Graphics in the \texttt{wxMaxima} GUI
(Document version 0.78)

by David E. Brown, based heavily on \texttt{wxMaxima} help files

March 19, 2012

THIS DOCUMENT IS UNDER CONSTRUCTION
However, the content is complete enough to be useful.
Contents

1 Introduction
  1.1 About this document ............................................. 5
  1.2 About wxMaxima and graphics (under construction) ........ 5
  1.3 System requirements (under construction) .................... 6
  1.4 Plotting formats for wxMaxima ................................. 7
    1.4.1 gnuplot .................................................... 7
    1.4.2 gnuplot_pipes ............................................ 7
    1.4.3 mgnuplot .................................................. 8
    1.4.4 Xmaxima .................................................. 8
  1.5 Comparison of wxMaxima’s plotting routines (to appear) .... 8

2 The bode package .................................................. 9

3 The descriptive package ........................................... 11
  3.1 Functions for statistical graphs ............................... 11
  3.2 Graphics options for statistical graphics .................... 14

4 The draw package .................................................. 16
  4.1 Introduction to the draw package ............................... 16
  4.2 The draw and wxdraw commands and their variants ............ 17
  4.3 Scenes ......................................................... 18
  4.4 Graphics objects ............................................... 19
    4.4.1 Graphics objects used in both 2D and 3D plots ........... 19
    4.4.2 Graphic objects used in 2D plots only ................... 25
    4.4.3 Graphic objects used in 3D plots only ................... 29
  4.5 Options for draw and its variants ............................. 32
    4.5.1 Options used with both 2D and 3D graphics objects .... 32
    4.5.2 Options used with 2D graphics objects only .............. 40
    4.5.3 Options used with 3D graphics objects only .............. 42
    4.5.4 Options for coordinate axes ............................... 48
    4.5.5 The terminal and its options ............................. 55
    4.5.6 Options for the graphics window .......................... 58
    4.5.7 Handling multiple options ................................. 62
  4.6 Additional topics ............................................... 62
    4.6.1 Multiple plots ............................................. 62
    4.6.2 File handling: input, output, and exporting graphics made
          by draw ................................................... 64
5 The drawdf package ........................................... 66
5.1 The drawdf and wxdrawdf functions .................. 66
5.2 Graphics objects for drawdf and wxdrawdf ........... 68
5.3 Options for drawdf and wxdrawdf .................... 69

6 The dynamics package ...................................... 76
6.1 Functions for dynamics .................................. 76
6.2 Options for dynamics .................................... 80

7 The finance package ....................................... 83

8 The fractals package ...................................... 84
8.1 Graphics objects for fractals ......................... 84
8.2 Graphics options for fractal ........................... 86

9 The graphs package ....................................... 88
9.1 Introduction to graphs ................................ 88
9.2 Graphics objects for graphs ......................... 88
9.3 Functions for modifying graphs ..................... 94
9.4 File handling for graphs .............................. 96
9.5 Rendering graphs ....................................... 97
9.6 Graphics options for graphs ......................... 98

10 The implicit_plot package ................................ 102

11 The picture package ...................................... 103

12 The plot commands ...................................... 106
12.1 The plot and related functions ..................... 106
12.2 Options for plot ...................................... 110
12.3 Options for gnuplot .................................. 116
12.4 Functions for gnuplot_pipes ......................... 117

13 The plotdf package ...................................... 119
13.1 The plotdf command .................................. 119
13.2 Graphics objects for plotdf ......................... 120
13.3 Options for plotdf ................................... 121
13.4 plotdf’s plot window ................................ 123
13.5 The versus_t window ................................ 124

14 The worldmap package .................................. 127

15 To do ..................................................... 132

16 Glossary (to appear?) .................................. 133

17 Subject index (to appear?) ............................... 134

18 Index of examples (to appear?) ......................... 135

19 Quick reference guide (to appear?) .................... 136

A GNU public license (typesetting not completed) .... 137
Copyright © 2010 and 2011 David E. Brown. Until I have properly researched and resolved such issues as copyright and licensing, assume that you are only allowed to save or copy those parts of this document not written by David E. Brown—and this under the GPL (see the appendix)—unless it’s for your own personal academic use in courses offered at or through Brigham Young University–Idaho.¹

This document 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.

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 LaTeX document preparation (electronic typesetting) system, with liberal help from wxMaxima itself.

¹I 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

---

1Speaking 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. 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-preprocessed) 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.
1.4 Plotting formats for wxMaxima

Maxima currently uses gnuplot and Xmaxima for plotting. (Xmaxima supercedes the older Openmath program.) 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 inherently 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.

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

<table>
<thead>
<tr>
<th>To use these with wxMaxima:</th>
<th>You must also have these installed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bode</td>
<td>N/A</td>
</tr>
<tr>
<td>descriptive</td>
<td>N/A</td>
</tr>
<tr>
<td>draw and its variants</td>
<td>gnuplot 4.2 or later</td>
</tr>
<tr>
<td>drawdf</td>
<td>gnuplot 4.2 or later</td>
</tr>
<tr>
<td>dynamics</td>
<td>N/A</td>
</tr>
<tr>
<td>finance</td>
<td>N/A</td>
</tr>
<tr>
<td>fractal</td>
<td>N/A</td>
</tr>
<tr>
<td>graph</td>
<td>N/A, but can use graphviz programs</td>
</tr>
<tr>
<td>implicit_plot</td>
<td>N/A</td>
</tr>
<tr>
<td>picture</td>
<td>N/A</td>
</tr>
<tr>
<td>plot and its variants</td>
<td>gnuplot, gnuplot_pipes, mgnuplot, or Xmaxima</td>
</tr>
<tr>
<td>plotdf</td>
<td>Xmaxima</td>
</tr>
<tr>
<td>worldmap</td>
<td>N/A</td>
</tr>
</tbody>
</table>
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.
\[ H1(s) := \frac{100 \times (1 + s)}{((s + 10) \times (s + 100))} \]
\[ bode\_phase(H1(s), \{w, 1/1000, 1000\}) \]
\[ H2(s) := \frac{1}{(1 + s/\omega_0)} \]
\[ bode\_phase(H2(s), \{w, 1/1000, 1000\}), \omega_0 = 10 \]
\[ H3(s) := \frac{1}{(1 + s/\omega_0)^2} \]
\[ bode\_phase(H3(s), \{w, 1/1000, 1000\}), \omega_0 = 10 \]
\[ H4(s) := 1 + s/\omega_0 \]
\[ bode\_phase(H4(s), \{w, 1/1000, 1000\}), \omega_0 = 10 \]
\[ H5(s) := \frac{1}{s} \]
\[ bode\_phase(H5(s), \{w, 1/1000, 1000\}) \]
\[ H6(s) := \frac{1}{((s/\omega_0)^2 + 2 \times \zeta \times (s/\omega_0) + 1)} \]
\[ bode\_phase(H6(s), \{w, 1/1000, 1000\}), \omega_0 = 10, \zeta = 1/10 \]
\[ H7(s) := (s/\omega_0)^2 + 2 \times \zeta \times (s/\omega_0) + 1 \]
\[ bode\_phase(H7(s), \{w, 1/1000, 1000\}), \omega_0 = 10, \zeta = 1/10 \]

**Graphics option**

*bode_phase_unwrap*

*Default value:* false

*Allowable values:* false, true

*See also:* bode_phase

**Example:** The effect of the `bode_phase_unwrap` option.

\[ H8(s) := \frac{0.5}{(0.0001 \times s^3 + 0.002 \times s^2 + 0.01 \times s)} \]
\[ bode\_phase(H8(s), \{w, 1/1000, 1000\}) \]
\[ block([bode\_phase_unwrap: false],
    bode\_phase(H8(s), \{w, 1/1000, 1000\});
block([bode\_phase_unwrap: true],
    bode\_phase(H8(s), \{w, 1/1000, 1000\});
\]
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)**
- **histogram(list, options,...)**
- **histogram(one_column_matrix)**
- **histogram(one_column_matrix, options,...)**
- **histogram(one_row_matrix)**
- **histogram(one_row_matrix, options,...)**

  **Options:** nclasses and all options used by draw

  **See also:** continuous_freq, discrete_freq

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

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

  **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.

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

**Example:** Two-dimensional scatter plot.

```plaintext
(%i1) load(descriptive)$
    s2 : read_matrix(file_search("wind.data"))$
    scatterplot(submatrix(s2, 1,2,3),
```
### 3.1 Functions for statistical graphs

#### Example: Three dimensional scatter plot.

```lisp
(%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.

```lisp
(%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.)

```lisp
(%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.

```lisp
(%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.

```lisp
```
3.1 Functions for statistical graphs

(%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,
3.2 Graphics options for statistical graphics

<table>
<thead>
<tr>
<th>Graphics option</th>
<th>Default value</th>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>&quot;Windspeed in knots&quot;,</td>
<td>barsplot</td>
</tr>
<tr>
<td>xlabel</td>
<td>&quot;Stations&quot;,</td>
<td>barsplot</td>
</tr>
<tr>
<td>color</td>
<td>red,</td>
<td>barsplot</td>
</tr>
<tr>
<td>line_width</td>
<td>2)</td>
<td>barsplot</td>
</tr>
</tbody>
</table>

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

Default value: []

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

box_width

Default value: 3/4

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

groups_gap

Default value: 1

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

nclasses

Default value: 10

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
3.2 Graphics options for statistical graphics

Default value: orderlessp
Appplies 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 given Maxima session.

Note: As of 31 December 2010, wxMaxima support for Macintosh operating systems is experimental. In particular, gnuplot cannot 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 gnuplot to 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 gnuplot window 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.
4.2 The draw and wxdraw commands and their variants

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)))$

\[ \text{draw2d (option(s), graphic_object(s),...)} \] 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.

\[ \text{draw3d(option(s), graphic_object(s),...)} \] Function

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

---

1This 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, gr2d, wxdraw, wxdraw3d
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
4.4 Graphics objects

\texttt{gr3d} builds a complete description of a 3D scene, for use by the \texttt{draw} command. The arguments to \texttt{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: \texttt{cylindrical}, \texttt{elevation\_grid}, \texttt{explicit}, \texttt{geomap}, \texttt{implicit}, \texttt{label}, \texttt{parametric}, \texttt{parametric\_surface}, \texttt{points}, \texttt{quadrilateral}, \texttt{spherical}, \texttt{triangle}, \texttt{tube}, \texttt{vector}.

4.4 Graphics objects

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

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

\texttt{explicit(fcn, var, minval, maxval)} \hspace{1cm} Graphics object

Options: \texttt{adapt\_depth, color, fill\_color, filled\_func, key, line\_type, line\_width, nticks}

See also: \texttt{filled\_function}

\texttt{explicit(fcn, var1, minval1, maxval1, var2, minval2, maxval2)} \hspace{1cm} Graphics object

Options: \texttt{color, enhanced\_3d, key, line\_type, line\_width, xu\_grid, yv\_grid}

\texttt{explicit} draws functions that are defined explicitly.

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

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

\begin{verbatim}
(%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))$
\end{verbatim}

3D: \texttt{explicit(fcn, var1, minval1, maxval1, var2, minval2, maxval2)} plots function \texttt{fcn}, which must be given explicitly in terms of variables \texttt{var1} and \texttt{var2}. \texttt{explicit} plots \texttt{fcn} for all combinations of values of \texttt{var1} and \texttt{var2} ranging from \texttt{minval1} to \texttt{maxval1} and from \texttt{minval2} to \texttt{maxval2}, respectively. The effect is to plot \texttt{fcn} over a rectangle.

Example: A 3D explicit object.

\begin{verbatim}
(%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))$
\end{verbatim}

\texttt{implicit(exprssn, x,xmin,xmax, y,ymin,ymax)} \hspace{1cm} Graphics object
4.4 Graphics objects

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 exprssn.

(%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),
            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, \texttt{label([string, x,y,z])} writes the string at point \([x,y,z]\).

\begin{verbatim}
(%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]))$
\end{verbatim}

\begin{verbatim}
parametric(xfun, yfun, par,parmin,parmax) Graphics object
parametric(xfun, yfun, zfun, par,parmin,parmax) Graphics object
\end{verbatim}

\textbf{Options:} color, enhanced3d, key, line_type, line_width, nticks

Draws parameterized curves in 2D and 3D.

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

Example:

\begin{verbatim}
(%i1) load(draw)$
draw2d(explicit(exp(x), x,-1,3),
        color = red,
        parametric(2*cos(%theta), %theta^2, %theta,0,2*%pi))$
\end{verbatim}

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

Example:

\begin{verbatim}
(%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")$
\end{verbatim}

\textbf{points([[[x1,y1], [x2,y2],...]], Graphics object}
\textbf{points([[x1,x2,...], [y1,y2,...]], Graphics object}
\textbf{points([[y1,y2,...]], Graphics object}
\textbf{points([[x1,y1,z1], [x2,y2,z2],...]], Graphics object}
\textbf{points([[x1,x2,...], [y1,y2,...], [z1,z2,...]], Graphics object}
\textbf{points(matrix)} Graphics object
\textbf{points(1d_y_array)} Graphics object
\textbf{points(1d_x_array, 1d_y_array)} Graphics object
\textbf{points(1d_x_array, 1d_y_array, 1d_z_array)} Graphics object
4.4 Graphics objects

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

Options: color, key, line_type, line_width, pointJoined, 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,
   pointsJoined = 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(pointsJoined = 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 coordinates 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))$
4.4 Graphics objects

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))$
```

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]))$
```

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]))$
```

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

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

Example: 2D vectors with different options for their heads.
4.4.2 Graphic objects used in 2D plots only

bars([x1,h1,w1], [x2,h2,w2], ...)  

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)  

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

ellipse(xc, yc, a,b, polarang,Δpolarang) 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,
4.4 Graphics objects

```
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 xrange.

```
(%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], [5,3,3], [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 \texttt{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 \texttt{yrange} and \texttt{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]])$%

\textbf{image}(im, x0, y0, width, height) \quad \textbf{Graphics object}

See also: \texttt{colorbox}, \texttt{make_level_picture}, \texttt{make_RGB_picture}, \texttt{palette}, \texttt{read_xpm}

\texttt{image} renders images in 2D. \texttt{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 \texttt{levels} picture object, pixel values are interpreted according to graphics option \texttt{palette}. If im is a matrix of vectors of length three or an \texttt{rgb} picture object, they are interpreted as red, green and blue color components.

Example: Changing the colors of an \texttt{image} using option \texttt{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(
           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 [link](http://www.telefonica.net/web2/biomates/maxima/gpdraw/image) for more elaborate examples.  

\begin{verbatim}
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 coordinates, 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))$
\end{verbatim}

\begin{verbatim}
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]]))$
\end{verbatim}


\footnote{This link worked as recently as 2012-02-02.}
4.4 Graphics objects

**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 \(\theta\) 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.
4.4 Graphics objects

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

\textbf{Note:} In older versions of \textit{wxMaxima}, \textit{elevation\_grid} was called \textit{mesh}.

\textbf{Example:} Random elevations.

\begin{verbatim}
(%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))$
\end{verbatim}

\textbf{Example:} A simple \textit{mesh} plot.

\begin{verbatim}
(%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]]))$
\end{verbatim}

\textbf{Example:} Plotting a triangle in 3D.

\begin{verbatim}
(%i1) load(draw)$
  draw3d(line_width = 2,
         mesh([[1,0,0], [0,1,0]], [[0,0,1], [0,0,1]]))$
\end{verbatim}

\textbf{Example:} Two quadrilaterals.

\begin{verbatim}
(%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]]),
\end{verbatim}

\textbf{Note:} In older versions of \textit{wxMaxima}, \textit{mesh} was used as follows:

\begin{verbatim}
mesh(mat, x0,y0, width,height) Graphic object
\end{verbatim}

\textbf{Options:} \textit{color, enhanced3d, key, line\_type, line\_width, transform}

The rows that \textit{mesh} takes as arguments are lists of points in 3D, each of which is a list of the form \([x_i, y_i, z_i]\). 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, \textit{mesh} is a generalization of the \textit{elevation\_grid} object.\(^3\)

\textbf{Example:} Using \textit{mesh} in older versions of \textit{wxMaxima}.

\begin{verbatim}
(%i1) load(draw)$
  m : apply(‘matrix, makelist(makelist(random(10.0), k,1,30), i,1,20))$
  draw3d(color = blue,
         mesh(m, 0,0, 3,2),
         xlabel = "x",
         ylabel = "y",
         surface_hide = true)$
\end{verbatim}
4.4 Graphics objects

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

```lisp
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:

```lisp
(%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))$
```

```lisp
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 coordinates, 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 coordinates.

```lisp
(%i1) load(draw)$

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

```lisp
tube(xfun,yfun,zfun,rfun, p,pmin,pmax)

Graphics object

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

tube 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.

```lisp
(%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.

```maxima
(%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.

```maxima
(%i1) load(draw)$
        im: apply('matrix, makelist(makelist(random(200),i,1,30),i,1,30))$
```
4.5 Options for `draw` and its variants

```
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  
Applied 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_angle`s.

```
(%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  
Applied 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.
4.5 Options for \texttt{draw} and its variants

\begin{verbatim}
(%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]))$
\end{verbatim}

\textbf{head\_length} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value:} 2
\textbf{Applies to:} \texttt{vector} graphics objects
\textbf{See also:} \texttt{head\_both}, \texttt{head\_angle}, \texttt{head\_type}

\textit{head\_length} indicates, in the same units as on the \textit{x}-axis, the length of arrow heads.

\textbf{Example:} Various \textit{head\_lengths}.

\begin{verbatim}
(%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]))$
\end{verbatim}

\textbf{head\_type} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value:} \texttt{filled}
\textbf{Applies to:} \texttt{vector} graphics objects
\textbf{See also:} \texttt{head\_both}, \texttt{head\_angle}, \texttt{head\_length}

\textit{head\_type} is used to specify how arrow heads are plotted. Possible values are: \texttt{filled} (closed and filled arrow heads), \texttt{empty} (closed but not filled arrow heads), and \texttt{nofilled} (open arrow heads).

\textbf{Example:} Various \textit{head\_types}.

\begin{verbatim}
(%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]))$
\end{verbatim}

\textbf{label\_alignment} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value:} \texttt{center}
\textbf{Applies to:} \texttt{label} graphics objects
\textbf{See also:} \texttt{color}, \texttt{label\_orientation}
4.5 Options for \texttt{draw} and its variants

\texttt{label\_alignment} is used to specify where to write labels with respect to the given coordinates. Possible values are: \texttt{center}, \texttt{left}, and \texttt{right}.

\textbf{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))$
```

\texttt{label\_orientation} \hspace{1cm} Graphics option

\textbf{Default value:} horizontal
\textbf{Applies to:} label graphics objects
\textbf{See also:} color, \texttt{label\_alignment}

\texttt{label\_orientation} is used to specify orientation of labels. Possible values are: horizontal and vertical.

\textbf{Example:} Both label orientations. (In this example, a dummy point is needed, because \texttt{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))$
```

\texttt{line\_type} \hspace{1cm} Graphics option

\textbf{Default value:} solid
\textbf{Applies to:} In 2D: \texttt{ellipse}, \texttt{explicit}, \texttt{implicit}, \texttt{parametric}, \texttt{points}, \texttt{polar}, \texttt{polygon}, \texttt{rectangle}, \texttt{vector}; in 3D: \texttt{explicit}, \texttt{parametric}, \texttt{parametric\_surface}, \texttt{points}
\textbf{See also:} \texttt{line\_width}

\texttt{line\_type} indicates how lines are displayed. Its possible values are \texttt{solid} and \texttt{dots}.

\textbf{Example:} Showing both options for \texttt{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))$
```
4.5 Options for `draw` and its variants

**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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

\(^4\)Floating point numbers, actually, of course.
4.5 Options for \texttt{draw} and its variants

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

If an entry in the vector \texttt{palette} is negative, the corresponding negative color component will be used.

\textbf{Note:} \texttt{palette = gray} and \texttt{palette = color} are short cuts for \texttt{palette = [3,3,3]} and \texttt{palette = [7,5,15]}, respectively.

\textbf{Example:} Compare the results of plotting the same \texttt{image} graphics object using various vectors of length 3 as the values of \texttt{palette}.

\begin{verbatim}
(%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))$
\end{verbatim}

\textbf{Example:} \texttt{palette} and \texttt{enhanced3d} together.

\begin{verbatim}
(%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))$
\end{verbatim}

Second, \texttt{palette} can be a user-defined lookup table. In this case, the format for building a lookup table of length \(n\) is \texttt{palette=[color\_1, color\_2, ..., color\_n]}, where \texttt{color\_1} is a well formed color (see option \texttt{color}), such that \texttt{color\_1} is assigned to the lowest gray level and \texttt{color\_n} to the highest. The rest of colors are interpolated.

\textbf{Example:} Setting \texttt{palette} equal to a user-defined lookup table.

\begin{verbatim}
(%i1) load(draw)$
draw3d(palette = [red, blue, yellow],
    enhanced3d = x,
    explicit(x^2+y^2, x,-1,1, y,-1,1))$
\end{verbatim}

\textbf{points\_joined} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value:} false

\textbf{Applies to:} \texttt{points} graphics object

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

\textbf{Example:} Points joined by different types of lines.

\begin{verbatim}
(%i1) load(draw)$
draw2d(xrange = [0,10],
    points\_joined = true, 
    draw2d(xrange = [0,10],
    points\_joined = impulses, 
\end{verbatim}
4.5 Options for \texttt{draw} and its variants

\begin{verbatim}
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]])$
\end{verbatim}

\textbf{point\_size} \hspace{1cm} \text{Graphics option}

\textbf{Default value}: 1

\textbf{Applies to}: points graphics objects, in both 2D and 3D

\textbf{point\_size} sets the size for plotted points. It must be a non-negative number. \textbf{Note}: \textbf{point\_size} has no effect when graphics option \textbf{point\_type} is set to \texttt{dot}.

\textbf{Example}: 10 small points and 20 large points.

\begin{verbatim}(%i1) load(draw)$
   \texttt{draw2d}(xrange = \texttt{[0,10]},
yrange = \texttt{[0,10]},
point_size = 3,
point_type = \texttt{diamant},
points([[1,1], [5,1], [9,1]]),
point_type = \texttt{filled\_down\_triangle},
points([[1,2], [5,2], [9,2]]),
point_type = \texttt{asterisk},
points([[1,3], [5,3], [9,3]]))$
\end{verbatim}

\textbf{point\_type} \hspace{1cm} \text{Graphics option}

\textbf{Default value}: 1

\textbf{Applies to}: points graphics objects, in both 2D and 3D

\textbf{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.

\begin{itemize}
\item -1: none
\item 0: dot
\item 1: plus
\item 2: multiply
\item 3: asterisk
\item 4: square
\item 5: filled_square
\item 6: circle
\item 7: filled_circle
\item 8: up_triangle
\item 9: filled_up_triangle
\item 10: down_triangle
\item 11: filled_down_triangle
\item 12: diamant
\item 13: filled_diamant
\end{itemize}

\textbf{Example}: Several different points styles.

\begin{verbatim}(%i1) load(draw)$
   \texttt{draw2d}(xrange = \texttt{[0,10]},
yrange = \texttt{[0,10]},
point_size = 3,
point_type = \texttt{diamant},
points([[1,1], [5,1], [9,1]]),
point_type = \texttt{filled\_down\_triangle},
points([[1,2], [5,2], [9,2]]),
point_type = \texttt{asterisk},
points([[1,3], [5,3], [9,3]]),
\end{verbatim}
4.5 Options for \texttt{draw} and its variants

\begin{verbatim}
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]])$
\end{verbatim}

\textbf{transform} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value}: none

Effects a transformation of 2D or 3D space. If \texttt{transform} is \texttt{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 \texttt{transform}. In the case of a 2D scene, the list takes the form \([f_1(x,y), f_2(x,y), x,y]\). In the case of a 3D scene, the list is of the form \([f_1(x,y,z), f_2(x,y,z), f_3(x,y,z), x,y,z]\). The names of the variables defined in the lists may be different from those used in the definitions of the graphics objects.

\textbf{Example}: Rotation in 2D.

\begin{verbatim}
(%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]))$
\end{verbatim}

\textbf{Example}: Translation in 3D.

\begin{verbatim}
(%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))$
\end{verbatim}

\textbf{unit_vectors} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value}: false

This option affects only \texttt{vector} graphics objects.

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

\textbf{Example}: A vector and a \texttt{unit_vector}.

\textit{39}
4.5 Options for draw and its variants

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
4.5 Options for `draw` and its variants

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.

```latex
(%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`.

```latex
(%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 x 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 x 50 = 2500 points, but I don’t think so, based on experiments I’ve done.
4.5 Options for draw and its variants

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));

42
4.5 Options for `draw` and its variants

**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 \([\text{lowest}, s, \text{highest}]\), contour lines are plotted from \(\text{lowest}\) to \(\text{highest}\) in steps of \(s\).

- When option `contour_levels` is bound to a set of numbers of the form \(\{n_1, n_2, \ldots, n\}\), contour lines are plotted at levels \(n_1, n_2, \ldots, n\)

**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.
4.5 Options for draw and its variants

```
(%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 coordinates \([f(x,y,z), x, y, z]\), there are exactly two of them that accept models defined in terms of the \([x,y]\) coordinates, 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, \(\text{enhanced3d} = -a\) is a shortcut for \(\text{enhanced3d} = [-\text{foo},\text{foo}]\).

```
(\%i1) load(draw)$
  draw3d(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)$
  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))$
```
4.5 Options for `draw` and its variants

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

```plaintext
(%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.

```plaintext
(%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.

```plaintext
(%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.

```plaintext
(%i1) load(draw)$
draw3d(rot_vertical = 170,
       explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
```
4.5 Options for \texttt{draw} and its variants

\textbf{\texttt{surface\_hide}} \hfill \textbf{Global graphics option}

\textbf{Default value:} \texttt{false}

If \texttt{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.

\textbf{Example:} The effect of the \texttt{surface\_hide} option.

\begin{verbatim}
(%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)))$
\end{verbatim}

\textbf{\texttt{tube\_extremes}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{[open, open]}

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

\textbf{Example:} A tube with an \texttt{open} end and a \texttt{closed} end/

\begin{verbatim}
(%i1) load(draw)$
   draw3d(tube_extremes = [open, closed],
          tube(0,0,a,1, a,0,8))$
\end{verbatim}

\textbf{\texttt{view}} \hfill \textbf{Global graphics option}

\textbf{Default value:} \texttt{[60, 30]}

\textbf{Supercedes:} \texttt{rot\_horizontal} and \texttt{rot\_vertical}

\texttt{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 \texttt{x}-axis, in the range \([0, 180]\). The second one is the horizontal rotation around the \texttt{z}-axis, in the range \([0, 360]\).

\textbf{Example:} Setting the \texttt{view} angle.

\begin{verbatim}
(%i1) load(draw)$
   draw3d(view = [170, 360],
          explicit(sin(x^2+y^2), x,-2,2, y,-2,2))$
\end{verbatim}

\textbf{\texttt{x\_voxel}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{10}

\textbf{\texttt{y\_voxel}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{10}

\textbf{\texttt{z\_voxel}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{10}

The following applies to \texttt{y\_voxel} and \texttt{z\_voxel} just as it does to \texttt{x\_voxel}.

\texttt{x\_voxel} is the number of voxels in the \texttt{x}-direction to be used by the 3D \texttt{implicit} graphics object.

\textbf{\texttt{xu\_grid}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{30}

\textbf{\texttt{yv\_grid}} \hfill \textbf{Graphics option}

\textbf{Default value:} \texttt{30}

\textbf{Applies to:} \texttt{explicit} and \texttt{parametric\_surface} graphics objects in 3D
4.5 Options for `draw` and its variants

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.

```lisp
(%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.

```lisp
(%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.

```lisp
(%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.

```lisp
(%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.
4.5 Options for \texttt{draw} and its variants

\begin{verbatim}
(\%i1) load(draw)$
    draw2d(logx = true,
    explicit(log(x), x,0.01,5))$
\end{verbatim}

\textbf{proportional\_axes} \hfill \textit{Global graphics option}

\textbf{Default value: none}

When \texttt{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 coordinate axes agree, thus “squaring up” the plot.

\textbf{Notes:} (1) \texttt{proportional\_axes} is overridden by \texttt{xrange}, \texttt{yrange}, and \texttt{zrange}. (2) Option \texttt{proportional\_axes} requires \texttt{gnuplot} version 4.2.6 or later.

\textbf{Example:} Compare these plots to see the effect of \texttt{proportional\_axes}.

\begin{verbatim}
(\%i1) load(draw)$
(\%i2) draw2d(ellipse(0,0, 1,2, 0,360))$
(\%i3) draw2d(proportional\_axes = xy,
    ellipse(0,0, 1,2, 0,360))$
\end{verbatim}

\textbf{Example:} \texttt{proportional\_axes} in a multiplot.

\begin{verbatim}
(\%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)))$
\end{verbatim}

\textbf{xaxis} \hfill \textit{Global graphics option}

\textbf{Default value: false}

\textbf{yaxis} \hfill \textit{Global graphics option}

\textbf{zaxis} \hfill \textit{Global graphics option}

\textbf{See also:} \texttt{xaxis\_color}, \texttt{xaxis\_type}, \texttt{xaxis\_width}

The following applies to \texttt{yaxis} and \texttt{zaxis} just as it does to \texttt{xaxis}.

If \texttt{xaxis} is true, the x-axis is drawn.

\textbf{Example:} A plot with the x-axis included.

\begin{verbatim}
(\%i1) load(draw)$
    draw2d(xaxis = true,
    xaxis\_color = blue,
    explicit(x^3, x,-1,1))$
\end{verbatim}

\textbf{Example:} A plot with the y-axis included.

\begin{verbatim}
(\%i1) load(draw)$
    draw2d(yaxis = true,
    yaxis\_color = blue,
    explicit(x^3, x,-1,1))$
\end{verbatim}

49
Example: A plot with the $z$-axis included.

\begin{verbatim}
(%i1) load(draw)$
draw3d(zaxis = true,  
zaxis_type = solid,  
zaxis_color = blue.  
explicit(x^2+y^2, x,-1,1, y,-1,1))$
\end{verbatim}

xaxis_color \hspace{1cm} \text{Global graphics option}
\text{Default value: black}
yaxis_color \hspace{1cm} \text{Global graphics option}
\text{Default value: black}
zaxis_color \hspace{1cm} \text{Global graphics option}
\text{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.

\begin{verbatim}
(%i1) load(draw)$
draw2d(xaxis = true,  
xaxis_color = blue,  
explicit(x^3, x,-1,1))$
\end{verbatim}

Example: A plot with a blue $y$-axis.

\begin{verbatim}
(%i1) load(draw)$
draw2d(yaxis = true,  
yaxis_color = blue,  
explicit(x^3, x,-1,1))$
\end{verbatim}

Example: A plot with a blue $z$-axis.

\begin{verbatim}
(%i1) load(draw)$
draw3d(zaxis = true,  
zaxis_type = solid,  
zaxis_color = blue.  
explicit(x^2+y^2, x,-1,1, y,-1,1))$
\end{verbatim}

Example: A plot with a blue $x$-axis.

\begin{verbatim}
(%i1) load(draw)$
draw3d(zaxis = true,  
zaxis_type = solid,  
zaxis_color = blue.  
explicit(x^2+y^2, x,-1,1, y,-1,1))$
\end{verbatim}
4.5 Options for draw and its variants

(%i1) load(draw)$
draw2d(key = "x+1",
    explicit(x+1, x,1,2),
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))$

\textbf{xaxis_type} \hspace{1cm} \text{Global graphics option}
\textbf{Default value: dots}
\textbf{yaxis_type} \hspace{1cm} \text{Global graphics option}
\textbf{Default value: dots}
\textbf{zaxis_type} \hspace{1cm} \text{Global graphics option}
\textbf{Default value: dots}
\textbf{See also: xaxis, xaxis_color, xaxis_width}

The following applies to \textbf{yaxis_type} and \textbf{zaxis_type} just as it does to \textbf{xaxis_type}. \textbf{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)$

\textbf{xaxis_width} \hspace{1cm} \text{Global graphics option}
\textbf{Default value: 1}
4.5 Options for \texttt{draw} and its variants

\texttt{yaxis\_width}
\hspace{1cm} Global graphics option
\textbf{Default value:} 1

\texttt{zaxis\_width}
\hspace{1cm} Global graphics option
\textbf{Default value:} 1

\textbf{See also:} \texttt{xaxis}, \texttt{xaxis\_color}, \texttt{xaxis\_type}

The following applies to \texttt{yaxis\_width} and \texttt{zaxis\_width} just as it does to \texttt{xaxis\_width}.
\texttt{xaxis\_width} is the width of the \texttt{x}-axis. Its value must be a positive number.

\textbf{Example:} Changing the width of the \texttt{x}-axis.
\begin{verbatim}
(%i1) load(draw)$
draw2d(explicit(x^3, x,-1,1),
xaxis = true,
xaxis_width = 3)$
\end{verbatim}

\textbf{Example:} Changing the width of the \texttt{y}-axis.
\begin{verbatim}
(%i1) load(draw)$
draw2d(explicit(x^3, x,-1,1),
yaxis = true,
yaxis_width = 3)$
\end{verbatim}

\textbf{Example:} Changing the width of the \texttt{z}-axis.
\begin{verbatim}
(%i1) load(draw)$
draw3d(explicit(x^2+y^2, x,-1,1, y,-1,1),
zaxis = true,
zaxis_width = 3)$
\end{verbatim}

\texttt{xlabel}
\hspace{1cm} Global graphics option
\textbf{Default value:} ""

\texttt{ylabel}
\hspace{1cm} Global graphics option
\textbf{Default value:} ""

\texttt{zlabel}
\hspace{1cm} Global graphics option
\textbf{Default value:} ""

The following applies to \texttt{ylabel} and \texttt{zlabel} just as it does to \texttt{xlabel}.

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

\textbf{Example:} Labeling the \texttt{x}- and \texttt{y}-axes.
\begin{verbatim}
(%i1) load(draw)$
draw2d(xlabel = "Time",
ylabel = "Population",
explicit(exp(u), u,-2,2))$
\end{verbatim}

\texttt{xrange\_secondary}
\hspace{1cm} Global graphics option
\textbf{Default value:} auto

\texttt{yrange\_secondary}
\hspace{1cm} Global graphics option
\textbf{Default value:} auto

The following applies to \texttt{xrange\_secondary} just as it does to \texttt{yrange\_secondary}.

If \texttt{xaxis\_secondary} is \texttt{true} and \texttt{xrange\_secondary} is \texttt{auto}, the range for the second \texttt{x}-axis is computed automatically. If the user wants a specific interval for the second \texttt{x}-axis, it must be given as a \texttt{wxMaxima} list, as in \texttt{xrange\_secondary} = \([-2, 3]\).

\textbf{Example:} Setting the secondary \texttt{yrange}.
4.5 Options for \texttt{draw} and its variants

\begin{verbatim}
(%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))$
\end{verbatim}

\textbf{xtics} \hfill \textit{Global graphics option}

- \textbf{Default value: auto}

\textbf{ytics} \hfill \textit{Global graphics option}

- \textbf{Default value: auto}

\textbf{ztics} \hfill \textit{Global graphics option}

- \textbf{Default value: auto}

The following applies to \texttt{ytics} and \texttt{ztics} just as it does to \texttt{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 \texttt{xtics} is bound to symbol \texttt{auto}, tic marks are drawn automatically.
- When option \texttt{xtics} is bound to symbol \texttt{none}, tic marks are not drawn.
- When option \texttt{xtics} is bound to a positive number, this is the distance between two consecutive tic marks.
- When option \texttt{xtics} is bound to a list of length three of the form [\texttt{start}, \texttt{incr}, \texttt{end}], tic marks are plotted from \texttt{start} to \texttt{end} at intervals of length \texttt{incr}.
- When option \texttt{xtics} is bound to a set of numbers of the form \texttt{n\textsubscript{1}, n\textsubscript{2},...}, tic marks are plotted at values \texttt{n\textsubscript{1}, n\textsubscript{2},...}.
- When option \texttt{xtics} is bound to a set of pairs of the form \texttt{["label\textsubscript{1}, n\textsubscript{1}], ["label\textsubscript{2}, n\textsubscript{2}],...}, tic marks at locations \texttt{n\textsubscript{1}, n\textsubscript{2},...} are labeled with \texttt{"label\textsubscript{1}, "label\textsubscript{2},...} respectively.

\textbf{Example:} Disabling tic marks.

\begin{verbatim}
(%i1) load(draw)$
   draw2d(xtics = 'none,
         explicit(x^3, x,-1,1))$
\end{verbatim}

\textbf{Example:} Tic marks every 1/4 unit.

\begin{verbatim}
(%i1) load(draw)$
   draw2d(xtics = 1/4,
         explicit(x^3, x,-1,1))$
\end{verbatim}

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

\begin{verbatim}
(%i1) load(draw)$
   draw2d(xtics = [-3/4,1/8,3/4],
         explicit(x^3, x,-1,1))$
\end{verbatim}
4.5 Options for `draw` and its variants

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

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

**Example:** Labeled tic marks.

```lisp
(%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. If `xtics_secondary_axis` is false, tic marks are plotted on the border.
4.5 Options for **draw** and its variants

### 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 6cm min width and 5cm min height.

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

```lisp
(%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.

```lisp
(%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.
4.5 Options for draw and its variants

```lisp
(%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`.

**filebgcolor**

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.

```lisp
(%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.
4.5 Options for `draw` and its variants

```lisp
(%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\textsuperscript{TM} platforms.

2. Terminal `aquaterm` is used by Macintosh\textsuperscript{TM} computers.


4. Terminal `pdfcairo` requires gnuplot version 4.3 or newer.

5. Terminal `wxt` requires wxWidgets.\textsuperscript{7}

6. Terminal `x11` is used on Unix-like machines.

**Example:** Various terminal types.

```lisp
(%i11) 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,
       explicit(x^2, x,-1,1))$
```

\textsuperscript{6}As of 14 November 2011, gnuplot is in version 4.4.3.

\textsuperscript{7}wxMaxima comes with wxWidgets, other implementations of Maxima might not.
4.5 Options for \texttt{draw} and its variants

```maxima
cpic_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))$

\textbf{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.)

```maxima
(%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))$
```

\textbf{Example:} Terminals in multiple screens.

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

\textbf{Example:} An animated gif file.

```maxima
(%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 \hspace{1cm} \textbf{Global graphics option}

\textbf{Default value:} 1

Can be used as an argument of function \texttt{draw}.

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

\textbf{Example:} Two columns in the graphics window.

```maxima
(%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)$
```
4.5 Options for `draw` and its variants

**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.
    
    
    **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.

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

**Example:** Setting the font in the x11 terminal.

```maxima
(%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.

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

**Example:** A grid in a polar plot.

```maxima
(%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.
4.5 Options for draw and its variants

(%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-coordinate 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))$
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)),
       /* other options */$
4.6 Additional topics

\[
\text{gr2d(explicit(x^4, x,-1,1))}
\]

**Example:** Multiple graphics windows in a single gnuplot window.

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

**Example:** Simultaneous multiple gnuplot windows.

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

\text{multiplot_mode(term)}

**Function**

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

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

**Example:** Multiplot of some power functions.

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

delay

**Global graphics option**

\text{Default value: 5}

This option affects only \text{animated gif images}.

Can be used as an argument of function \text{draw}.

**See also:** \text{pic.height, pic.width, terminal}

Option \text{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.

\textbf{\footnotesize{\textsuperscript{9}}\text{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.}}
4.6 Additional topics

(%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, filebgcolor, 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 terminals 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

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

Default value: ""

After draw plots its scenes, you can save the plot co"ordinates 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 co"ordinates are saved. Its default value is the empty string, so by default, no co"ordinates 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)  
drawdf(dv/du, [u,v], options_and_objects)  
drawdf(dv/du, [u,umin,umax], [v,vmin,vmax], options_and_objects)  
drawdf([dx/dt,dy/dt], options_and_objects)  
drawdf([du/dt,dv/dt], [u,v], options_and_objects)  
drawdf([du/dt,dv/dt], [u,umin,umax], [v,vmin,vmax], options_and_objects)  

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. \( \frac{dy}{dx}, \frac{dx}{dt} \) and \( \frac{dy}{dt} \) are expressions that depend on \( x \) and \( y \). \( \frac{dv}{du}, \frac{du}{dt} \) and \( \frac{dv}{dt} \) 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 \( \frac{dv}{du} \) or \( \frac{du}{dt} \), 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 \texttt{draw2d} and \texttt{gr_2d} from the \texttt{draw} package are also supported by \texttt{drawdf}. Additional graphics objects and options are supported by \texttt{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.

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

**Example:** A basic direction field using a single ODE.

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

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

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

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

```lisp
(%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 \texttt{parametric} graphics object.

```lisp
(%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 \texttt{makelist} to plot several solution curves.

```lisp
(%i1) load(drawdf)$
  colors : [ 'red,'blue,'purple,'orange,'green]$
```
5.2 Graphics objects for drawdf and wxdrawdf

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

drawdf 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 coordinates [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,
examples and documentation for drawdf and wxdrawdf

Example: The same model, with two saddles plotted.

```plaintext
(%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

solns_at

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.

```plaintext
(%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

```plaintext
(%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

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 \texttt{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 \texttt{both} of option \texttt{direction}. If \texttt{direction} is \texttt{forward}, then the solution curve is plotted only for positive values of $t$. If \texttt{direction} is \texttt{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]))$
```

\textbf{duration} \hfill \textbf{Graphics option}

- **Default value:** 10
- **Applies to:** \texttt{soln_at, solns_at}
- **Allowable values:** Any floating-point number

\textit{duration} controls the lengths of solution curves by specifying the time duration of numerical integration. If \textit{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 \textit{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. \textit{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]))$
```
5.3 Options for drawdf and wxdrawdf

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

**See also:** color, soln_arrows  

**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 derivatives 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))$
```
5.3 Options for drawdf and wxdrawdf

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 4th-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
5.3 Options for drawdf and wxdrawdf

Default value: \([30, 22]\)

\(\text{field_grid} = [\text{cols,rows}]\) specifies the number of columns and rows in the grid.

Example: A coarser field grid than the default.

\[
\text{(%i1)} \quad \text{load(drawdf)}$
\]
\[
\text{drawdf}[x(1-x-y), y(3/4-y-x/2)],
\quad [x,0,1.1],
\quad [y,0,1],
\quad \text{field_grid} = [15,15],
\quad \text{soln_arrows} = \text{true},
\quad \text{solns_at}([1,0.8], [0.8,1])$
\]

\(\text{field_nsteps}\) \hspace{1cm} \text{Graphics option}

Default value: (undetermined, at present)

Allowable values: Any floating-point number

See also: \(\text{field_degree, field_duration}\)

When option \(\text{field_degree}\) 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 \(\text{field_nsteps}\). For each arrow, integration stops automatically if the arrow becomes long enough to potentially interfere with any neighboring arrow(s). If \(\text{field_nsteps}\) is negative or 0, no direction field is plotted.

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

\[
\text{(%i1)} \quad \text{load(drawdf)}$
\]
\[
\text{drawdf}[x(1-x-y), y(3/4-y-x/2)],
\quad [x,0,1.1],
\quad [y,0,1],
\quad \text{field_degree} = '\text{solns},
\quad \text{field_nsteps} = 1,
\quad \text{drawdf}[x(1-x-y), y(3/4-y-x/2)],
\quad [x,0,1.1],
\quad [y,0,1],
\quad \text{field_degree} = '\text{solns},
\quad \text{field_nsteps} = 5)$
\]

\(\text{field_tstep}\) \hspace{1cm} \text{Graphics option}

Default value: 0.1

Allowable values: Any floating-point number

See also: \(\text{field_degree}\)

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

\(\text{nsteps}\) \hspace{1cm} \text{Graphics option}

Default value: (undetermined, at present)

Allowable values: Any floating-point number

\(\text{drawdf}\) uses a numerical integrator to plot solution curves. \(\text{nsteps}\) 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 \(\text{nsteps}\) 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

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

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

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.
5.3 Options for drawdf and wxdrawdf

(%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 4th 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,...,n, and the vertical coördinates are the corresponding values y(n) of the sequence defined by the recurrence relation y(n+1) = F(y(n)), 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

\[
\begin{align*}
u(n+1) &= F(u(n), v(n)) \\
v(n+1) &= G(u(n), v(n)),
\end{align*}
\]

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.

\[
\text{ifs}([r1, \ldots, rm], [A_1, \ldots, A_m], [[x_1, y_1], \ldots, [x_m, y_m]], [x_0, y_0], \text{n, options})
\]

**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 \( 2 \times 2 \) 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 \( r_1, \ldots, r_m \). 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:

\[
\begin{align*}
\text{load(dynamics)}$
\text{a1: matrix([[0.85,0.04]], [-0.04,0.85]])}$
\text{a2: matrix([0.2,-0.26], [0.23,0.22])}$
\text{a3: matrix([-0.15,0.28], [0.26,0.24])}$
\text{a4: matrix([0,0], [0,0.16])}$
\text{p1: [0,1.6]}$
\text{p2: [0,1.6]}$
\text{p3: [0,0.44]}$
\text{p4: [0,0]}$
\text{w: [85,92,99,100]}$
\text{ifs(w, [a1,a2,a3,a4], [p1,p2,p3,p4], [5,0], 50000, [style,dots]);}
\end{align*}
\]

\[
\text{julia(x, y, options)}
\]

**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 \( \text{hue} + \text{huerange} \).

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.

```lisp
(%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`.

```lisp
(%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`.

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

Example: To enlarge the region around the lower bifurcation near \( x = -1.25 \) use:

```lisp
(%i1) load(dynamics)$
```

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

\[ \text{rk}(\text{ODE}, \text{var}, \text{initial}, \text{domain list}) \]

Function \[ \text{rk}([\text{ODE}_1, \ldots, \text{ODE}_m], [v_1, \ldots, v_m], [\text{init}_1, \ldots, \text{init}_m], \text{domain list}) \] Function

The command \( \text{rk}(\text{ODE}, \text{var}, \text{initial}, \text{domain list}) \) solves numerically a first-order ordinary differential. The command \( \text{rk}([\text{ODE}_1, \ldots, \text{ODE}_m], [v_1, \ldots, v_m], [\text{init}_1, \ldots, \text{init}_m], \text{domain list}) \) second form solves numerically a system of \( m \) such equations. Both commands use the 4th order Runge-Kutta method. \( \text{var} \) is the dependent variable. \( \text{ODE} \) 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 \text{domain list}, which must be a list of the form \( [\text{variable name}, \text{initial value}, \text{final value}, \text{step size}] \). For instance, the \text{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 \( v_1, v_2, \ldots, v_m \) in the syntax given above. The initial values for those variables will be \( \text{init}_1, \text{init}_2, \ldots, \text{init}_m \). There will still be just one independent variable defined by domain, as in the previous case. \( \text{ODE}_1, \text{ODE}_2, \ldots, \text{ODE}_m \) 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 \( \text{ODE}_1, \text{ODE}_2, \ldots, \text{ODE}_m \) 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 \text{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 \( \frac{dx}{dt} = t - x^2 \) 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 \text{results}.

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

Example: Numerical solution of the system

\[
\frac{dx}{dt} = 4 - x^2 - 4y^2 \quad \frac{dy}{dt} = y^2 - x^2 + 1,
\]

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 \text{sol}.

```lisp
(%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])$
```
6.2 Options for dynamics

staircase(F, y₀, m, options)

Function

Options: All those used by plot2d

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

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 \((y₀, y₀)\) 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 \((y₁, y₁)\) on the line, and the procedure is repeated \( m \) times until the point \((yₙ, yₙ)\) is reached.

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

(%i1) load(dynamics)$
(%i2) 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 coordinates, on the complex plane, of the point in the center of the region shown. The default value of 0 for both coordinates 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.
6.2 Options for dynamics

\[\text{[hue,n]}\]  
**dynamics option**

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

*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.

\[\text{[huerange,n]}\]  
**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.

\[\text{[levels,n]}\]  
**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.

\[\text{[pixels,n]}\]  
**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.

\[\text{[radius,r]}\]  
**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.

\[\text{[saturation,n]}\]  
**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.

\[\text{[size,side_length]}\]  
**dynamics option**

- **Default setting:** [size,400]  

\[\text{[size,width,height]}\]  
**dynamics option**

- **Default setting:** [size,400,400]  

81
6.2 Options for dynamics

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]

Default setting: [value,0.96]

Allowable values:

Applies to: julia, mandelbrot

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

```lisp
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.

```lisp
(%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)

fernfale 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. 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)

hilbertmap 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)

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)

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)

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])$
8.2 Graphics options for fractal

`sierpinskiale(n)`  
`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.  

**Example:** The triangular Sierpinski gasket.

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

`sierpinskimap(iterations)`  
`sierpinski_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.

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

`snowmap([cx_1, cx_2,...,cx_n], iterations)`  
`snowmap` plots the Koch snowflake map over the vertex set of an initial closed polygonal, in the complex plane. Because coordinate 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.

```lisp
(%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,1+%i,0],4)])$  
plot2d([discrete, snowmap([0,%i,1+%i,1,0],4)])$
```

`treefale(n)`  
`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.

**Example:** A tree fractal.

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

8.2 Graphics options for fractal

`julia_parameter`  
Default value: `%i`

---

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

```maxima
(%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.
9.2 Graphics objects for graphs

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

- **complement_graph(graph)**
  - Returns the complement of the graph `graph`.

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

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

- **copy_graph(graph)**
  - Returns a copy of the graph `graph`.

- **create_graph(v_list, e_list)**
  - 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.

```lisp
(%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.

```lisp
(%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.
9.2 Graphics objects for graphs

```
(%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. n must 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, g2)    Function
    Returns the direct product of graphs g1 and g2.

\(^2\)Author’s note to self: A 0-1 matrix? Weights?
```
Example:

```lisp
(%i1) load(graphs)$
   grid : graph_product(path_graph(3), path_graph(4))$
   draw_graph(grid)$
```

`graph_union(g1, g1)`
Returns the union (sum) of graphs $g1$ and $g2$.

`great_rhomboicosidodecahedron_graph()`
Returns the great rhomboicosidodecahedron graph.

`great_rhombicuboctahedron_graph()`
Returns the great rhombicuboctahedron graph.

`grid_graph(n, m)`
Returns the $n \times m$ grid. $m$ and $n$ must be positive integers.

`grotzch_graph()`
Returns the Grotzch graph.

`heawood_graph()`
Returns the Heawood graph.

`icosahedron_graph()`
Returns the icosahedron graph.

`icosidodecahedron_graph()`
Returns the icosidodecahedron graph.

`induced_subgraph(V, g)`
Returns the graph induced on the subset $V$ of vertices of the graph $g$.

Example:

```lisp
(%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)`
Returns the line graph of the graph $g$.

`make_graph(vrt, f)`
Returns the graph $f(vrt)$.
9.2 Graphics objects for graphs

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
9.2 Graphics objects for graphs

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()  
Returns the Tutte graph.  

underlying_graph(g)  
Returns the underlying graph of the directed graph g.  

wheel_graph(n)  
Returns the wheel graph on $n + 1$ vertices.

### 9.3 Functions for modifying graphs

**add_edge(e, gr)**  
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)**  
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)**  
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:
```
9.3 Functions for modifying graphs

add_vertices(v_list, gr) Function

Add 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:
0 : 3 2 1
1 : 0
2 : 0
3 : 0

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:
0 : 3 2 1
1 : 3
2 : 3
3 : 4 2 1 0
4 : 7 6 5 3
5 : 4
6 : 4
7 : 4

(%i4) contract_edge([3,4], g)$
   print_graph(g)$

Graph on 7 vertices with 6 edges.
Adjacencies:
0 : 3
1 : 3
2 : 3
3 : 5 6 7 2 1 0
4 : 7 6 5 3
5 : 4
6 : 4
7 : 4
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
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, f1)

Exports graphs in the list gr_list to the file f1 in the sparse6 format.

sparse6.import(f1)

Returns a list of graphs from the file f1 in the sparse6 format.

9.5 Rendering graphs

draw.graph(graph)

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,
9.6 Graphics options for graphs

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.

draw_graph_program
9.6 Graphics options for graphs

The color used for displaying edges.

**edge_coloring**

Default value:
The coloring of edges. The coloring must be specified in the format as returned by the function `edge_coloring`.

**edge_partition**

Default value:
A partition \([\{e_1,e_2,\ldots\},\ldots,\{e_k,\ldots,e_m\}]\) of edges of the graph. The edges of each list in the partition will be drawn using a different color.

**edge_type**

Default value:
Defines how edges are displayed. See the `line_type` option for the `draw` package.

**edge_width**

Default value:
The width of edges.

**file_name**

Default value:
The filename of the drawing if terminal is not screen.

**fixed_vertices**

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**

Default value: 15
The angle for the arrows displayed on arcs (in directed graphs).

**head_length**

Default value: 0.1
The length for the arrows displayed on arcs (in directed graphs).

**label_alignment**

Default value: center
Determines how to align the labels/ids of the vertices. Can be left, center or right.

**program**

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**

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**
- **Default value:**
  Display edges specified in the list `e_list` using a different color.

**show_edge_color**
- **Default value:**
  The color used for displaying edges in the `show_edges` list.

**show_edge_type**
- **Default value:**
  Defines how edges in `show_edges` are displayed. See the `line_type` option for the `draw` package.

**show_edge_width**
- **Default value:**
  The width of edges in `show_edges`.

**show_id**
- **Default value:** `false`
  If true then id’s of the vertices are displayed.

**show_label**
- **Default value:** `false`
  If true then labels of the vertices are displayed.

**show_vertex_type**
- **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**
- **Default value:**
  The size of vertices in `show_vertices`.

**show_vertex_color**
- **Default value:**
  The color used for displaying vertices in the `show_vertices` list.

**show_vertices**
- **Default value:** `[]`
  Display selected vertices in the using a different color.

**show_weight**
- **Default value:** `false`
  If true then weights of the edges are displayed.

**spring_embedding_depth**
- **Default value:** `50`
  The number of iterations in the spring embedding graph drawing algorithm.
9.6 Graphics options for graphs

terminal
   Default value:
   The terminal used for drawing (see the terminal option in the draw package).

vertex_color
   Default value:
   The color used for displaying vertices.

vertex_coloring
   Default value:
   Specifies coloring of the vertices. The coloring col must be specified in the format as returned by vertex_coloring.

vertex_partition
   Default value: []
   A partition \([v_1,v_2,\ldots],[v_k,\ldots,v_n]\) of the vertices of the graph. The vertices of each list in the partition will be drawn in a different color.

vertex_size
   Default value:
   The size of vertices.

vertex_type
   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)
   Converts a list \(v\text{\_list}\) of vertices to a list of edges of the cycle defined by \(v\text{\_list}\).

vertices_to_path(v_list)
   Converts a list \(v\text{\_list}\) of vertices to a list of edges of the path defined by \(v\text{\_list}\).

vertex_type
   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

```
%i1) load(implicit_plot)$
%i2) implicit_plot(x^2 = y^3 - 3*y + 1, [x, -4, 4], [y, -4, 4],
    [gnuplot_preamble, "set zeroaxis"])$
```

**Function**

```
implicit_plot(expr, x_range, y_range), options
implicit_plot([expr_1,..., expr_n], x_range, y_range, options)
```

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 \(\text{width}-1\) and \(\text{height}-1\), 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)
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.

\[
\begin{align*}
\text{(%i1) load(draw)}$
\text{make_level_picture(matrix([[3,2,5], [7,-9,3000]]));}
\text{(%o2) picture(level, 3, 2, \{Array: \{3 2 5 7 0 255\}\})}
\end{align*}
\]

Example: Level picture from numeric list.

\[^1\text{At least, in version 5.24.0 of wxMaxima, anyway.}\]
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 */
   (%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).

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
12.1 The plot and related functions

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-coordinates of the points to be plotted. There are two transformations defined by default: polar_to_xy and spherical_to_xyz.

polar_to_xy

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 coordinates), and transform them into x- and y-coordinates. Example: Use of the system function polar_to_xy to transform from cylindrical to rectangular coordinates. 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)

plot2d([plot_1, ..., plot_n], options)

plot2d([plot_1, ..., plot_n], x_range, options)

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 coordinates. 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 coordinates of the points to be plotted and the second list as the y coordinates. 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 coordinates 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 coordinates 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
12.1 The plot and related functions

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 co-ordinates 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)  \hspace{1cm} \text{Function}
plot3d([expr_1,...,expr_n], x_range,y_range, options)  \hspace{1cm} \text{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])$
```
12.1 The plot and related functions

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:

\[
(%i1) \text{plot3d}(\log (x^2y^2), [x, -2, 2], [y, -2, 2],
    \text{[grid, 29, 29],}
    \text{[palette, get_plot_option(palette,5)]})$
\]

Example: Two surfaces in the same plot, sharing the same domain. In gnuplot the two surfaces will use the same palette:

\[
(%i1) \text{plot3d}([2^(-x^2 + y^2), 4*\text{sin}(3*(x^2+y^2))/(x^2+y^2),}
    [x, -3, 3], [y, -2, 2])$
\]

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:

\[
(%i1) \text{plot3d}([[2^(-x^2 + y^2), [x,-2,2], [y,-2,2]],}
    4*\text{sin}(3*(x^2+y^2))/(x^2+y^2), [x, -3, 3], [y, -2, 2]],
    \text{[plot_format,xmaxima],}
    \text{[palette,[value,0.65,0.7,0.1,0.9]]})$
\]

Example: Plot of a Klein bottle, defined parametrically:

\[
(%i1) \text{expr_1: } 5 \cos(\text{x})*(\cos(\text{x}/2)*\cos(\text{y}) + \sin(\text{x}/2)*\sin(2*\text{y}) + 3.0) - 10.0$
\]
\[
\text{expr_2: } -5 \sin(\text{x})*((\cos(\text{x}/2)*\cos(\text{y}) + \sin(\text{x}/2)*\sin(2*\text{y}) + 3.0)$
\]
\[
\text{expr_3: } 5*(-\sin(\text{x}/2)*\cos(\text{y}) + \cos(\text{x}/2)*\sin(2*\text{y}))$
\]
\[
\text{plot3d}([\text{expr_1, expr_2, expr_3}], [\text{x, -}\pi, \pi], [
    \text{y, -}\pi, \pi], [
    \text{grid, 40, 40}])$
\]

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:

\[
(%i1) \text{M: matrix([1, 2, 3, 4], [1, 2, 3, 2], [1, 2, 3, 4],}
    [1, 2, 3, 3])$
\]
\[
f(\text{x, y}) := \text{float('M [round(\text{x}), round(\text{y})]])}$
\]
\[
\text{plot3d}(f(\text{x}, \text{y}), [\text{x, 1, 4}], [\text{y, 1, 4}], [
    \text{grid, 4, 4}])$
\]

apply: subscript must be an integer; found: round(\text{x})

spherical_to_xyz

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:

\[
(%i1) \text{plot3d}(5, [\theta, 0, \%pi], [\phi, 0, 2*\%pi],
    \text{[plot_format,xmaxima],}
    \text{[transform_xy, spherical_to_xyz],}
    \text{[palette,[value,0.65,0.7,0.1,0.9]]})$
\]
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 azimut. This option sets the value of azimut, 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
12.2 Options for plot

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 co"ordinates. 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
Options for `plot`

- **logy** [logy]
  - **Default value:** false
  - **Graphics option**
  - 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]
  - **Default value:** black
  - **Graphics option**
  - 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]
  - **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]]
  - **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)
  - **Graphics option**
  - 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]
  - **Default value:** In Windows™: gnuplot; in other platforms: gnuplot_pipes
  - **Graphics option**
  - **Allowable values:** gnuplot, Xmaxima, gnuplot, gnuplot_pipes
  - This option sets the format to be used for plotting.

- **plot_options**
  - **Default value:**
  - **System variable**
  - **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 \( \text{realpart}(\text{function}) \). 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 \( \log(\text{abs}(x)) \); 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 variable `plot_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.

```plaintext
(%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]],
        [transform_xy,color,grid,run Viewer,axes,plot_format,palette],
        [transform_xy,color,grid,run Viewer,axes,plot_format,palette]]
```

113
12.2 Options for `plot`

```
[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, ..., type_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*
12.2 Options for plot

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.

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.
12.3 Options for gnuplot

Option ylabel cannot be used with set_plot_option. See the example under xlabel.

\[ z \ [z, \text{min}, \text{max}] \] 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:

\[
(%i1) \text{plot3d}(\log(x^2*y^2), [x, -2, 2], [y, -2, 2], [z, -8, 4], \\
[\text{palette, false}], \\
[\text{color, magenta, blue}])$
\]

\[ zlabel \ [zlabel, \text{string}] \] 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.

\[ \text{gnuplot\_term} \] 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\textsuperscript{TM} page description language. If the option gnuplot_out_file is set to filename, gnuplot writes the Postscript\textsuperscript{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 size 1000,1000\"] 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.
12.4 Functions for gnuplot

**`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:

Depreciated. Superseded by `legend`, described in the preceding subsection.

**`gnuplot_curve_styles`**

**Graphics option**

Default value:

Depreciated. Superseded 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
12.4 Functions for gnuplot_pipes

Closes the pipe to gnuplot used by the gnuplot_pipes format.

\texttt{gnuplot\_restart()}

\textbf{Function}

Closes the pipe to gnuplot used by the gnuplot_pipes format and opens a new pipe.

\texttt{gnuplot\_replot(command\_string)}

\textbf{Function}

Allowable values: Any valid gnuplot command, including the empty string

Updates the gnuplot window. If \texttt{gnuplot\_replot} is called with a gnuplot command in a string \texttt{command\_string}, then \texttt{command\_string} is sent to gnuplot before replotting the window.

\texttt{gnuplot\_reset()}

\textbf{Function}

Resets the state of gnuplot used by the gnuplot_pipes format. To update the gnuplot window, call \texttt{gnuplot\_replot} after \texttt{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 4th order Runge-Kutta method.\(^1\)

13.1 The plotdf command

\[
\begin{align*}
\text{plotdf(dy/dx, options)} & \quad \text{Function} \\
\text{plotdf(dv/du, [u,v], options)} & \quad \text{Function} \\
\text{plotdf(dv/du, [u,v], [u,uMin, uMax], [v,vMin, vMax], options)} & \quad \text{Function}
\end{align*}
\]

\(^1\)Unfortunately, I have not been able to find or figure out how.
13.2 Graphics objects for \texttt{plotdf}

\texttt{plotdf([dx/dt, dy/dt], options)} \hspace{2cm} \textbf{Function}
\texttt{plotdf([du/dt, dv/dt], [u,v], options)} \hspace{2cm} \textbf{Function}
\texttt{plotdf([du/dt, dv/dt], [u,v], [u,uMin, uMax], [v,vMin, vMax], options)} \hspace{2cm} \textbf{Function}

\texttt{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 \texttt{parameters} option, or with a range of allowed values specified by the \texttt{sliders} option.

\textbf{Example:} A basic example using a single ODE: $y' - y = e^x$, which is rewritten as $y' = e^x + y$.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf(exp(-x)+y)$
\end{verbatim}

\textbf{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.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf([t,exp(-t)+y], [t,y])$
\end{verbatim}

\textbf{Example:} Renaming and assigning the ranges of the vertical and horizontal variables.

\begin{verbatim}
(%i1) load(plotdf)$
pplotdf([v,-k*z/m], [z,v],
   [z, -5,5],
   [v, -3,3],
   [parameters,"m=2,k=2"],
   [trajectory_at,3,0])$
\end{verbatim}

\section*{13.2 Graphics objects for \texttt{plotdf}}

\texttt{sliders} \hspace{2cm} \textbf{Graphics object}

\texttt{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 \texttt{name = min : max}

\textbf{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 $y = x'$.

\begin{verbatim}
(%i1) load(plotdf)$
pplotdf([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])$
\end{verbatim}

\textbf{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.

\begin{verbatim}
(%i1) load(plotdf)$
pplotdf([v,-k*z/m], [z,v],
   [parameters, "m=2,k=2"],
   [sliders, "m=1:5"],
   [trajectory_at, 6,0])$
\end{verbatim}
13.3 Options for \texttt{plotdf}

\texttt{trajectory\_at} \hspace{1cm} \textbf{Graphics object}

\texttt{trajectory\_at} defines the coördinates \texttt{xinitial} and \texttt{yinitial} for the starting point of an integral curve.

\textbf{Example:} A direction field with a trajectory. An asymptote to the trajectory is also plotted.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf(x-y^2, [y,-5,5], [x,-4,16],
[xfun, "sqrt(x);-sqrt(x)"],
[trajectory_at, -1,3])$
\end{verbatim}

\texttt{xfun} \hspace{1cm} \textbf{Graphics object}

\texttt{xfun} defines a string with semicolon-separated sequence of functions of \texttt{x} to be displayed, on top of the direction field. (Those functions will be parsed by \texttt{Tcl} and not by \texttt{Maxima}.)

\textbf{Example:} \texttt{xfun} used to add an asymptote of a trajectory to a direction field.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf(x-y^2, [xfun, "sqrt(x)"],
[trajectory_at, -1,3],
[y,-5,5], [x,-4,16])$
\end{verbatim}

\textbf{Example:} The same example, modified to show the syntax for plotting more than one additional function.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf(x-y^2, [xfun, "sqrt(x);-sqrt(x)"],
[trajectory_at, -1,3],
[y,-5,5], [x,-4,16])$
\end{verbatim}

13.3 \textbf{Options for plotdf}

Options for \texttt{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 \texttt{trajectory\_at}. The direction of the integration can be controlled with the \texttt{direction} option, which can have values of \texttt{forward}, \texttt{backward} or \texttt{both}. The number of integration steps is given by \texttt{nsteps} and the time interval between them is set using \texttt{tstep} option.

\texttt{direction} \hspace{1cm} \textbf{Graphics option}

\textbf{Default value:} both

\textbf{Allowable values:} backward, both, forward

\texttt{direction} defines the direction of the independent variable that will be followed to compute an integral curve. Option value \texttt{forward} makes the independent variable increase \texttt{nsteps} times, with increments \texttt{tstep}; \texttt{backward} makes the independent variable decrease; or \texttt{both}, which leads to an integral curve that extends \texttt{nsteps} forward, and \texttt{nsteps} backward.

\textbf{Example:} Different directions. Compare the effect of the default value \texttt{both} of \texttt{direction} with that of the value \texttt{forward}.

\begin{verbatim}
(%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],
[trajectory_at, 2,2])$
\end{verbatim}
13.3 Options for \texttt{plotdf}

\begin{verbatim}
[direction, forward],
[parameters, "k=-1,m=1.0,c=0,b=1"],
[trajectory_at, 2,2])
\end{verbatim}

\textbf{Author's note}: There seems to be a bug in the code for this option. Giving it the value \texttt{backward} or the value \texttt{both} seems to have the same effect as giving it the value \texttt{forward}. However, changing the \texttt{direction} in the configuration menu of the plot window works just fine.

\texttt{nsteps}

\textbf{Default value}: 100

\textbf{Allowable values}: Any floating-point number

\texttt{nstep} defines the number of steps of length \texttt{tstep} that will be used for the independent variable, to compute an integral curve. If \texttt{nsteps} is 0 or negative, no integral curve is plotted.

\textbf{Example}: Differing values of \texttt{nstep}.

\begin{verbatim}
(%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])$
\end{verbatim}

\texttt{parameters([`name1=value1, name2=value2,...,nameN=valueN'])}

\textbf{Graphics option}

\textbf{Default value}: (empty)

Use this option to set numeric values of parameters used in the definition of differential equations for use with \texttt{plotdf}. The names and values of the parameters must be given in a string with a comma-separated sequence of pairs \texttt{name = value}. This option is particularly useful when combined with the \texttt{sliders} graphics object.

\textbf{Example}: A simple harmonic oscillator.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf([v,-k*z/m], [z,v],
[parameters,`m=2,k=2'],
[trajectory_at,6,0])$
\end{verbatim}

\texttt{tinitial}

\textbf{Default value}: 0

\textbf{Allowable values}: Any floating-point number

\texttt{tinitial} defines the initial value of variable \texttt{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 \texttt{t}.

\textbf{Example}: Different values of \texttt{tinitial}.

\begin{verbatim}
(%i1) load(plotdf)$
plotdf([y,-(k*x + c*y + b*x^3)/m],
[tinitial, 0],
[nsteps, 60],
\end{verbatim}
Author’s note: There seems to be a bug in the code for this option. Changing the value of \texttt{tinitial} seems to have no effect.

\textbf{tstep}  \hspace{1cm} \textbf{Graphics option}

\textbf{Default value}: 0.1

\textbf{Allowable values}: Any floating-point number

\texttt{tstep} defines the length of the increments of the independent variable \( t \), used to compute an integral curve. If only one expression \( \frac{dy}{dx} \) is given to \texttt{plotdf}, the variable \( x \) will be directly proportional to \( t \). If \texttt{tstep} is 0 or negative, no integral curve is plotted.

13.4 \texttt{plotdf}’s plot window

When \texttt{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, \texttt{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 \texttt{sliders}, \texttt{plotdf} provides a button bar in the plot window.

In order from left to right, the buttons are \texttt{close}, \texttt{configure}, \texttt{replot}, \texttt{save}, \texttt{zoom in}, \texttt{zoom out}, and \texttt{versus \_ t}. \texttt{close}, \texttt{replot}, \texttt{save}, \texttt{zoom in}, and \texttt{zoom out} behave exactly as one might expect, except that \texttt{replot} replots only the most recently plotted solution curve. \texttt{save} saves the plot in the encapsulated \texttt{Postscript} \textsuperscript{TM} format. Clicking \texttt{versus \_ t} opens the \texttt{versus \_ t} window. (See \texttt{versus \_ t} for details.)

Clicking \texttt{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 \( \frac{dy}{dx} \) or \( \frac{dy}{dt}, \frac{dx}{dt} \), as desired. If you change any settings, you’ll need to click \texttt{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 \( \frac{dy}{dx} \) or \( \frac{dy}{dt} \) and \( \frac{dx}{dt} \), according to which of these you have selected. This gives you a place to edit your ODE or your system of ODE’s.

- \texttt{Trajectory at}: To add a trajectory to the plot, enter coordinates for the desired point and press the \texttt{Enter} key.

- \texttt{fieldlines}: Sets the color of the trajectories. If empty, no trajectories are plotted.

- \texttt{vectors}: Sets the color of the vectors that make up the plot of the direction field. If empty, the direction field is not plotted.

- \texttt{curves}: Sets the color of the orthogonal trajectories of the direction field. If empty, no orthogonal trajectories are plotted.

- \texttt{direction}: Sets the direction in which trajectories are plotted. See \texttt{direction} for details.
13.5 The \texttt{versus\_t} window

- \texttt{nsteps}: Sets the number of time steps used by the numerical integrator in plotting trajectories. See \texttt{nsteps} for details.

- \texttt{versus\_t}: Sets the \texttt{versus\_t} window state. “1” is for “display the \texttt{versus\_t} window” and “0” is for “do not display the \texttt{versus\_t} window. See \texttt{versus\_t} for details.

- \texttt{tinitial}: Sets the initial time for use by the numerical integrator. See \texttt{tinitial} for details.

**Author’s note:** This option does not seem to work on my system; perhaps there’s a bug in the code.

- \texttt{ycenter}: Sets the center of the vertical range of the plot. The range actually used is \([\texttt{ycenter} - \texttt{yradius}, \texttt{ycenter} + \texttt{yradius}]\). If you set the vertical range in the \texttt{wxMaxima} session window by using \([\texttt{y}, \texttt{y\_min}, \texttt{y\_max}]\) as an option in the \texttt{plotdf} command, then \texttt{ycenter} is \((\texttt{y\_min} + \texttt{y\_max})/2\).

- \texttt{xcenter}: Sets the center of the horizontal radius of the plot. The range actually used is \([\texttt{xcenter} - \texttt{xradius}, \texttt{xcenter} + \texttt{xradius}]\). If you set the horizontal range in the \texttt{wxMaxima} session window by using \([\texttt{x}, \texttt{x\_min}, \texttt{x\_max}]\) as an option in the \texttt{plotdf} command, then \texttt{xcenter} is \((\texttt{x\_min} + \texttt{x\_max})/2\).

- \texttt{yradius}: Sets the radius of the vertical range of the plot. The range actually used is \([\texttt{ycenter} - \texttt{yradius}, \texttt{ycenter} + \texttt{yradius}]\). If you set the vertical range in the \texttt{wxMaxima} session window by using \([\texttt{y}, \texttt{y\_min}, \texttt{y\_max}]\) as an option in the \texttt{plotdf} command, then \texttt{yradius} is \((\texttt{y\_min} - \texttt{y\_max})/2\).

- \texttt{xradius}: Sets the radius of the horizontal range of the plot. The range actually used is \([\texttt{xcenter} - \texttt{xradius}, \texttt{xcenter} + \texttt{xradius}]\). If you set the horizontal range in the \texttt{wxMaxima} session window by using \([\texttt{x}, \texttt{x\_min}, \texttt{x\_max}]\) as an option in the \texttt{plotdf} command, then \texttt{xradius} is \((\texttt{x\_min} - \texttt{x\_max})/2\).

- \texttt{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.\(^2\)

- \texttt{xfun}: Includes functions in the plot, without requiring them to be trajectories or orthogonal trajectories of the direction field. See \texttt{xfun} for details.

- \texttt{parameters}: Sets the values of any optional parameters that may have been included in the \texttt{dy/dx}, \texttt{dy/dt}, or \texttt{dx/dt} fields. See \texttt{parameters} for details.

**Note:** Bear in mind that if you change any of the settings in the \texttt{Plot Setup} dialog, you need to click \texttt{replot} to cause the new settings to take effect.

### 13.5 The \texttt{versus\_t} window

**\texttt{versus\_t}**

\textbf{Graphics option}

**Default value:** 0

**Allowable values:** Any floating-point number

\texttt{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 \texttt{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 \texttt{close}, \texttt{configure}, \texttt{replot}, \texttt{save}, \texttt{zoom in}, \texttt{zoom out}, and \texttt{help}. \texttt{close}, \texttt{replot}, \texttt{save}, \texttt{zoom in}, \texttt{zoom out}, and \texttt{help} behave exactly as one might expect.

\(^2\)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 \([y_{center} - y_{radius}, y_{center} + y_{radius}]\).

- **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 \([x_{center} - x_{radius}, x_{center} + x_{radius}]\).

- **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 \([y_{center} - y_{radius}, y_{center} + y_{radius}]\).

- **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 \([x_{center} - x_{radius}, x_{center} + x_{radius}]\).

\(^3\)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],
      [versus_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 coordinates of points that define boundaries. Each component of **boundaries_array** is an array of floating-point numbers representing the coordinates 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 coordinates 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,
/* Vertices of boundary #1: {(4,3), (5,4), (6,4), (5,1)} */
(bnd1: make_array(flonum,8),
/* 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,
```

/* Vertices of boundary #3: {(1,1), (4,3)} */

(bnd3: make_array(flonum,4),

/* Vertices of boundary #4: {(4,3), (5,1)} */

(bnd4: make_array(flonum,4),

/* Pack all together in boundaries_array */

(boundaries_array: make_array(any,5),
  boundaries_array[0] : bnd0, boundaries_array[1] : bnd1,

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.

Example: Drawing Africa and selected countries, by name.

(%) 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.)

(%) 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.

1See 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](http://www.telefonica.net/web2/biomates/maxima/gpdraw/geomap) for more elaborate examples.

```plaintext
numbered_boundaries(nlist) Function
```

Draws a list of polygonal segments (boundaries), labeled by its numbers (boundaries_array coordinates). This is of great help when building new geographical entities.

```plaintext
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, %)$
```

```plaintext
make_poly_continent(continent_name) Function
make_poly_country(country_name) Function
```

**Example:** India.

---

2 This URL worked as recently as 2012-02-02.
Function

make_polygon(nlist)

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)
**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

Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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
totally 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.