which Maxima can create new files. After a plot has been created, the file maxout.FORMAT can be executed again with the appropriate plotting program. If a Maxima plotting command fails to show anything, maxout.FORMAT can be inspected for possible sources of problems.

12.1 The plot and related functions

`contour_plot(expr, x_range, y_range, options)` **Function**

See also: implicit_plot

`contour_plot` plots the contours (level curves) of `expr` over the region `x_range` by `y_range`. Any additional arguments are treated the same as in `plot3d`.

Example: Two basic contour plots.

```
(%i1) contour_plot(x^2 + y^2, [x, -4, 4], [y, -4, 4])$

(%i2) F(x, y) := x^3 + y^2;
      contour_plot(F, [u, -4, 4], [v, -4, 4])$
```

You can add any options accepted by `plot3d`. For example, if option `legend` is `false`, the legend is not included in the graphic. gnuplot shows 3 contours, by default. To increase the number of levels, it is necessary to specify a custom gnuplot preamble. **Example:** Using the `gnuplot_preamble` option to change the number of contours:

```
(%i1) contour_plot(u^3 + v^2, [u, -4, 4], [v, -4, 4], [legend,false],
        [gnuplot_preamble, "set cntrparam levels 12"])$
```

`make_transform([var1, var2, var3], fx, fy, fz)` **Function**

See also: polar_to_xy and spherical_to_xyz

Returns a function suitable for use in the option transform_xy of plot3d. Variables var1, var2, and var3 are dummy variable names, representing the variables used by the plot3d command: first the two independent variables and then the function that depends on those two variables. The functions fx, fy, fz must depend only on var1, var2, and var3, and will give the corresponding x-, y- and z-coördinates of the points to be plotted. There are two transformations defined by default: polar_to_xy and spherical_to_xyz.

polar_to_xy **System function**

Can be used as the value of the transform_xy option of plot3d. Its effect will be to interpret the two independent variables in plot3d as the radial distance from the origin to the point and the azimuthal angle, respectively (polar coördinates), and transform them into x- and y-coördinates.

Example: Use of the system function polar_to_xy to transform from cylindrical to rectangular coördinates. This example also shows how to eliminate the legend and the box around the plot.

```
(%i1) plot3d(r^.33*cos(th/3), [r, 0, 1], [th, 0, 6*%pi],
        [grid, 12, 80],
        [transform_xy, polar_to_xy],
        [box, false],
        [legend,false])$
```

plot2d(plot, x_range, options) **Function**
plot2d([plot_1, ..., plot_n], options) **Function**
plot2d([plot_1, ..., plot_n], x_range,options) **Function**

In the above syntax, plot, plot_1, ..., plot_n can be expressions, function names, or a list with the any of these forms: [discrete, [x1, ..., xn], [y1, ..., yn]], [discrete, [[x1, y1], ..., [xn, ..., yn]]], or [parametric, x_expr, y_expr, t_range].

Displays a plot of one or more expressions as a function of one variable or parameter.

plot2d displays one or several plots in two dimensions. When expressions or function name are used to define the plots, they should all depend on the same single variable var. The use of x_range is mandatory, to provide the name of the variable and its minimum and maximum values. The syntax for x_range is: [variable, min, max].

A plot can also be defined in the discrete or parametric forms. The discrete form is used to plot a set of points with given coördinates. A discrete plot is defined by a list starting with the keyword discrete, followed by one or two lists of values. If two lists are given, they must have the same length; the first list will be interpreted as the x coördinates of the points to be plotted and the second list as the y coördinates. If only one list is given after the discrete keyword, each element on the list should also be a list with two values that correspond to the x and y coördinates of a point.

A parametric plot is defined by a list starting with the keyword parametric, followed by two expressions or function names and a range for the parameter. The range for the parameter must be a list with the name of the parameter followed by its minimum and maximum values: [param, min, max]. The plot will show the path traced out by the point with coördinates given by the two expressions or functions, as param increases from min to max.

A range for the vertical axis is an optional argument with the form: [y, min, max] (the keyword y is always used for the vertical axis). If that option is used, the plot will show that exact vertical range, independently of the values reached by the plot. If the vertical range is not specified, it will be set up according to the minimum and maximum values of the second coordinate of the plot points.

All other options should also be lists, starting with a keyword and followed by one or more values. See plot_options. If there are several plots to be plotted, a legend will be written to identify each of the expressions. The labels that should be used in that legend can be given with

the option legend. If that option is not used, Maxima will create labels from the expressions or function names.

Example: A basic plot.

```
(%i1) plot2d(sin(x), [x, -%pi, %pi])$
```

Example: The aspect ratio of the plot depends on the plotting program used. Compare the two plots. (In this example, option box is false, preventing a box from being drawn around the plot.)

```
(%i1) plot2d(x^2-1, [x, -3, 3], [y, -2, 10],
                  [box, false], [plot_format, xmaxima])$

(%i2) plot2d(x^2-1, [x, -3, 3], [y, -2, 10],
                  [box, false])$
```

Example: Plotting functions by name.

```
(%i1) f(x) := x^2 $
      g(x) := abs(x^3)$
      h(x) := if x < 0 then x^4 - 1 else 1 - x^5 $
      plot2d([f, g, h], [u, -1, 1], [y, -1.5, 1.5])$
```

Example: Two turns around a "circle," plotting only 7 points.

```
(%i1) plot2d([parametric, cos(t), sin(t),
        [t, -2*%pi, 2*%pi], [nticks, 8]])$
```

Example: Plot of a function together with the parametric representation of a circle. The size of the plot has been adjusted with the x and y options, to make the circle look round. These values work well on my screen; you might have to adjust the values for your screen.

```
(%i1) plot2d([[parametric, cos(t), sin(t), [t,0,2*%pi], [nticks, 80]],
        abs(x)], [x,-2.4,2.4], [y, -1.4, 1.4])$
      plot2d: some values were clipped.
```

Example: A plot of a discrete set of points, defining x and y coördinates separately:

```
(%i1) plot2d([discrete, [10, 20, 30, 40, 50], [.6, .9, 1.1, 1.3, 1.4]])$
```

Example: The same points shown in the previous example, defining each point separately and without any lines joining the points:

```
(%i1) plot2d([discrete, [[10, .6], [20, .9], [30, 1.1], [40, 1.3], [50, 1.4]]],
        [style, points])$
```

`plot3d(expr, x_range, y_range, options)` **Function**
`plot3d([expr_1,...,expr_n], x_range,y_range, options)` **Function**

Displays a plot of one or more surfaces defined as functions of two variables or in parametric form.

The functions to be plotted may be specified as expressions or function names. The mouse can be used to rotate the plot looking at the surface from different sides.

Example: Plot of a typical function:

```
(%i1) plot3d(2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2])$
```

Example: Avoiding excessively large values of z by choosing a grid that does not fall on any asymptotes. This example also shows how to select one of the predefined palettes, in this case the fourth one:

```
(%i1) plot3d(log (x^2*y^2), [x, -2, 2], [y, -2, 2],
      [grid, 29, 29],
      [palette, get_plot_option(palette,5)])$
```

Example: Two surfaces in the same plot, sharing the same domain. In gnuplot the two surfaces will use the same palette:

```
(%i1) plot3d([2^(-x^2 + y^2), 4*sin(3*(x^2+y^2))/(x^2+y^2),
        [x, -3, 3], [y, -2, 2]])$
```

Example: The same two surfaces, but now with different domains; in Xmaxima each surface will be plotted using a different palette, chosen from the list defined by the option palette:

```
(%i1) plot3d([[2^(-x^2 + y^2),[x,-2,2], [y,-2,2]],
        4*sin(3*(x^2+y^2))/(x^2+y^2), [x, -3, 3], [y, -2, 2]],
        [plot_format,xmaxima])$
```

Example: Plot of a Klein bottle, defined parametrically:

```
(%i1) expr_1: 5*cos(x)*(cos(x/2)*cos(y)+sin(x/2)*sin(2*y)+3.0)-10.0$
      expr_2:-5*sin(x)*(cos(x/2)*cos(y) + sin(x/2)*sin(2*y) + 3.0)$
      expr_3: 5*(-sin(x/2)*cos(y) + cos(x/2)*sin(2*y))$
      plot3d([expr_1, expr_2, expr_3], [x, -%pi, %pi], [y, -%pi, %pi],
        [grid, 40, 40])$
```

Example: Definition of a function of two variables using a matrix. Notice the single quote in the definition of the function, to prevent plot3d from failing when it realizes that the matrix will require integer indices:

```
(%i1) M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],
            [1, 2, 3, 3])$
      f(x, y) := float('M [round(x), round(y)])$
      plot3d(f(x,y), [x, 1, 4], [y, 1, 4], [grid, 4, 4])$
apply: subscript must be an integer; found: round(x)
```

spherical_to_xyz — System function

The system function spherical_to_xyz can be given as a value to the transform_xy option of plot3d. Its effect will be to interpret the two independent variables and the function in plot3d as the spherical coördinates of a point (first, the angle with the z-axis, then the angle of the x, y-projection with the x-axis, and finally the distance from the origin) and transform them into x-, y- and z-coördinates.

Example: Plot of a sphere using the transformation from spherical to rectangular coördinates. In Xmaxima the three axes are scaled in the same proportion, rendering the sphere symmetric, as it should be. A palette with different shades of a single color is used:

```
(%i1) plot3d(5, [theta, 0, %pi], [phi, 0, 2*%pi],
        [plot_format,xmaxima],
        [transform_xy, spherical_to_xyz],
        [palette,[value,0.65,0.7,0.1,0.9]])$
```

12.2 Options for plot

Each option used with plot functions is a list. The first entry in the list is one of the keywords found in this section of this document. At least one more entry must appear in the list: an allowable value for the option named by the keyword. Additional allowable values may follow. Most of the options can be used in any of the plotting commands (plot2d, plot3d, contour_plot, implicit_plot) or in the function set_plot_option. Exceptions are noted below.

adapth_depth [adapth_depth, integer] **Graphics option**

Default value: 5

The maximum number of splittings used by the adaptive plotting routine.

axes [axes, symbol] **Graphics option**

Default value: true

Allowable values: true, false, x, or y

If symbol is false, no axes will be shown; if equal to x or y only the x- or y-axis will be shown, respectively, and if it is equal to true, both axes will be shown. This option is used only by plot2d and implicit_plot.

azimut [azimuth, number] **Graphics option**

Default value: 30

See also: elevation

A plot3d plot can be thought of as starting with its x- and y-axis in the horizontal and vertical axis, as in plot2d, with the z-axis coming out of the paper perpendicularly. The z-axis is then rotated around the x-axis through angle elevation, and then the x, y- plane is rotated around the new z-axis through angle azimuth. This option sets the value of azimuth, in degrees.

See the example under elevation_plot

box [box, symbol] **Graphics option**

Default value: true

If box is set to true, a box will be drawn around the plot. If box is set to false, no box will be drawn.

Example: A basic plot, without the box:

```
(%i1) plot2d(sin(x), [x, -%pi, %pi],
        [box,false])$
```

color [color, color_1, ..., color_n] **Graphics option**

Default value: blue, red, green, magenta, black, cyan (in that order)

In plot2d and implicit_plot, it defines the color (or colors) for the various curves. In plot3d, it defines the colors used for the mesh lines of the surfaces, when no palette is being used; one side of the surface will have color color_1 and the other side will have color color_2 (or the same color if there is only one color).

If there are more curves or surfaces than colors, the colors will be repeated in sequence. When using gnuplot, the colors could be: blue, red, green, magenta, black, cyan or black; in Xmaxima, the colors can be those or an hexadecimal RGB color name. If color does not recognize the color name it's been given, it will use black instead.

colorbox [colorbox, symbol] **Graphics option**

Default value: false

Allowable values: false, true

If colorbox is true, then when plot3d uses a palette of different colors to represent the different values of z, a box will be shown on the right, indicating the colors used according to the scale of values of z. This option does not work in Xmaxima.

See the example under elevation_plot.

elevation [elevation, number] **Graphics option**

Default value: 60

See also: azimut

A plot3d plot can be thought of as starting with its x-and y-axis in the horizontal and vertical axis, as in plot2d, and the z-axis coming out of the paper perpendicularly. The z-axis is then rotated around the x-axis through angle elevation, and then the x, y-plane is rotated around the new z axis through angle azimuth. This option sets the value of elevation, in degrees.

Example: By setting the elevation equal to zero, a surface can be seen as a map in which each color represents a different level. The option colorbox is used to show the correspondence between colors and levels, and the mesh lines are disabled to make the colors easier to see.

```
(%i1) plot3d(cos (-x^2 + y^3/4), [x, -4, 4], [y, -4, 4],
        [mesh_lines_color, false],
        [elevation, 0],
        [azimuth, 0],
        [colorbox, true],
        [grid, 150, 150])$
```

get_plot_option(keyword, index) **Function**

See also: plot_options, set_plot_option, and Section 12.2, on options for plot

Returns a value of the option with name keyword, stored in the global variable plot_options. A value of 1 for the index will return the keyword itself; a value of 2 returns the first parameter following the keyword, and so on.

grid [grid, integer, integer] **Graphics option**

Default value: 30, 30

Sets the number of grid points to use in the x- and y-directions for three-dimensional plotting.

Example: Plot of a spherical harmonic, using the predefined transformation spherical_to_xyz to transform from spherical to rectangular coördinates. The grid option is used to get a plot that's easier to read:

```
(%i1) plot3d(sin(2*theta)*cos(phi), [theta, 0, %pi], [phi, 0, 2*%pi],\
        [transform_xy, spherical_to_xyz],
        [grid,30,60])$
```

legend [legend, string_1, ..., string_n] **Graphics option**

Default value: See the description, below.

legend [legend, false] **Graphics option**

Default value: See the description, below.

It specifies the labels for the plots when various plots are shown. If there are more plots than the number of labels given, they will be repeated. If given the value false, no legends will be shown. By default, the names of the expressions or functions will be used, or the words discrete1, discrete2, ..., for discrete sets of points. This option can not be set with set_plot_option.

logx [logx] **Graphics option**

Default value: false

logy [logy] **Graphics option**

Default value: false

Makes the horizontal (respectively, vertical) axes to be scaled logarithmically. It can not be used with set_plot_option.

Example: A plot with a logarithmic scale.

```
(%i1) plot2d(exp(3*s), [s, -2, 2], [logy])$
```

mesh_lines_color [mesh_lines_color, color] **Graphics option**

Default value: black

It sets the color used by plot3d to draw the mesh lines, when a palette is being used. It accepts the same colors as for the option color (see the list of allowed colors in color). It can also be given a value false to eliminate the mesh lines completely.

nticks [nticks, integer] **Graphics option**

Default value: 29

When plotting functions with plot2d, it is gives the initial number of points used by the adaptive plotting routine for plotting functions. When plotting parametric functions with plot2d or plot3d, it sets the number of points that will be shown for the plot.

palette [palette, [palette_1], ..., [palette_n]] **Graphics option**

Default value: For gnuplot: [hue, 0.25, 0.7, 0.8, 0.5]; for Xmaxima: [hue, 0.25, 0.7, 0.8, 0.5], [hue, 0.65, 0.8, 0.9, 0.55], [hue, 0.55, 0.8, 0.9, 0.4], [hue, 0.95, 0.7, 0.8, 0.5] (in that order)

palette [palette, false] **Graphics option**

Default value:

It can consist of one palette or a list of several palettes. Each palette is a list with a keyword followed by four numbers. The first three numbers, which must be between 0 and 1, define the hue, saturation and value of a basic color to be assigned to the minimum value of z. The keyword specifies which of the three attributes (hue, saturation or value) will be increased according to the values of z. The last number indicates the increase corresponding to the maximum value of z. That last number can be greater than 1 or negative; the corresponding values of the modified attribute will be rounded modulo 1.

gnuplot only uses the first palette in the list; Xmaxima will use the palettes in the list sequentially, when several surfaces are plotted together; if the set of palettes is exhausted, they will be repeated sequentially.

The color of the mesh lines will be given by the option mesh_lines_color. If palette is given the value false, the surfaces will not be shaded but represented with a mesh of curves only. In that case, the colors of the lines will be determined by the option color.

plot_format [plot_format, format] **Graphics option**

Default value: In Windows^{TM}: gnuplot; in other platforms: gnuplot_pipes

Allowable values: gnuplot, Xmaxima, gnuplot, gnuplot_pipes

This option sets the format to be used for plotting.

plot_options **System variable**

Default value:

See also: set_plot_option, get_option

Elements of this list set the default options for plotting. If an option is present in a plot2d or plot3d call, that value takes precedence over the default option. Otherwise, the value in

plot_options is used. Default options are assigned by set_plot_option. There are other local options specific to each plotting command, and not included in this list of global options.

Each element of plot_options is a list of two or more items. The first item is the name of the option, and the remainder comprises the value or values assigned to the option. In some cases, the assigned value is a list, which may include several items.

plot_real_part [plot_realpart, symbol] **Graphics option**

Default value: false

Allowable values: false, true

When set to true, the functions to be plotted will be considered as complex functions whose real value should be plotted; this is equivalent to plotting realpart(function). If set to false, nothing will be plotted when the function does not give a real value. For instance, when x is negative, log(x) gives a complex value, with real value equal to log(abs(x)); if plot_real_part were true, log(-5) would be plotted as log(5), while nothing would be plotted if plot_real_part were false.

point_type [point_type, type_1, ..., type_n] **Graphics option**

Default value: bullet, circle, plus, times, asterisk, box, square,triangle, delta, wedge, nabla, diamond, lozenge (in that order)

Allowable values: asterisk, box, bullet, circle, delta, diamond, lozenge, nabla, plus, square, times, triangle, wedge

In gnuplot, each set of points to be plotted with the style points or linespoints will be represented with objects taken from this list, in sequential order. If there are more sets of points than objects in this list, the point types will be repeated sequentially.

psfile [psfile, string] **Graphics option**

Default value:

Saves the plot into a Postscript™ file with name equal to string, rather than showing it in the screen. By default, the file will be created in the directory defined by the variable maxima_tempdir; the value of that variable can be changed to save the file in a different directory.

run_viewer [run_viewer, symbol] **Graphics option**

Default value: true

Allowable values: false, true

Controls whether or not the appropriate viewer for the plot format should be run.

set_plot_option(option) **Function**

See also: get_option, plot_options

Accepts most of the options listed in this subsection, and saves them to the global variableplot_options. set_plot_option evaluates its argument and returns the complete list plot_options (after modifying the option(s) given).

Example: Modification of the grid values.

```
(%i1) set_plot_option([grid, 30, 40]);
(%o1) [[t,-3,3],[grid,30,40],[transform_xy,false],[run_viewer,true],
      [axes,true],[plot_format,gnuplot_pipes],[color,blue,red,green,magenta,
          black,cyan],
      [point_type,bullet,circle,plus,times,asterisk,box,square,triangle,
          delta,wedge,nabla,diamond,lozenge],
      [palette,[hue,0.25,0.7,0.8,0.5],[hue,0.65,0.8,0.9,0.55],
          [hue,0.55,0.8,0.9,0.4],[hue,0.95,0.7,0.8,0.5]],
```

```
[gnuplot_term,default],[gnuplot_out_file,false],
[nticks,29],[adapt_depth,5],[gnuplot_preamble,],
[gnuplot_default_term_command,"set term pop"],
[gnuplot_dumb_term_command,"set term dumb 79 22"],
[gnuplot_ps_term_command,"set size 1.5, 1.5;
    set term postscript eps enhanced color solid 24"],
[plot_realpart,false]]
```

`style [style, type_1, ..., type1_n]` **Graphics option**

Default value: `lines`

`style [style, [style_1], ..., [style_n]]` **Graphics option**

Default value: `lines`

Allowable values: `dots`, `lines`, `linespoints`, `points`; in gnuplot, also `impulses`

See also: `color`, `point_type`

The styles that will be used for the various functions or sets of data in a 2d plot. The word `style` must be followed by one or more styles. If there are more functions and data sets than the styles given, the styles will be repeated. Each style can be either `lines` for line segments, `points` for isolated points, `linespoints` for segments and points, or `dots` for small isolated dots. gnuplot accepts also an `impulses` style.

Each of the styles can be enclosed inside a list with some additional parameters. `lines` accepts one or two numbers: the width of the line and an integer that identifies a color. The default color codes are: 1: blue, 2: red, 3: magenta, 4: orange, 5: brown, 6: lime and 7: aqua. If you use gnuplot with a terminal different than X11, those colors might be different; for example, if you use the option `[gnuplot_term,ps]`, color index 4 will correspond to black, instead of orange.

`points` accepts one, two, or three parameters. The first parameter is the radius of the points. The second parameter is an integer that selects the color, using the same codes used for `lines`. The third parameter is currently used only by gnuplot, and it corresponds to several objects instead of points. The default types of objects are: 1: filled circles, 2: open circles, 3: plus signs, 4: x, 5: *, 6: filled squares, 7: open squares, 8: filled triangles, 9: open triangles, 10: filled inverted triangles, 11: open inverted triangles, 12: filled lozenges and 13: open lozenges.

`linespoints` accepts up to four parameters: line width, points radius, color and type of object to replace the points. Its default value is `lines` (will plot all sets of points joined with lines of thickness 1 and the first color given by the option `color`).

`t [t, min, max]` **Graphics option**

Default value: `-3, 3`

Parameter range for parametric plots.

Example: A plot of the butterfly curve, defined parametrically.

```
(%i1) r: exp(cos(t))-2*cos(4*t)-sin(t/12)^5$
      plot2d([parametric, r*sin(t), r*cos(t),
        [t, -8*%pi, 8*%pi], [nticks, 2000]])$
```

`transform_xy [transform_xy, symbol]` **Graphics option**

Default value: `false`

Allowable values: Either `false` or the result obtained by using the function `transform_xy`

See also: `make_transform`, `polar_to_xy` and `spherical_to_xyz`

If different from `false`, it will be used to transform the 3 coördinates in `plot3d`.

`x [x, min, max]` **Graphics option**

Default value:
When used as the first option in a 2d-plotting command (or any of the first two in plot3d), it indicates that the first independent variable is x and it sets its range. It can also be used again after the first option (or after the second option in plot3d) to define the effective horizontal domain that will be shown in the plot.

xlabel [xlabel, string] — Graphics option

Default value:
Specifies the string that will label the first axis; if this option is not used, that label will be:

- "x," when plotting functions with plot2d or implicit_plot,
- the name of the first variable, when plotting surfaces with plot3d or contours with contour_plot, or
- the first expression in the case of a parametric plot.

xlabel can not be used with set_plot_option.

Example: A plot of experimental data points together with the theoretical function that predicts the data:

```
(%i1) xy: [[10, .6], [20, .9], [30, 1.1], [40, 1.3], [50, 1.4]]$
      plot2d([[discrete, xy], 2*%pi*sqrt(l/980)], [l,0,50],
        [style, points, lines],
        [color, red, blue],
        [point_type, asterisk],
        [legend, "Experiment", "Theory"],
        [xlabel, "Length of pendulum (cm)"],
        [ylabel, "Period(s)"])$
```

y [y, min, max] — Graphics option

Default value:
When used as one of the first two options in plot3d, it indicates that one of the independent variables is y and it sets its range. Otherwise, It defines the effective domain of the second variable that will be shown in the plot.

Example: If the function to be plotted grows too quickly, it might be necessary to limit the values in the vertical axis using the y option.

```
(%i1) plot2d(sec(x), [x, -2, 2], [y, -20, 20])$
plot2d: some values were clipped.
```

ylabel [ylabel, string] — Graphics option

Default value:
Specifies the string that will label the second axis. If this option is not used, that label will be:

- "y", when plotting functions with plot2d or implicit_plot,
- the name of the second variable, when plotting surfaces with plot3d or contours with contour_plot, or
- the second expression in the case of a parametric plot.

Option ylabel can not be used with set_plot_option.
See the example under xlabel.

z [z, min, max] **Graphics option**
Default value:
Used in plot3d to set the effective range of values of z that will be shown in the plot.

Example: Use of the z option to rein in the graph an unbounded function. (In this case the function is minus infinity on the x- and y- axes). This example also shows how to plot with only lines and no shading:

```
(%i1) plot3d(log (x^2*y^2 ), [x, -2, 2], [y, -2, 2], [z, -8, 4],
           [palette, false],
           [color, magenta, blue])$
```

zlabel [zlabel, string] **Graphics option**
Default value: See the description, below.
This option specifies the string that will label the third axis, when using plot3d. If this option is not used, that label will be:

- "z," when plotting surfaces, or
- the third expression in the case of a parametric plot.

xlabel can not be used with set_plot_option, and it will be ignored by plot2d and implicit_plot.

12.3 Options for gnuplot

There are several plot options specific to gnuplot. All of them consist of a keyword (the name of the option), followed by a string that must be a valid gnuplot command, to be passed directly to gnuplot. In most cases, there exists a corresponding plotting option that will produce a similar result and whose use is recommended over that of the gnuplot-specific option.

gnuplot_term **Graphics option**
Default value: default
Allowable values: default, dumb, jpeg, png, ps, svg,or any other valid gnuplot graphical format
Sets the output terminal type for gnuplot, as described below:
default: gnuplot output is displayed in a separate graphical window.
dumb: gnuplot output is displayed in the Maxima console by an "ASCII art" approximation to graphics.
ps: gnuplot generates commands in the Postscript^{TM} page description language. If the option gnuplot_out_file is set to filename, gnuplot writes the Postscript^{TM} commands to the file whose name is filename. Otherwise, it is saved as maxplot.ps.
Any other valid gnuplot term specification: gnuplot can generate output in many other graphical formats such as png, jpeg, svg etc. To create plots in any of these formats the gnuplot_term can be set to any supported gnuplot term name (symbol) or even to a full gnuplot term specification with any valid options (string). For example, [gnuplot_term,png] creates output in png (Portable Network Graphics) format while [gnuplot_term,"png size 1000,1000"] creates a png of size 1000 pixels by 1000 pixels. If the option gnuplot_out_file is set to filename, gnuplot writes the output to file filename. Otherwise, the output is saved to a file called maxplot.term, in which "term" is the name of the gnuplot terminal used.

gnuplot_out_file **Graphics option**

Default value: `file_name`

When used in conjunction with the gnuplot_term option, `gnuplot_out_file` saves the plot in the file called `file_name`, in one of the graphic formats supported by gnuplot. If you want to create a Postscript™ file, you can use the option `psfile` instead, which will also work in Xmaxima (or, Openmath), and does the same thing with just one option.

gnuplot_pm3d **Graphics option**

Default value: `true`

With a value of `false`, this option prevents the usage of `pm3d` mode, which is enabled by default.

gnuplot_preamble **Graphics option**

Default value: (the empty string)

Allowable values: Any valid collection of valid gnuplot commands

Inserts gnuplot commands before the plot is drawn. Multiple commands should be separated by semicolons. The example shown produces a plot with a logarithmic scale.

[Author's note: Please provide the example.]

gnuplot_curve_titles **Graphics option**

Default value:

Deprecated. Superceded by `legend`, described in the preceding subsection.

gnuplot_curve_styles **Graphics option**

Default value:

Deprecated. Superceded by `style`, described in the preceding subsection.

gnuplot_default_term_command **Graphics option**

Default value: `set term pop`

The gnuplot command to set the terminal type for the `default` terminal.

gnuplot_dumb_term_command **Graphics option**

Default value: `set term dumb 79 22`

This is the gnuplot command used to set the terminal type for the `dumb` terminal. The default value of `set term dumb 79 22` makes the text output 79 characters by 22 characters.

gnuplot_ps_term_command **Graphics option**

Default value: `set size 1.5, 1.5; set term postscript eps enhanced color solid 24`

This is the gnuplot command used to set the terminal type for the `PostScript`™ terminal. The default value of `set size 1.5, 1.5; set term postscript eps enhanced color solid 24` sets the terminal size to 1.5 times gnuplot's default, and the font size to 24, among other things. See the gnuplot documentation for `set term postscript` for more information.

12.4 Functions for gnuplot_pipes

gnuplot_start() **Function**

Opens the pipe to gnuplot used for plotting by the gnuplot format. It is not necessary to manually open the pipe before plotting.

gnuplot_close() **Function**

Closes the pipe to gnuplot used by the gnuplot_pipes format.

`gnuplot_restart()` **Function**

Closes the pipe to gnuplot used by the gnuplot_pipes format and opens a new pipe.

`gnuplot_replot(command_string)` **Function**

Allowable values: Any valid gnuplot command, including the empty string

Updates the gnuplot window. If `gnuplot_replot` is called with a gnuplot command in a string `command_string`, then `command_string` is sent to gnuplot before replotting the window.

`gnuplot_reset()` **Function**

Resets the state of gnuplot used by the gnuplot_pipes format. To update the gnuplot window, call `gnuplot_replot` after `gnuplot_reset`.

Chapter 13

The plotdf package

(Note to programmers and system administrators: plotdf requires Xmaxima. It can be used from the console or any other interface to Maxima (including wxMaxima), but the resulting file will be sent to Xmaxima for plotting. Please make sure Xmaxima is installed before using plotdf.)

plotdf is a third-party package for Maxima that draws direction fields (slope fields) of systems of two first-order ordinary differential equations (ODE's). It can also plot the direction field of a single first-order ODE and the phase portraits of some second-order ODE's. The main function provided by the plotdf package is the plotdf function. Since this is an additional package, in order to use it you must first load it with load(plotdf).

To plot the direction field of a single ODE, you must write the ODE in the form $\frac{dy}{dx} = F(x, y)$ and use the formula for the function F as the first argument of plotdf. If the independent and dependent variables are not x, and y, as in the equation above, must be named explicitly in a list given as an argument to the plotdf command. (See the examples.)

To plot the direction field of a set of two autonomous ODE's, you must write the ODE's in the form $\frac{dx}{dt} = F(x, y)$, $\frac{dy}{dt} = G(x, y)$ and use a list containing the formulas for $F(x, y)$ and $G(x, y)$ (in that order) as the first argument of plotdf. The first expression in the list will be taken to be the time derivative of the variable represented on the horizontal axis, and the second expression will be the time derivative of the variable represented on the vertical axis. These two variables do not have to be x and y, but if they are not, then you must include among the options a list naming the two variables, first the one for the horizontal axis and then the one for the vertical axis. (See the examples.)

If only one ODE is given, plotdf will assume $x = t$ and $F(x, y) = 1$, transforming the non-autonomous equation into a system of two autonomous equations. A second-order equation $ax'' + bx' + cx = f(x)$, $a \neq 0$, can also be represented, by using a system of the form [y, F(x,y)]. This is done by setting $y = x'$, which gives $ay' + by + cx = f(x)$, or $y' = f(x) - (by + cx)/a$. For the list that defines the system, use [y, f(x)-(by + cx)/a]. (See the example under sliders.)

plotdf can plot solution curves, as well. The Adams-Moulton method is used for the integration; it is also possible to switch to an adaptive 4^{th} order Runge-Kutta method.[1]

13.1 The plotdf command

plotdf(dy/dx, options) **Function**

plotdf(dv/du, [u,v], options) **Function**

plotdf(dv/du, [u,v], [u,uMin, uMax], [v,vMin, vMax], options) **Function**

[1] Unfortunately, I have not been able to find or figure out how.

`plotdf([dx/dt, dy/dt], options)` **Function**

`plotdf([du/dt, dv/dt], [u,v], options)` **Function**

`plotdf([du/dt, dv/dt], [u,v], [u,uMin, uMax], [v,vMin, vMax], options)` **Function**

`plotdf` displays the direction field in two dimensions x and y. `dy/dx`, `dx/dt` and `dy/dt` are expressions that depend on `x` and `y`. `dv/du`, `du/dt` and `dv/dt` are expressions that depend on `u` and `v`. In addition to those two variables, the expressions can also depend on a set of parameters, via the `parameters` option, or with a range of allowed values specified by the `sliders` option.

Example: A basic example using a single ODE: $y' - y = e^x$, which is rewritten as $y' = e^x + y$.

```
(%i1) load(plotdf)$
      plotdf(exp(-x)+y)$
```

Example: The same direction field, but with the ODE written as a system of autonomous ODE's. Note that the variable t is the same as the variable x of the previous example.

```
(%i1) load(plotdf)$
      plotdf([t,exp(-t)+y], [t,y])$
```

Example: Renaming and assigning the ranges of the vertical and horizontal variables.

```
(%i1) load(plotdf)$
      plotdf([v,-k*z/m], [z,v],
        [z, -5,5],
        [v, -3,3],
        [parameters,"m=2,k=2"],
        [trajectory_at,3,0])$
```

13.2 Graphics objects for `plotdf`

`sliders` **Graphics object**

`sliders` defines a list of parameters that will be changed interactively using slider buttons, and the range of variation of those parameters. The names and ranges of the parameters must be given in a string with a comma-separated sequence of elements `name = min :  max`

Example: Sliders. The system of ODE's represents The direction field of the Duffing equation $mx'' + cx' + kx + bx^3 = 0$, by introducing the variable $y = x'$.

```
(%i1) load(plotdf)$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
        [parameters, "k=-1,m=1.0,c=0,b=1"], /*NO SPACES BETWEEN PARAMETERS!*/
        [sliders, "k=-2:2,m=-1:1"],
        [tstep,0.1])$
```

Example: Sliders again. The system represents a harmonic oscillator, defined by the two equations $dz/dt = v$ and $dv/dt = -kz/m$. The slider allows you to change the value of m interactively, leaving k fixed at 2.

```
(%i1) load(plotdf)$
      plotdf([v,-k*z/m], [z,v],
        [parameters, "m=2,k=2"],
        [sliders, "m=1:5"],
        [trajectory_at, 6,0])$
```

trajectory_at **Graphics object**

trajectory_at defines the coördinates xinitial and yinitial for the starting point of an integral curve.

Example: A direction field with a trajectory. An asymptote to the trajectory is also plotted.

```
(%i1) load(plotdf)$
      plotdf(x-y^2, [y,-5,5], [x,-4,16],
        [xfun, "sqrt(x);-sqrt(x)"],
        [trajectory_at, -1,3])$
```

xfun **Graphics object**

xfun defines a string with semicolon-separated sequence of functions of x to be displayed, on top of the direction field. (Those functions will be parsed by Tcl and not by Maxima.)

Example: xfun used to add an asymptote of a trajectory to a direction field.

```
(%i1) load(plotdf)$
      plotdf(x-y^2, [xfun, "sqrt(x)"],
        [trajectory_at, -1,3],
        [y,-5,5], [x,-4,16])$
```

Example: The same example, modified to show the syntax for plotting more than one additional function.

```
(%i1) load(plotdf)$
      plotdf(x-y^2, [xfun,  "sqrt(x);-sqrt(x)"],
        [trajectory_at, -1,3],
        [y,-5,5], [x,-4,16])$
```

13.3 Options for plotdf

Options for plotdf can be given within the command or selected in the configuration menu of the plot window. Integral curves can be obtained by clicking on the plot, or with the option trajectory_at. The direction of the integration can be controlled with the direction option, which can have values of forward, backward or both. The number of integration steps is given by nsteps and the time interval between them is set using tstep option.

direction **Graphics option**

Default value: both

Allowable values: backward, both, forward

direction defines the direction of the independent variable that will be followed to compute an integral curve. Option value forward makes the independent variable increase nsteps times, with increments tstep; backward makes the independent variable decrease; or both, which leads to an integral curve that extends nsteps forward, and nsteps backward.

Example: Different directions. Compare the effect of the default value both of direction with that of the value forward.

```
(%i1) load(plotdf)$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
        [parameters, "k=-1,m=1.0,c=0,b=1"],
        [trajectory_at, 2,2])$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
```

```
        [direction, forward],
        [parameters, "k=-1,m=1.0,c=0,b=1"],
        [trajectory_at, 2,2])$
```

Author's note: There seems to be a bug in the code for this option. Giving it the value `backward` or the value `both` seems to have the same effect as giving it the value `forward`. However, changing the `direction` in the `configuration` menu of the plot window works just fine.

nsteps — Graphics option

Default value: `100`

Allowable values: Any floating-point number

`nstep` defines the number of steps of length `tstep` that will be used for the independent variable, to compute an integral curve. If `nsteps` is 0 or negative, no integral curve is plotted.

Example: Differing values of `nsteps`.

```
(%i1) load(plotdf)$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
        [nsteps,20],
        [parameters, "k=-1,m=1.0,c=0,b=1"],
        [trajectory_at, 2,2])$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
        [nsteps,60],
        [parameters, "k=-1,m=1.0,c=0,b=1"],
        [trajectory_at, 2,2])$
```

parameters([``name1=value1, name2=value2,...,nameN=valueN'']) — Graphics option

Default value: (empty)

Use this option to set numeric values of paramters used in the definition of differential equations for use with `plotdf`. The names and values of the parameters must be given in a string with a comma-separated sequence of pairs `name = value`. This option is particularly useful when combined with the `sliders` graphics object.

Example: A simple harmonic oscillator.

```
(%i1) load(plotdf)$
      plotdf([v,-k*z/m], [z,v],
        [parameters,``m=2,k=2''],
        [trajectory_at,6,0])$
```

tinitial — Graphics option

Default value: `0`

Allowable values: Any floating-point number

`tinitial` defines the initial value of variable `t` used to compute integral curves. Since the differential equations are autonomous, that setting will only affect in the plot of the curves as functions of `t`.

Example: Different values of `tinitial`.

```
(%i1) load(plotdf)$
      plotdf([y,-(k*x + c*y + b*x^3)/m],
        [tinitial, 0],
        [nsteps, 60],
```

```
    [parameters, "k=-1,m=1.0,c=0,b=1"],
    [trajectory_at, 2,2])$
plotdf([y,-(k*x + c*y + b*x^3)/m],
    [tinitial, 5],
    [nsteps, 60],
    [parameters, "k=-1,m=1.0,c=0,b=1"],
    [trajectory_at, 2,2])$
```

Author's note: There seems to be a bug in the code for this option. Changing the value of `tinitial` seems to have no effect.

`tstep` **Graphics option**

Default value: `0.1`

Allowable values: Any floating-point number

`tstep` defines the length of the increments of the independent variable `t`, used to compute an integral curve. If only one expression `dy/dx` is given to `plotdf`, the variable `x` will be directly proportional to `t`. If `tstep` is 0 or negative, no integral curve is plotted.

13.4 plotdf's plot window

When `plotdf` renders a direction field, it does not stop running. The plot window is interactive in four ways. First, if you click anywhere in the direction field, `plotdf` plots a solution curve through the point on which you've clicked. Second, you can drag the plot around in the plot window by holding down the right mouse button and moving the cursor. Third, if you use the option `sliders`, `plotdf` provides a button bar in the plot window.

In order from left to right, the buttons are `close`, `configure`, `replot`, `save`, `zoom in`, `zoom out`, and `versus_t`. `close`, `replot`, `save`, `zoom in`, and `zoom out` behave exactly as one might expect, except that `replot` replots only the most recently plotted solution curve. `save` saves the plot in the encapsulated Postscript™ format. Clicking `versus_t` opens the `versus_t` window. (See `versus_t` for details.)

Clicking `configure` causes the `Plot Setup` dialog to open. If your direction field is based on a single ODE, the plot settings for a single ODE will be presented; otherwise the settings for an autonomous system. You can change which settings are presented by clicking either `dy/dx` or `dy/dt,dx/dt`, as desired. If you change any settings, you'll need to click `replot` (after closing the `Plot Setup` dialog) to cause the new settings to take effect.

Here are the available settings, in order as they appear in the dialog:

- Either `dy/dx` or `dy/dt` and `dx/dt`, according to which of these you have selected. This gives you a place to edit your ODE or your system of ODE's.
- `Trajectory at`: To add a trajectory to the plot, enter coördinates for the desired point and press the `Enter` key.
- `fieldlines`: Sets the color of the trajectories. If empty, no trajectories are plotted.
- `vectors`: Sets the color of the vectors that make up the plot of the direction field. If empty, the direction field is not plotted.
- `curves`: Sets the color of the orthogonal trajectories of the direction field. If empty, no orthogonal trajectories are plotted.
- `direction`: Sets the direction in which trajectories are plotted. See `direction` for details.

- nsteps: Sets the number of time steps used by the numerical integrator in plotting trajectories. See nsteps for details.
- versus_t: Sets the versus_t window state. "1" is for "display the versus_t window" and "0" is for "do not display the versus_t window. See versus_t for details.
- tinitial: Sets the initial time for use by the numerical integrator. See tinitial for details. **Author's note:** This option does not seem to work on my system; perhap's there's a bug in the code.
- ycenter: Sets the center of the vertical range of the plot. The range actually used is [ycenter − yradius, ycenter + yradius]. If you set the vertical range in the wxMaxima session window by using [y, y_min, y_max] as an option in the plotdf command, then ycenter is $(y\_min + y\_max)/2$.
- xcenter: Sets the center of the horizontal radius of the plot. The range actually used is [xcenter − xradius, xcenter + xradius]. If you set the horizontal range in the wxMaxima session window by using [x, x_min, x_max] as an option in the plotdf command, then xcenter is $(x\_min + x\_max)/2$.
- yradius: Sets the radius of the vertical range of the plot. The range actually used is [ycenter − yradius, ycenter + yradius]. If you set the vertical range in the wxMaxima session window by using [y, y_min, y_max] as an option in the plotdf command, then yradius is $(y\_min - y\_max)/2$.
- xradius: Sets the radius of the horizontal range of the plot. The range actually used is [xcenter − xradius, xcenter + xradius]. If you set the horizontal range in the wxMaxima session window by using [x, x_min, x_max] as an option in the plotdf command, then xradius is $(x\_min - x\_max)/2$.
- line_width: Sets the width of the plotted trajectory curves. Allowable values are non-negative floating-point numbers. However, values below a certain threshold are ignored.[2]
- xfun: Includes functions in the plot, without requiring them to be trajectories or orthogonal trajectories of the direction field. See xfun for details.
- parameters: Sets the values of any optional parameters that may have been included in the dy/dx, dy/dt, or dx/dt fields. See parameters for details.

Note: Bear in mind that if you change any of the settings in the Plot Setup dialog, you need to click replot to cause the new settings to take effect.

13.5 The versus_t window

versus_t **Graphics option**

Default value: 0

Allowable values: Any floating-point number

versus_t is used to create a second plot window, with a plot of an integral curve, as two functions x and y of the independent variable t. If versus_t is given any value different from 0, this second plot window will be displayed; a buttonbar is included.

In order from left to right, the buttons are close, configure, replot, save, zoom in, zoom out, and help. close, replot, save, zoom in, zoom out, and help behave exactly as one might

[2] I do not know what the threshold value is.

expect, except that replot replots only the most recently plotted solution curve. save saves the plot in the encapsulated Postscript™ format.

Clicking configure causes the Plot Setup dialog to open. If your direction field is based on a single ODE, the plot settings for a single ODE will be presented; otherwise the settings for an autonomous system. You can change which settings are presented by clicking either dy/dx or dy/dt,dx/dt, as desired. If you change any settings, you'll need to click replot, after closing the Plot Setup dialog, to cause the new settings to take effect.

Here are the available settings, in order as they appear in the dialog:

- y=f(x): Allows plotting of curves without requiring them to be x- or y-components of solution curves. See xfun for details.
- Number of mesh grids: Sets the mesh size for plotting the functions given in the y=f(x) option. Must be a (possibly floating-point) number greater than or equal to 3.
- parameters: Sets the values of any optional parameters that may have been included in the dy/dx, dy/dt, or dx/dt fields. See parameters for details.
- line_width: Sets the width of the plotted trajectory curves. Allowable values are non-negative floating-point numbers. However, values below a certain threshold are ignored, and the threshold value is used.[3]
- autoscale: If set to x, causes the horizontal axis to be scaled automatically for the given yrange. If set to y, causes the vertical axis to be scaled automatically, according to the xrange. The default value is x.
- linecolors: Sets the colors of curves drawn in the versus_t window. Colors are used in the order listed, can be given by names or in hexadecimal RGB format, and must be separated by spaces.
- ycenter: This option sets the center of the vertical range of the plot. If autoscale is set to y, the value of ycenter is calculated by versus_t. Otherwise, it is calculated from the vertical range [y, y_min, y_max] given either by default or as an option in the plotdf command. In any case, the value of ycenter is $(y\_min + y\_max)/2$ and the vertical range actually used is [ycenter − yradius, ycenter + yradius].
- xcenter: This option sets the center of the horizontal range of the plot. If autoscale is set to x, the value of xcenter is calculated by versus_t. Otherwise, it is calculated from the horizontal range [x, x_min, x_max] given either by default or as an option in the plotdf command. In any case, the value of xcenter is $(x\_min + x\_max)/2$ and the horizontal range actually used is [xcenter − xradius, xcenter + xradius].
- yradius: This option sets the radius of the vertical range of the plot. If autoscale is set to y, the value of yradius is calculated by versus_t. Otherwise, it is calculated from the vertical range [y, y_min, y_max] given either by default or as an option in the plotdf command. In any case, the value of yradius is $(y\_min - y\_max)/2$ and the vertical range actually used is [ycenter − yradius, ycenter + yradius].
- xradius: This option sets the radius of the horizontal range of the plot. If autoscale is set to x, the value of xradius is calculated by versus_t. Otherwise, it is calculated from the horizontal range [x, x_min, x_max] given either by default or as an option in the plotdf command. In any case, the value of xradius is $(x\_min - x\_max)/2$ and the horizontal range actually used is [xcenter − xradius, xcenter + xradius].

[3] I do not know what the threshold value is.

Note: Bear in mind that if you change any of the settings in the `Plot Setup` dialog, you need to click `replot` to cause the new settings to take effect.

Example: Displaying the `versus_t` window.

```
(%i1) load(plotdf)$
      plotdf([w,-g*sin(a)/l - b*w/m/l], [a,w],
        [a,-10,2],
        [w,-14,14],
        [parameters, "g=9.8,l=0.5,m=0.3,b=0.05"],
        [nsteps,300],
        [tstep, 0.01],
        [sliders, "m=0.1:1"],
        [trajectory_at, 1.05,-9],
        [verus_t, 1])$
```

Author's note: On my system, the `versus_t` window is displayed first, but disappears as the window containing the direction field is displayed, leaving an icon for the `versus_t` window in the taskbar. However, if I try to display the `versus_t` window by clicking the appropriate button in the plot window, `versus_t` works just fine. I believe this is an issue with my operating system (Ubuntu 11), and would appreciate feedback on how well the `versus_t` option works in other operating systems.

Chapter 14

The worldmap package

Package worldmap creates maps and has features intended for making maps of regions of Earth. However, it can be used for other things. Boundaries between geopolitical entities are made of line segments, in *polygon* paths. This package automatically loads Package draw. Objects and functions associated with picture are discussed in this section.

boundaries_array **Global variable**

Default value: false

See also: geomap

boundaries_array is where the graphics object geomap looks for coördinates of points that define boundaries. Each component of boundaries_array is an array of floating-point numbers representing the coördinates of a polygonal segment or map boundary.

geomap(numlist) **Graphics object**

geomap(numlist, 3Dprojection) **Graphics object**

Options: color, line_type, line_width

Draws cartographic maps in 2D and 3D. This function works together with global variable boundaries_array. Argument numlist is a list containing numbers or lists of numbers. All these numbers must be integers greater or equal than zero, representing the components of global array boundaries_array. Each component of boundaries_array is an array of floating-point numbers, the coördinates of a polygonal segment or map boundary. geomap(numlist) flattens its arguments and draws the associated boundaries in boundaries_array.

2D

Example: A simple map defined by hand.

```
(%i1) load(worldmap)$
      /* Vertices of boundary }0: {(1,1), (2,5), (4,3)} */
      (bnd0: make_array(flonum,6),
        bnd0[0] : 1.0,    bnd0[1] : 1.0,    bnd0[2] : 2.0,
        bnd0[3] : 5.0,    bnd0[4] : 4.0,    bnd0[5] : 3.0)$
      /* Vertices of boundary #1: {(4,3), (5,4), (6,4), (5,1)} */
      (bnd1: make_array(flonum,8),
        bnd1[0] : 4.0,    bnd1[1] : 3.0,    bnd1[2] : 5.0,    bnd1[3] : 4.0,
        bnd1[4] : 6.0,    bnd1[5] : 4.0,    bnd1[6] : 5.0,    bnd1[7] : 1.0)$
      /* Vertices of boundary #2: {(5,1), (3,0), (1,1)} */
      (bnd2: make_array(flonum,6),
        bnd2[0] : 5.0,    bnd2[1] : 1.0,    bnd2[2] : 3.0,
```

```
  bnd2[3] : 0.0,    bnd2[4] : 1.0,    bnd2[5] : 1.0)$
/* Vertices of boundary #3: {(1,1), (4,3)} */
(bnd3: make_array(flonum,4),
  bnd3[0] : 1.0,    bnd3[1] : 1.0,    bnd3[2] : 4.0,    bnd3[3] : 3.0)$
/* Vertices of boundary #4: {(4,3), (5,1)} */
(bnd4: make_array(flonum,4),
  bnd4[0] : 4.0,    bnd4[1] : 3.0,    bnd4[2] : 5.0,    bnd4[3] : 1.0)$
/* Pack all together in boundaries_array */
(boundaries_array: make_array(any,5),
  boundaries_array[0] : bnd0,     boundaries_array[1] : bnd1,
  boundaries_array[2] : bnd2,     boundaries_array[3] : bnd3,
  boundaries_array[4] : bnd4)$
draw2d(geomap([0,1,2,3,4]))$
```

Package `worldmap` sets global variable boundaries_array equal to real world boundaries in terms of longitude and latitude. These data are in the public domain, but I no longer have a working URL for them.Package `worldmap` also defines boundaries for countries, continents and coastlines as lists with the necessary components of boundaries_array.[1]

Example: Drawing Africa and selected countries, by name.

```
(%i1) load(worldmap)$
      c1: gr_2d(geomap([Canada, United_States, Mexico,Cuba]))$
      c2: gr_2d(geomap([Africa]))$
      c3: gr_2d(geomap([Oceania, China, Japan]))$
      c4: gr_2d(geomap([France, Portugal, Spain, Morocco, Western_Sahara]))$
      draw(columns = 2, c1,c2,c3,c4)$
```

Package `worldmap` is also useful for plotting countries as polygons. In this case, graphics object geomap is no longer necessary and the polygon object is used instead. Since lists are now used and not arrays, map-rendering will be slower.

Example: Plotting a country as a polygon. (See also make_poly_country and make_poly_continent to understand the following code.)

```
(%i1) load(worldmap)$
      my_map: append([color  = white], /* borders are white */
                  [fill_color = red],           make_poly_country(Bolivia),
                  [fill_color = cyan],          make_poly_country(Paraguay),
                  [fill_color = green],         make_poly_country(Colombia),
                  [fill_color = blue],          make_poly_country(Chile),
                  [fill_color = "#23ab0f"],     make_poly_country(Brazil),
                  [fill_color = goldenrod],     make_poly_country(Argentina),
                  [fill_color = midnight-blue], make_poly_country(Uruguay))$
      apply(draw2d, my_map)$
```

3D

`geomap(numlist)` projects map boundaries on the sphere of radius 1 centered at (0,0,0). It is possible to change the sphere or the projection type by using `geomap(numlist,3Dprojection)`. The available 3D projections are:

- `[spherical_projection, x,y,z, r]`, which projects map boundaries on the sphere of radius r centered at (x, y, z). This is the default projection.

[1] See file share/draw/worldmap.mac for more information.

- `[conic_projection, x,y,z, r, alpha]`, which re-projects spherical map boundaries on the cones of angle `alpha`, with axis passing through the poles of the globe of radius r centered at (x, y, z). Both the northern and southern cones are tangent to the sphere.
- `[cylindrical_projection, x,y,z, r,rc]`, which re-projects spherical map boundaries on the cylinder of radius rc and axis passing through the poles of the globe of radius r centered at (x, y, z).

Example: Spherical projection.

```
(%i1) load(worldmap)$
      draw3d(geomap(Australia), /* default projection */
         geomap(Australia, [spherical_projection, 2,2,2, 3]))$
```

Example: Conic projection.

```
(%i1) load(worldmap)$
      draw3d(geomap(World_coastlines,
        [conic_projection, 0,0,0, 1, 90]))$
```

Example: Cylindrical projection.

```
(%i1) load(worldmap)$
      draw3d(geomap([America_coastlines,Eurasia_coastlines],
        [cylindrical_projection, 2,2,2, 3,4]))$
```

See also http://www.telefonica.net/web2/biomates/maxima/gpdraw/geomap for more elaborate examples.[2]

`numbered_boundaries(nlist)` **Function**

Draws a list of polygonal segments (boundaries), labeled by its numbers (`boundaries_array` coördinates). This is of great help when building new geographical entities.

`make_poly_continent(continent_name)` **Function**
`make_poly_continent(country_list)` **Function**

These functions make the necessary polygons to draw a colored continent or the countries in a list.

Example: Drawing a continent and a list of countries.

```
(%i1) load(worldmap)$
      /* A continent */
      make_poly_continent(Africa)$
      apply(draw2d, %)$

      /* A list of countries */
      make_poly_continent([Germany, Denmark, Poland])$
      apply(draw2d, %)$
```

`make_poly_continent(continent_name)` **Function**
`make_poly_country(country_name)` **Function**

`make_poly_country` makes the necessary polygons to draw a single country. The country can be defined with more than just one polygon, to accommodate the mapping of islands.

Example: India.

[2] This URL worked as recently as 2012-02-02.

```
(%i1) load(worldmap)$
      make_poly_country(India)$
      apply(draw2d, %)$
```

make_polygon(nlist) **Function**

Returns a polygon object from boundary indices. Argument nlist is a list of components of boundaries_array.

For example, Bhutan is defined by boundary numbers 171, 173 and 1143, so the command make_polygon ([171,173,1143]) appends arrays of coördinates boundaries_array[171], boundaries_array[173], and boundaries_array[1143] and returns a polygon object suitable for plotting by draw. To avoid an error message, arrays must be compatible in the sense that any two consecutive arrays have two coördinates in the extremes in common. In this example, the two first components of boundaries_array[171] are equal to the last two coördinates of boundaries_array[173], and the two first of boundaries_array[173] are equal to the two first of boundaries_array[1143]; therefore, boundary numbers 171, 173 and 1143 (in this order) are compatible and the colored polygon can be drawn.

Example: Bhutan.

```
(%i1) load(worldmap)$
      Bhutan;
(%o2) [[171,173,1143]]

(%i3) array_171 : boundaries_array[171];
      array_173 : boundaries_array[173];
      array_1143 : boundaries_array[1143];
(%o3) Lisp array [8]
(%o4) Lisp array [78]
(%o5) Lisp array [38]

(%i6) /* Illustrating the compatibility between arrays */
      print(array_171[0],  array_171[1])$
      print(array_173[76], array_173[77])$
88.75054900000001  27.14727
88.75054900000001  27.14727

(%i8) /* Illustrating the compatibility between arrays */
      print(array_173[0],  array_173[1])$
      print(array_1143[0], array_1143[1])$
91.659554  27.76511
91.659554  27.76511

(%i10) /* Illustrating the compatibility between arrays */
       print(array_171[6],   array_171[7])$
       print(array_1143[36], array_1143[37])$
88.917877  27.321039
88.917877  27.321039

(%i12) Bhutan_polygon: make_polygon([171,173,1143])$
       draw2d(Bhutan_polygon)$
```

numbered_boundaries(nlist) **Function**

Example: Map of Europe labeling borders with their component number in boundaries_array.

```
(%i1) load(worldmap)$
      european_borders : region_boundaries(-31.81,74.92,49.84,32.06)$
      numbered_boundaries(european_borders)$
```

(**Author's note:** On my computer, this input results in the error message "`Error in APPLY [or a callee]: Lisp's arglist maximum surpassed`". I haven't had a chance to investigate this yet.)

`region_boundaries(x_1,y_1,x_2,y_2)` **Function**

Detects those polygonal segments of global variable boundaries_array that are fully contained in the rectangle with vertices (x_1, y_1) (upper left) and (x_2, y_2) (lower right).

Example: Returns segment numbers for plotting southern Italy. Compare with the example under fcn:region_boundaries_plus.

```
(%i1) load(worldmap)$
      region_boundaries(10.4,41.5,20.7,35.4);
(%o2) [1846, 1863, 1864, 1881, 1888, 1894]

(%i3) draw2d(geomap(%))$
```

`region_boundaries_plus(x_1,y_1,x_2,y_2)` **Function**

Detects those polygonal segments of global variable boundaries_array, at least one vertex of which is contained in the rectangle defined by vertices (x_1, y_1) (upper left) and (x_2, y_2) (lower right).

Example: Returns segment numbers for plotting southern Italy. Compare with the example under fcn:region_boundaries.

```
(%i1) load(worldmap)$
      region_boundaries_plus(10.4,41.5,20.7,35.4);
(%o2) [1060, 1062, 1076, 1835, 1839, 1844, 1846, 1858,
      1861, 1863, 1864, 1871, 1881, 1888, 1894, 1897]
(%i3) draw2d(geomap(%))$
```

Chapter 15

To do

- Continually audit and standardize grammar, punctuation, spelling, syntax, wording, typesetting, alphabetizing, spurious "OK's," wording, links, etc.
- Tighten up the LaTeX code.
- Make sure the global graphics options show as global and the others do not.
- Things to finish putting in:
  - Cross-references
    - "Allowable values"
    - "Applies to"
    - "See Also"
    - "Options"
  - Default values
  - Examples:
    - Get to the bottom of the examples that don't work
    - At least one example for each entry?
    - Graded examples?
    - Examples that connect the commands to typical tasks in typical math classes?
    - Graphics for the examples?
- A list of what wxMaxima tool to use for making different kinds of plots?
- Glossary?
- Index?

Chapter 16

Glossary (to appear?)

Chapter 17

Subject index (to appear?)

Chapter 18

Index of examples (to appear?)

Chapter 19

Quick reference guide (to appear?)

Appendix A

GNU public license (typesetting not completed)

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object

code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in

other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public,

the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

¡one line to give the program's name and a brief idea of what it does.¿ Copyright (C) ¡year¿ ¡name of author¿

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

¡signature of Ty Coon¿, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.