Maxima by Example:
Ch. 2, Plots, Files, Read, Write, and Fit *

Edwin L. Woollett

June 14, 2013

Contents

*This version uses **Maxima 5.26.0**. Check **http://www.csulb.edu/~woollett/** for the latest version of these notes. Send comments and suggestions to **woollett@charter.net**

```
COPYING AND DISTRIBUTION POLICY
This document is part of a series of notes titled
"Maxima by Example" and is made available
via the author's webpage http://www.csulb.edu/~woollett/
to aid new users of the Maxima computer algebra system.

NON-PROFIT PRINTING AND DISTRIBUTION IS PERMITTED.

You may make copies of this document and distribute them
to others as long as you charge no more than the costs of printing.

These notes (with some modifications) will be published in book form
eventually via Lulu.com in an arrangement which will continue
to allow unlimited free download of the pdf files as well as the option
of ordering a low cost paperbound version of these notes.
```

Feedback from readers is the best way for this series of notes to become more helpful to new users of Maxima. *All* comments and suggestions for improvements will be appreciated and carefully considered.

```
LOADING FILES
The defaults allow you to use the brief version load(fft) to load in the
Maxima file fft.lisp.
To load in your own file, such as qxxx.mac
using the brief version load(qxxx), you either need to place
qxxx.mac in one of the folders Maxima searches by default, or
else put a line like:

file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$

in your personal startup file maxima-init.mac (see Ch. 1, Introduction to
  Maxima for more information about this).

Otherwise you need to provide a complete path in double quotes,
as in load("c:/work2/qxxx.mac"),

We always use the brief load version in our examples, which are generated
using the XMaxima graphics interface on a Windows XP computer, and copied
into a fancy verbatim environment in a latex file which uses the fancyvrb
and color packages.
```

```
Maxima.sourceforge.net. Maxima, a Computer Algebra System. Version 5.26.0
 (2012). http://maxima.sourceforge.net/
```

2.1 Introduction to plot2d

You should be able to use any of our examples with either **wxMaxima** or **Xmaxima**. If you substitute the word **wxplot2d** for the word **plot2d**, you should get the same plot (using **wxMaxima**), but the plot will be drawn "in-line" in your notebook rather than in a separate gnuplot window, and the vertical axis labeling will be rotated by 90 degrees as compared to the gnuplot window graphic produced by **plot2d**.

To save a plot as an image file, using **wxMaxima**, right click on the inline plot, choose a name and a destination folder, and click ok.

To save a plot drawn in a separate **gnuplot** window, right click the small icon in the upper left hand corner of the plot window, choose Options, Copy to Clipboard, and then open any utility which can open a picture file and select Edit, Paste, and then File, Save As. A standard utility which comes with Windows XP is the accessory Paint, which will work fine in this role to save the clipboard image file. The freely available Infanview is a combination picture viewer and editor which can also be used for this purpose. Saving the image via the gnuplot window route results in a larger image.

2.1.1 First Steps with plot2d

The syntax of **plot2d** is

```
plot2d( object-list, draw-parameter-list, other-option-lists).
```

The required object list (the first item) may be simply one object (not a list). The object types may be expressions (or functions), all depending on the same draw parameter, discrete data objects, and parametric objects. If at least one of the plot objects involves a draw parameter, say **p**, then a draw parameter range list of the form **[p, pmin, pmax]** should follow the object list.

We start with the simplest version which only controls how much of the expression to plot, and does not try to control the canvas width or height.

```
(%i1) plot2d ( sin(u), [’u, 0, %pi/2] )$
```

which produces (on the author's Windows XP system) approximately:

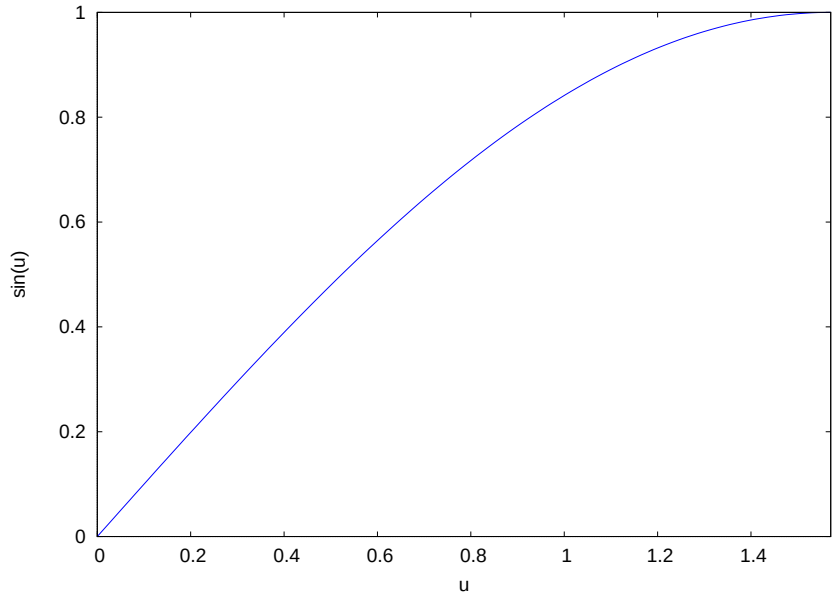

Figure 1: plot2d (sin(u), [’u, 0, %pi/2])

Since the plot for the source tex file for this pdf was produced using

```
plot2d ( sin(u), ['u,0,%pi/2], [psfile,"ch2p01.eps"] )$
```

there is one obvious difference compared to what you see on your console: the vertical axis label **sin(u)** is rotated by 90 degrees, and hence appears as does the inline **wxmaxima** plot using **wxplot2d**.

We see that **plot2d** has made the canvas width only as wide as the drawing width, and has made the canvas height only as high as the drawing height. Now let's add a horizontal range (canvas width) control list in the form **['x,-0.2,1.8]**. Notice the special role the symbol **x** plays here in **plot2d**. **u** is a plot parameter, and **x** is a horizontal range control parameter.

```
(%i2) plot2d ( sin(u),['u,0,%pi/2],['x, -0.2, 1.8] )$
```

which produces approximately:

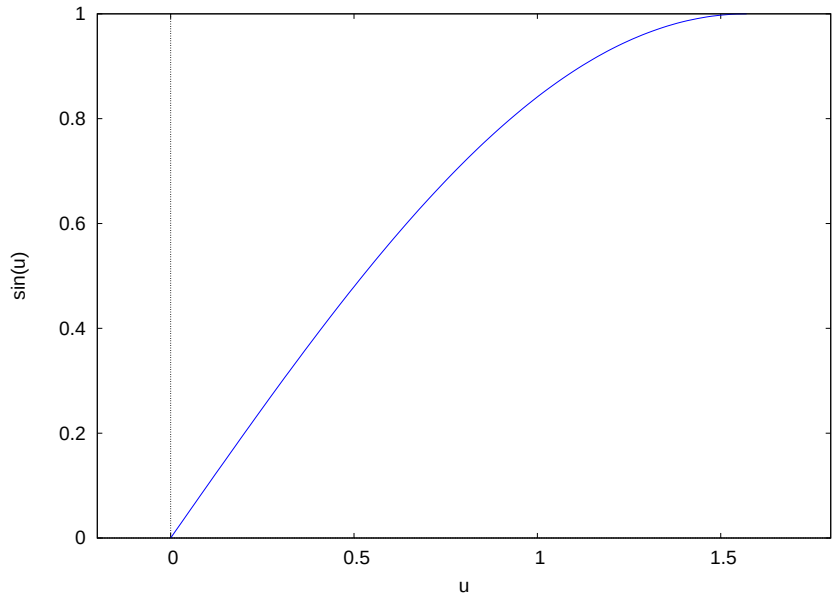

Figure 2: plot2d (sin(u), ['u, 0, %pi/2], ['x,-0.2,1.8])

We see that we now have separate draw width and canvas width controls included. If we try to put the canvas width control list before the draw width control list, we get an error message:

```
(%i3) plot2d(sin(u),['x,-0.2,1.8], ['u,0,%pi/2] )$
set_plot_option: unknown plot option: u
 -- an error. To debug this try: debugmode(true);
```

However, if the expression variable happens to be **x**, the following command includes both draw width and canvas width using separate **x** symbol control lists, and results in the correct plot:

```
(%i4) plot2d ( sin(x), ['x,0,%pi/2], ['x,-0.2,1.8] )$
```

in which the first (required) **x** drawing parameter list determines the drawing range, and the second (optional) **x** control list determines the canvas width.

Despite the special role the symbol **y** also plays in **plot2d**, the following command produces the same plot as above.

```
(%i5) plot2d ( sin(y), ['y,0,%pi/2], ['x,-0.2,1.8] )$
```

The **optional** vertical canvas height control list uses the special symbol **y**, as shown in

```
(%i6) plot2d ( sin(u), ['u,0,%pi/2], ['x,-0.2,1.8], ['y,-0.2, 1.2] )$
```

which produces

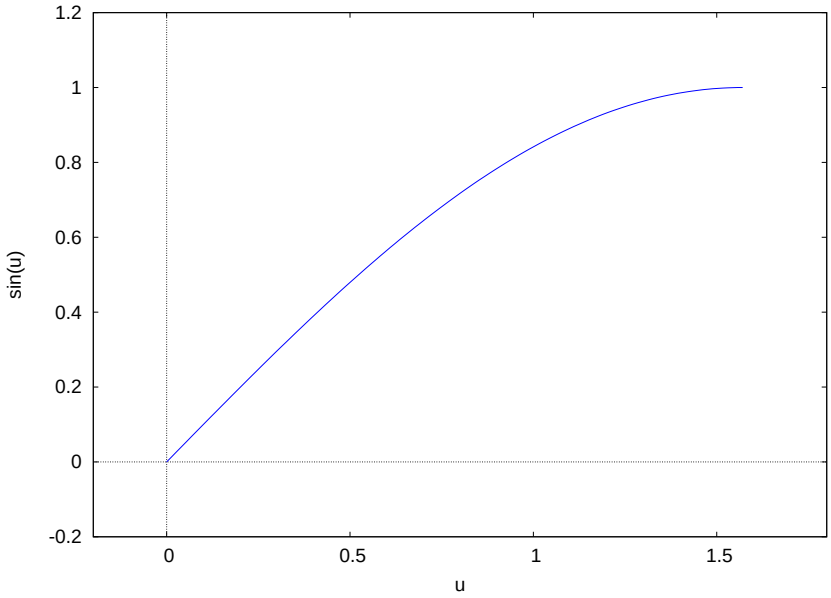

Figure 3: plot2d (sin(u), ['u,0,%pi/2], ['x,-0.2,1.8], ['y,-0.2, 1.2])

and the following alternatives produce exactly the same plot.

```
(%i7) plot2d ( sin(u), ['u,0,%pi/2], ['y,-0.2, 1.2], ['x,-0.2,1.8] )$
(%i8) plot2d ( sin(x), ['x,0,%pi/2], ['x,-0.2,1.8], ['y,-0.2, 1.2] )$
(%i9) plot2d ( sin(y), ['y,0,%pi/2], ['x,-0.2,1.8], ['y,-0.2, 1.2] )$
```

2.1.2 Parametric Plots

For orientation, we will draw a sine curve using the parametric plot object syntax and using a parametric parameter **t**.

```
(%i1) plot2d ( [parametric, t, sin(t), ['t, 0, %pi] ] )$
```

which produces

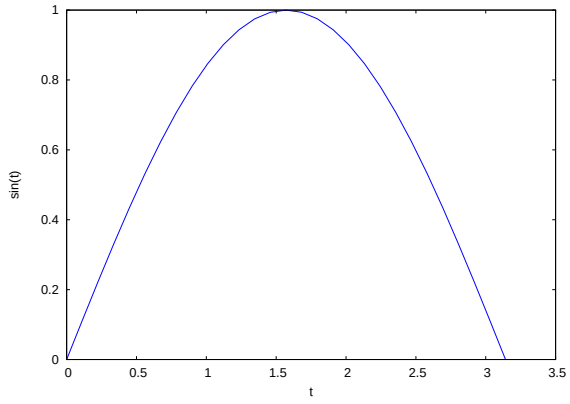

Figure 4: plot2d ([parametric, t, sin(t), ['t, 0, %pi]])

As the **plot2d** section of the manual asserts, the general syntax for a **plot2d** parametric plot is

```
plot2d (... [parametric,x_expr,y_expr,t_range],...)
```

in which **t_range** has the form of a list: **['t,tmin,tmax]** if the two expressions are functions of **t**, say. There is no restriction on the name used for the parametric parameter.

We see that

```
plot2d ( [ parametric,  fx(t),  fy(t), [ 't, tmin, tmax ] ] )$
```

plots pairs of points **(fx (ta), fy(ta))** for **ta** in the interval **[tmin, tmax]**. We have used **no** canvas width control list **['x, xmin, xmax]** in this minimal version.

2.1.3 Can We Draw A Circle?

This is a messy subject. We will only consider the separate gnuplot window mode (not the embedded plot mode) and assume a maximized gnuplot window (as large as the monitor allows).

We use a parametric plot to create a "circle", letting **fx(t) = cos(t)** and **fy(t) = sin(t)**, and again adding no canvas width or height control lists.

```
(%i2) plot2d ([parametric, cos(t), sin(t), ['t,-%pi,%pi]])$
```

If this plot is viewed in a maximized gnuplot window, the height to width ratio is about 0.6 on the author's equipment. The corresponding eps file for the figure included here has a height to width ratio of about 0.7 when viewed with GSView, and about the same ratio in this pdf file:

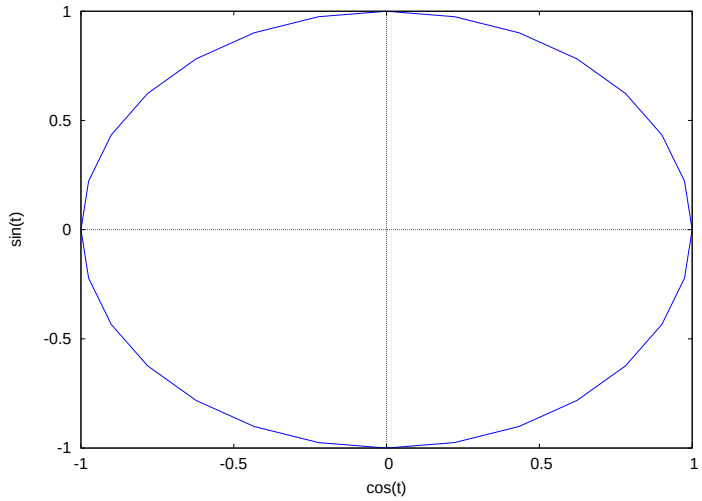

Figure 5: plot2d ([parametric, cos(t), sin(t), ['t, -%pi, %pi]])

There are two approaches to better "roundness". The first approach is to use the **plot2d** option **[gnuplot_preamble,"set size ratio 1;"]**, as in

```
(%i3) plot2d ([parametric, cos(t), sin(t), ['t,-%pi,%pi]],
          [gnuplot_preamble,"set size ratio 1;"])$
```

With this command, the author gets a height to width ratio of about 0.9 using the fullscreen gnuplot window choice. The eps file save of this figure, produced with the code

```
plot2d ([parametric, cos(t), sin(t), ['t,-%pi,%pi]],
           [gnuplot_preamble,"set size ratio 1;"],
           [psfile,"ch2p06.eps"] )$
```

had a height to width ratio close to 1 when viewed with GSView and close to that value in this pdf file:

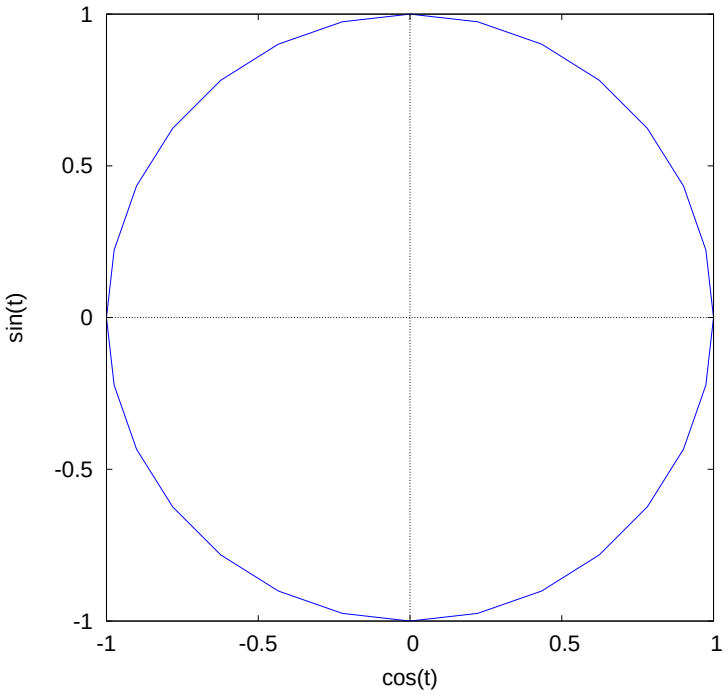

Figure 6: plot2d (adding set size ratio 1 option)

We conclude that the gnuplot preamble method of getting an approximate circle works quite well for an eps graphics file, and we will use that method to produce figures for this pdf file.

The alternative approach is to handset the x-range and y-range experimentally until the resulting "circle" measures the same width as height, for example forcing the horizontal canvas width to be, say, 1.6 as large as the vertical canvas height.

```
(%i4) plot2d ([parametric, cos(t), sin(t), ['t, -%pi, %pi] ], ['x,-1.6,1.6])$
```

has a height to width ratio of about 1.0 (fullscreen Gnuplot window) on the author's equipment. Notice above that the vertical range is determined by the curve properties and extends over **(y= −1, y = 1)**. The y-range here is 2, the x-range is 3.2, so the x-range is 1.6 times the y-range.

We now make a plot consisting of two plot objects, the first being the explicit expression **u^3**, and the second being the parametric plot object used above. We now need the syntax

```
plot2d ([plot-object-1, plot-object-2], possibly-required-draw-range-control,
                    other-option-lists )
```

Here is an example :

```
(%i5) plot2d ([u^3,[parametric, cos(t), sin(t), ['t,-%pi,%pi]]],
                ['u,-1.1,1.1],['x,-1.5,1.5],['y,-1.5,1.5],
                 [gnuplot_preamble,"set size ratio 1;"])$
```

in which **['u,-1.1,1.1]** is required to determine the drawing range of **u^3**, and we have added separate horizontal and vertical canvas control lists as well as the gnuplot_preamble option to approximate a circle, since it is quicker than fiddling with the x and y ranges by hand.

To get the corresponding eps file figure for incorporation in this pdf file, we used the code

```
plot2d ([u^3,[parametric, cos(t), sin(t), ['t,-%pi,%pi]]],
                ['u,-1.1,1.1],['x,-1.5,1.5],['y,-1.5,1.5],
                 [gnuplot_preamble,"set size ratio 1;"],
                 [psfile,"ch2p07.eps"])$
```

which produces

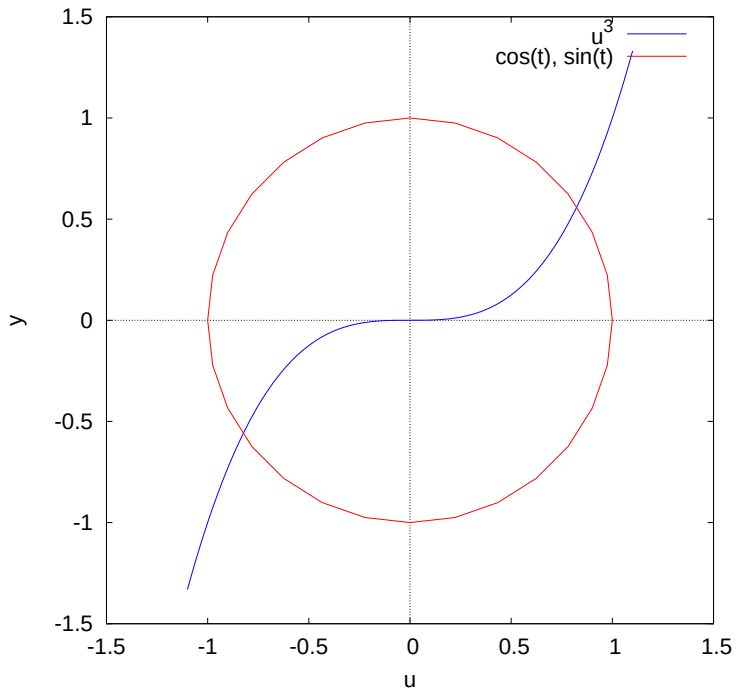

Figure 7: Combining an Explicit Expression with a Parametric Object

in which the horizontal axis label (by default) is **u**.

We now add a few more options to make this combined plot look cleaner and brighter (more about some of these later).

```
(%i6) plot2d (
              [ [parametric, cos(t), sin(t), ['t,-%pi,%pi],[nticks,200]],u^3],
                ['u,-1,1], ['x,-1.2,1.2] , ['y,-1.2,1.2],
                [style, [lines,8]],  [xlabel," "], [ylabel," "],
                [box,false], [axes, false],
                [legend,false],[gnuplot_preamble,"set size ratio 1;"])$
```

and the corresponding eps file (with **[lines,20]** for increased line width) produces:

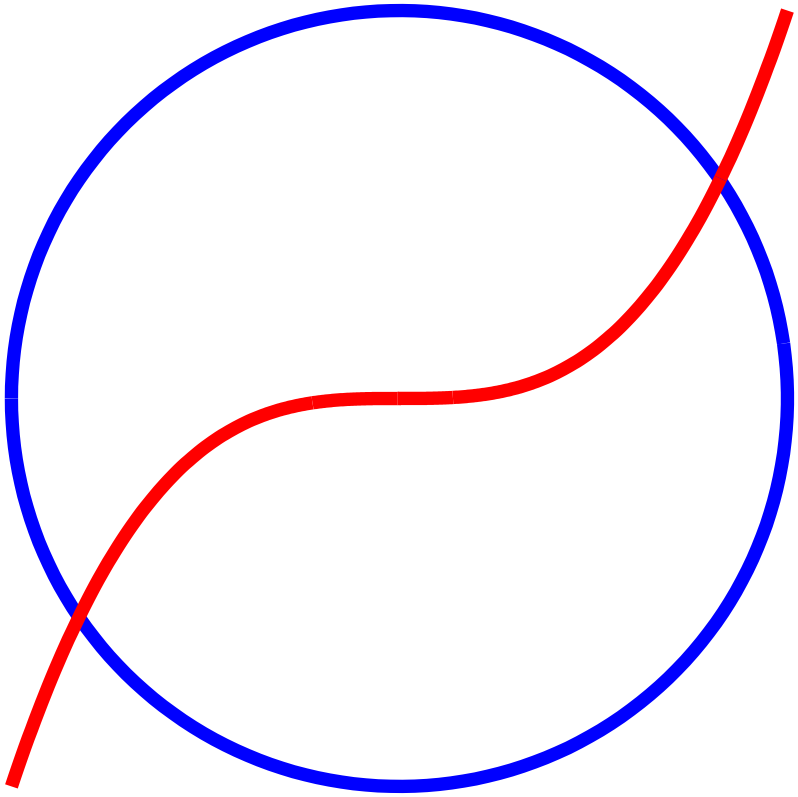

Figure 8: Drawing of $\mathbf{u^3}$ Over a Circle

The default value of **nticks** inside **plot2d** is **29**, and using **[nticks, 200]** yields a much smoother parametric curve.

2.1.4 Line Width and Color Controls

Each element to be included in the plot can have a separate **[lines, nlw, nlc]** entry in the **style** option list, with **nlw** determining the line width and **nlc** determining the line color. The default value of **nlw** is **1**, a very thin weak line. The use of **nlw = 5** creates a strong wider line.

The default color choices (if you don't provide a specific vlue of **nlc**) consist of a rotating color scheme which starts with **nlc = 1** (blue) and then progresses through **nlc = 6** (black) and then repeats.

You will see the colors with the associated values of **nlc**, using the following code which draws a set of thick vertical lines in various colors (a version of a histogram). This code also shows an example of using **discrete** list objects (to draw thick straight lines), and the use of various options available. You should run this code with your own hardware-software setup, to see what the default **plot2d** colors are with your system.

```
(%i1) plot2d(
         [ [discrete,[[0,0],[0,5]]], [discrete,[[2,0],[2,5]]],
           [discrete,[[4,0],[4,5]]],[discrete,[[6,0],[6,5]]],
           [discrete,[[8,0],[8,5]]],[discrete,[[10,0],[10,5]]],
           [discrete,[[12,0],[12,5]]],[discrete,[[14,0],[14,5]]] ],

           [style,  [lines,30,0],[lines,30,1],[lines,30,2],
           [lines,30,3],[lines,30,4],[lines,30,5],[lines,30,6],
           [lines,30,7]],

           ['x,-2,20],   ['y,-2,8],
           [legend,"0","1","2","3","4","5","6","7"],
           [xlabel," "],   [ylabel," "],
           [box,false],[axes,false])$
```

Note that none of the objects being drawn are expressions or functions, so a draw parameter range list is not only not necessary but would make no sense, and that the optional horizontal canvas width control list above is **['x,-2,20]**.

The interactive gnuplot **plot2d** colors available on a Windows XP system thus are: **0 = black, 1 = blue, 2 = red, 3 = light green**, **4 = violet, 5 = very dark brown, 6 = black, 7 = blue, 8 = red, ...**

We cannot use **plot2d** to produce an eps figure (to use in this pdf) which will show the true **plot2d** colors since **plot2d** access to postscript **eps** file colors are (with version 5.25.1 and the author's Windows XP system): **0 = light blue or aquamarine, 1 = dark blue, 2 = red**, **3 = light green, 4 = violet, 5 = black, 6 = light blue or aquamarine, 7 = dark blue**. You can check this for your own system by adding the plot2d item **[psfile,"ztest1.eps"]** to the above code and looking at the result using GSView.

Instead we use **draw2d**. We must first load the **draw** package.

```
(%i2) load(draw);
(%o2) C:/PROGRA~1/MAXIMA~1.1-G/share/maxima/5.25.1/share/draw/draw.lisp
```

You can then interactively create a histogram with the following code using **draw2d** (or **wxdraw2d**, if you wish to make an inline plot using **wxmaxima**: just replace "draw2d" with "wxdraw2d").

```
(%i3) draw2d(
      xrange = [-2,20],
      yrange = [-3,8],
      axis_left = false,
      axis_bottom = false,
      axis_top = false,
      axis_right = false,
      xtics = 'none,
      ytics = 'none,
      fill_density = 1,
      key = "0",      fill_color = black,
      bars([0,5,1]),
      key = "1",      fill_color = blue,
      bars([2,5,1]),
      key = "2",      fill_color = red,
      bars([4,5,1]),
      key = "3",      fill_color = light-green,
      bars([6,5,1]),
      key = "4",      fill_color = violet,
      bars([8,5,1]),
```

```
      key = "5",      fill_color = "#803300",
      bars([10,5,1]),
      key = "6",      fill_color = black,
      bars([12,5,1]),
      key = "7",      fill_color = blue,
      bars([14,5,1]))$
```

We have used a hexidecimal value for one **fill_color** assignment in order to get a very dark brown (almost black) color, which is what the default **plot2d** number 5 color looks like with a Windows XP system.

To get an eps figure suitable for inclusion at this point, we then add several lines to the end of the above code:

```
draw2d(
      xrange = [-2,20],
      yrange = [-3,8],
      axis_left = false,
      axis_bottom = false,
      axis_top = false,
      axis_right = false,
      xtics = 'none,
      ytics = 'none,
      fill_density = 1,
      key = "0",      fill_color = black,
      bars([0,5,1]),
      key = "1",      fill_color = blue,
      bars([2,5,1]),
      key = "2",      fill_color = red,
      bars([4,5,1]),
      key = "3",      fill_color = light-green,
      bars([6,5,1]),
      key = "4",      fill_color = violet,
      bars([8,5,1]),
      key = "5",      fill_color = "#803300",
      bars([10,5,1]),
      key = "6",      fill_color = black,
      bars([12,5,1]),
      key = "7",      fill_color = blue,
      bars([14,5,1]),
      dimensions = [1000,800],
      terminal = 'eps_color,
      file_name = "plot2d_default_colors")$
```

which code produces the file **plot2d_default_colors.eps**.

Here is the result:

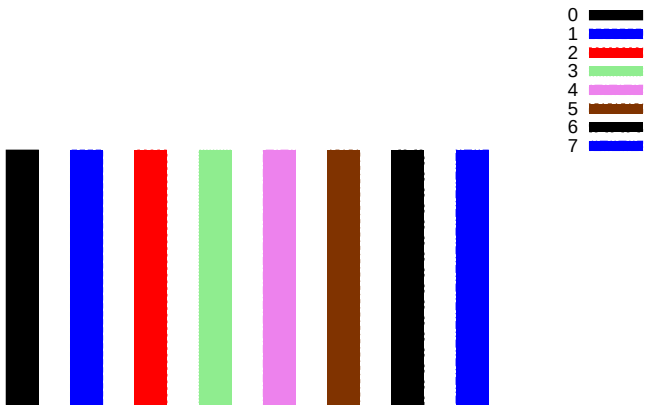

Figure 9: Cyclic **plot2d** Colors with a Windows XP System

For a simple example which uses color and line width controls, we plot the expressions **u^2** and **u^3** on the same canvas, using lines in black and red colors, and add a height control list, which has the syntax **['y, ymin, ymax]**.

```
(%i4) plot2d( [u^2,u^3],['u,0,2], ['x, -.2, 2.5],
          [style, [lines,5,6],[lines,5,2]],
           ['y,-1,4] )$
plot2d: some values were clipped.
```

The plot2d warning should not be of concern here.

The width and height control list parameters have been chosen to make it easy to see where the two curves cross for positive **u**. If you move the cursor over the crossing point, you can read off the coordinates from the cursor position printout in the lower left corner of the plot window.

This produces the plot:

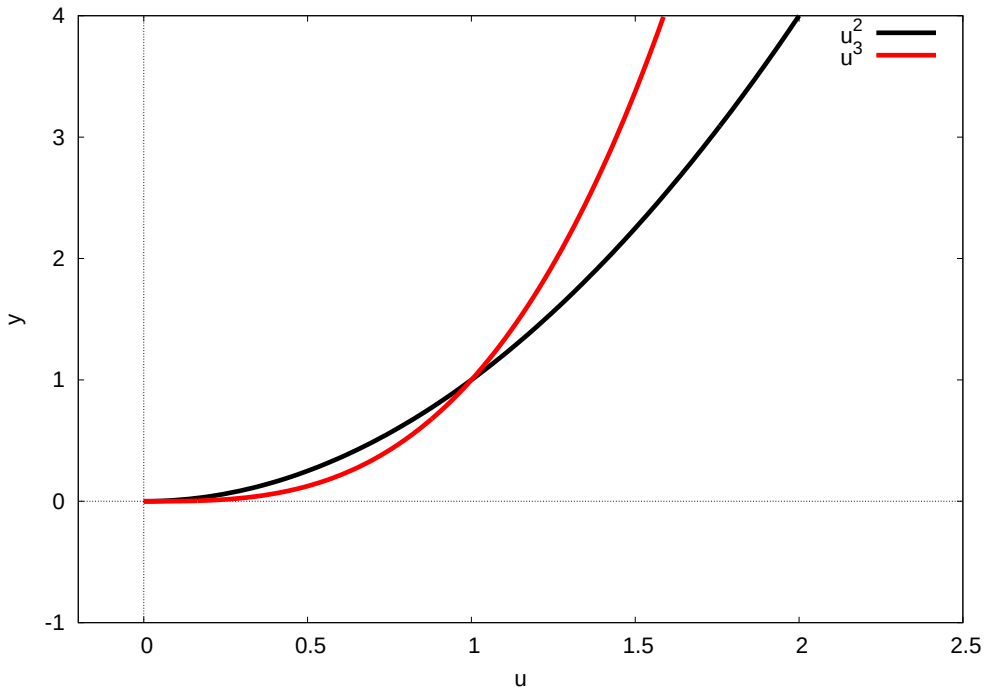

Figure 10: Black and Red Curves

This figure was included in this chapter by creating an eps file version (for inclusion in a latex file) using the code

```
plot2d( [u^2, u^3],['u,0,2], ['x, -.2, 2.5],
          [style, [lines,5,5],[lines,10,2]],
           ['y,-1,4],[psfile, "ch2p11.eps"])$
```

(Note again that when using **plot2d** to create eps files, we must pay attention to some color differences (eps black is nlc = 5).) Also, this eps file version creates the key legend u^2 (for example) instead of the **plot2d** console window version which has **u^2** as the legend.

2.1.5 Discrete Data Plots: Point Size, Color, and Type Control

We have seen some simple parametric plot examples above. Here we make a more elaborate plot which includes discrete data points which locate on the curve special places, with information on the key legend about those special points. The actual parametric curve color is chosen to be black (6) with some thickness (4), using **[lines,4,6]** in the **style** list. We force large size points with special color choices, using the maximum amount of control in the **[points, nsize, ncolor, ntype]** style assignments.

```
(%i1) obj_list : [ [parametric, 2*cos(t),t^2,['t,0,2*%pi]],
                     [discrete,[[2,0]]],[discrete,[[0,(%pi/2)^2]]],
               [discrete,[[-2,%pi^2]]],[discrete,[[0,(3*%pi/2)^2]]] ]$
(%i2) style_list : [style, [lines,4,6],[points,5,1,1],[points,5,2,1],
                    [points,5,3,1],[points,5,4,1]]$
(%i3) legend_list : [legend, " ","t = 0","t = pi/2","t = pi",
                   " t = 3*pi/2"]$
(%i4) plot2d( obj_list, ['x,-3,4], ['y,-1,40],style_list,
                   [xlabel,"X = 2 cos( t ), Y = t ^2 "],
                    [ylabel, " "] ,legend_list )$
```

This produces the plot:

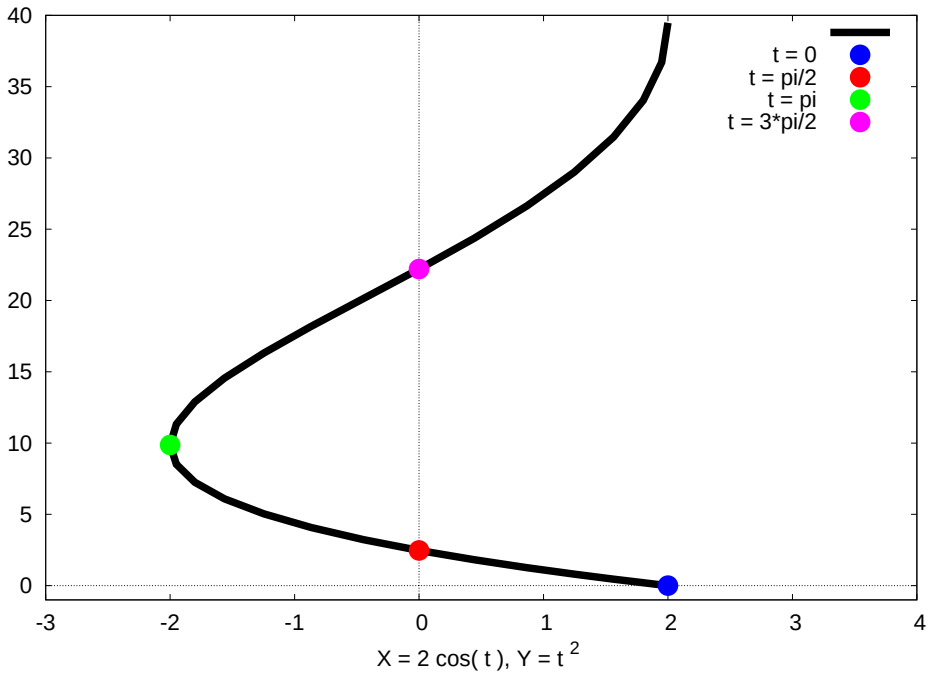

Figure 11: Parametric Plot with Discrete Points

The **points** style option has any of the following forms: **`[points]`**, or **`[points, point_size]`**, or **`[points, point_size, point_color]`**, or **`[points, point_size, point_color, point_type]`**, in order of increasing control.

Using the option **`[style, [points]]`** has the same result as using the option **`[style, [points,3,1,1]]`** , which (with the author's Windows system) results in all plotted points being the same: a decent size filled blue circle. For comparison, the smallest point size **`1`** (instead of **`3`**) is rather small. We can thus regard the option **`[style, [points]]`** as a convenient default point style for quick work.

The default point colors are the same cyclic colors used by the **lines** style. The default point type is a cyclic order starting with **1 = filled circle** , then continuing **2 = open circle**, **3 = plus sign**, **4 = diagonal crossed lines as capital X**, **5 = star**, **6 = filled square**, 7 = open square, 8 = filled triangle point up, **9 = open triangle point up**, **10 = filled triangle point down**, **11 = open triangle point down 12 = filled diamond**, **13 = open diamond = 0**, **14 = filled circle**, which is the same as **1**, and repeating the same cycle. Thus if you use **`[points,5]`** you will get good size points, and both the color and shape will cycle through the default order. If you use **`[points,5,2]`** you will force a red color but the shape will depend on the contents and order of the rest of the objects list.

You can see the default points cycle through both colors and shapes with the code:

```
plot2d([ [discrete,[[0,.5]]],[discrete,[[0.1,.5]]],
                  [discrete,[[0.2,.5]]],[discrete,[[0.3,.5]]],
                 [discrete,[[0.4,.5]]],[discrete,[[0.5,.5]]],
                  [discrete,[[0.6,.5]]],[discrete,[[0.7,.5]]],
                  [discrete,[[0,0]]],[discrete,[[0.1,0]]],
                  [discrete,[[0.2,0]]],[discrete,[[0.3,0]]],
                  [discrete,[[0.4,0]]],[discrete,[[0.5,0]]],
                  [discrete,[[0.6,0]]],[discrete,[[0.7,0]]]   ],
```

```
                [style,[points,5]],[xlabel,""],[ylabel,""],
                ['x,-0.2,1],['y,-0.2,0.7],
                [box,false],[axes,false],[legend,false])$
```

which will produce something like

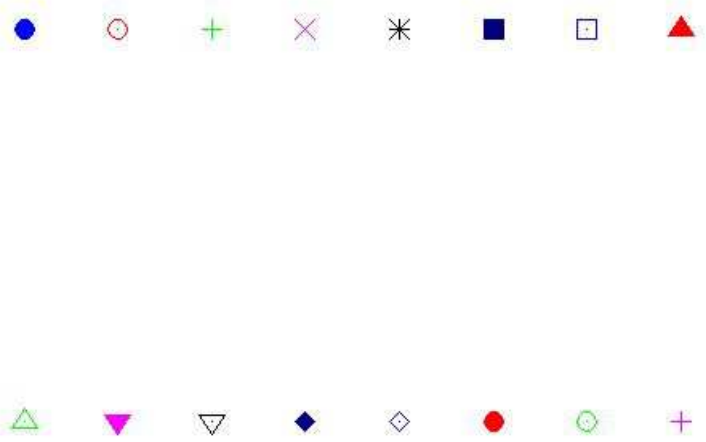

Figure 12: default point colors and styles, 1-8, then 9-16

You can also experiment with one shape at a time by defining a function **dopt(n)** which selects the shape with the integer **n** and leaves the color blue:

```
dopt(n) := plot2d ([discrete,[[0,0]]],[style,[points,15,1,n]],
                   [box,false],[axes,false],[legend,false])$
```

Next we combine a list of twelve (x,y) pairs of points with the key word **discrete** to form a discrete object type for **plot2d**, and then look at the data points without adding the optional canvas width control. Note that using only one discrete list for all the points results in all data points being displayed with the same size, color and shape.

```
(%i5) data_list : [discrete,
         [ [1.1,-0.9],[1.45,-1],[1.56,0.3],[1.88,2],
              [1.98,3.67],[2.32,2.6],[2.58,1.14],
           [2.74,-1.5],[3,-0.8],[3.3,1.1],
           [3.65,0.8],[3.72,-2.9] ] ]$
(%i6) plot2d( data_list, [style,[points]])$
```

This produces the plot

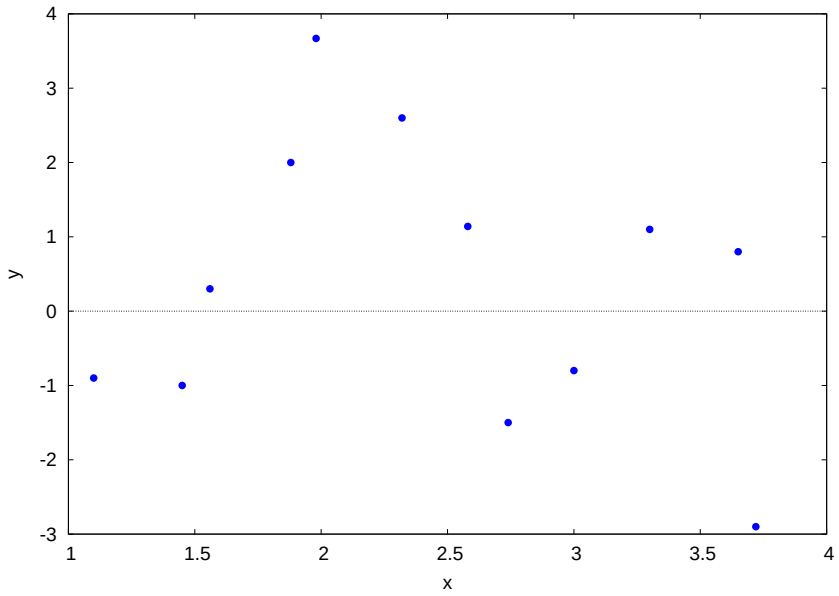

Figure 13: Twelve Data Points with Same Size, Color, and Shape

We now combine the data points with a curve which is a possible fit to these data points over the draw parameter range `[u, 1, 4]`.

```
(%i7) plot2d( [sin(u)*cos(3*u)*u^2, data_list],
                ['u,1,4], ['x,0,5],['y,-10,8],
           [style,[lines,4,1],[points,4,2,1]])$
plot2d: some values were clipped.
```

which produces (approximately) the plot

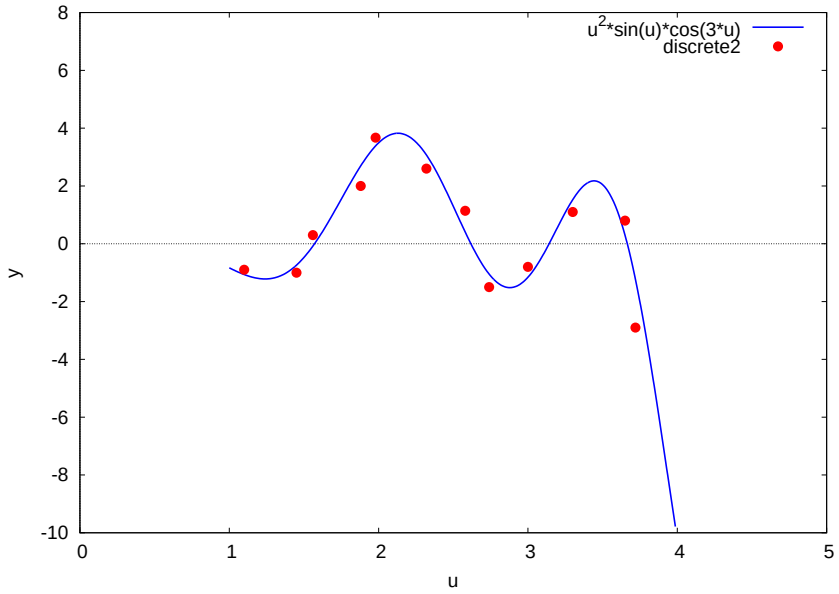

Figure 14: Curve Plus Data

2.1.6 More gnuplot_preamble Options

Here is an example of using the **gnuplot_preamble** options to add a grid, a title, and position the plot key at the bottom center of the canvas. Note the use of a semi-colon between successive gnuplot instructions.

```
(%i1) plot2d([ u*sin(u),cos(u)],['u,-4,4] ,['x,-8,8],
              [style,[lines,5]],
          [gnuplot_preamble,"set grid; set key bottom center;
               set title 'Two Functions';"])$
```

which produces (approximately) the plot

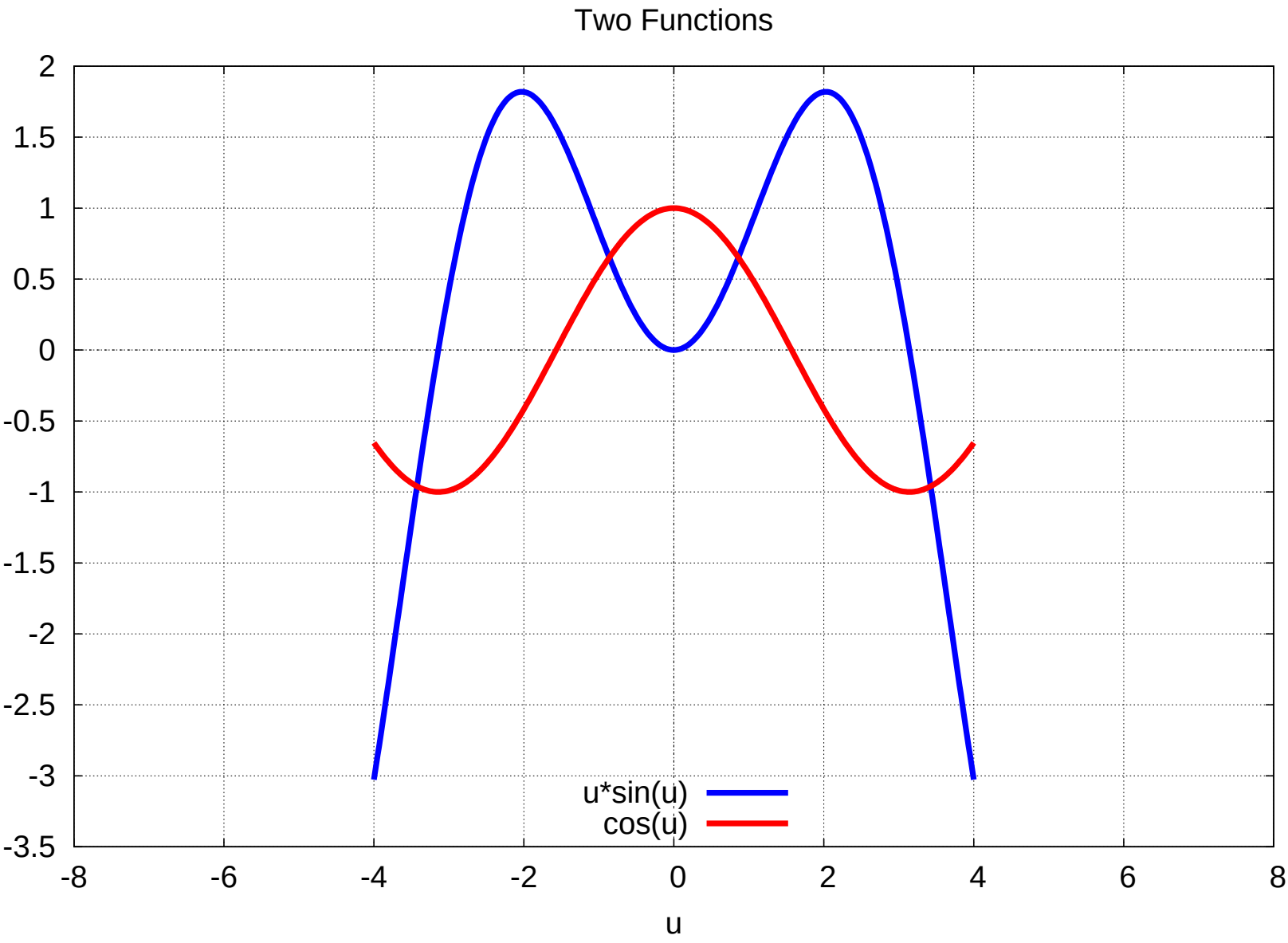

Figure 15: Using the gnuplot_preamble Option

Another option you can use is **`set zeroaxis lw 5;`** to get more prominent **`x`** and **`y`** axes. Another example of a key location would be **`set key top left;`**. We have also previously used **`set size ratio 1;`** to get a "rounder" circle.

2.1.7 Creating Various Kinds of Graphics Files Using plot2d

The first method of exporting **plot2d** drawings as special graphics files is to have the plot drawn in the Gnuplot window (ie., the Xmaxima route, or using **plot2d** rather than **wxplot2d** if you are using **wxmaxima**). Then left-click the Gnuplot icon in the upper-left-hand corner of the Gnuplot window, choose Options, Copy to Clipboard. Then open an application which accomodates the graphics format you desire, and paste the clipboard image into the application, and then use Save As, selecting the graphics type of save desired.

The second method of exporting **plot2d** drawings as special graphics files is to use the **gnuplot_term** option as part of your **plot2d** command. If you do not add an additional option of the form

```
  [gnuplot_out_file, "myname.ext"]
```

where **ext** is replaced by an appropriate graphics type extension, then **plot2d** creates a file with the name **maxplot.ext** in your current working directory.

For example,

```
(%i1) plot2d (sin(u),['u,0,%pi], [gnuplot_term,'jpeg])$
```

will create the graphics file **maxplot.jpeg**, and a further commmand

```
(%i2) plot2d (cos(u),['u,0,%pi], [gnuplot_term,'jpeg])$
```

will overwrite the previous file **maxplot.jpeg** to create the new graphics file for the plot of **cos**. To provide a different name for different plots, you would write, for example,

```
(%i3) plot2d (cos(u),['u,0,%pi],
        [gnuplot_out_file,"mycos1.jpeg"],
       [gnuplot_term,'jpeg])$
```

or

```
(%i4) plot2d (cos(u),['u,0,%pi],
        [gnuplot_out_file,"c:/work2/mycos2.jpg"],
       [gnuplot_term,'jpeg])$
```

Turning to other graphics file formats, and ignoring the naming option part,

```
(%i5) plot2d (sin(u),['u,0,%pi],
       [gnuplot_term,'png])$
```

will create **maxplot.png**.

```
(%i6) plot2d (sin(u),['u,0,%pi],
       [gnuplot_term,'eps])$
```

will create **maxplot.eps**.

```
(%i7) plot2d (sin(u),['u,0,%pi],
       [gnuplot_term,'svg])$
```

will create **maxplot.svg** (a "scalable vector graphics" file openable by **inkscape**).

```
(%i8) plot2d (sin(u),['u,0,%pi],
       [gnuplot_term,'pdf])$
```

will create **maxplot.pdf**. This last example does not really work with a Windows system unless the plot has been designed to have no labels, since otherwise latin letters are converted into Greek letters in the resulting **pdf** file.

2.1.8 Using qplot for Quick Plots of One or More Functions

The file **qplot.mac** is posted with Ch. 2 and contains a function called **qplot** which can be used for quick plotting of functions in place of **plot2d**.

The function **qplot** (**q** for "quick") accepts the default cyclic colors but always uses thicker lines than the **plot2d** default, adds more prominent x and y axes to the plot, and adds a grid. Here are some examples of use. (We include use with **discrete** lists only for completeness, since there is no way to get the **points** style with **qplot**.)

```
(%i1) load(qplot);
(%o1)                          c:/work2/qplot.mac
(%i2) qplot(sin(u),['u,-%pi,%pi])$
(%i3) qplot(sin(u),['u,-%pi,%pi],['x,-4,4])$
(%i4) qplot(sin(u),['u,-%pi,%pi],['x,-4,4],['y,-1.2,1.2])$
(%i5) qplot([sin(u),cos(u)],['u,-%pi,%pi])$
(%i6) qplot([sin(u),cos(u)],['u,-%pi,%pi],['x,-4,4])$
(%i7) qplot([sin(u),cos(u)],['u,-%pi,%pi],['x,-4,4],['y,-1.2,1.2])$
(%i8) qplot ([parametric, cos(t), sin(t), ['t,-%pi,%pi]],
              ['x,-2.1,2.1],['y,-1.5,1.5])$
```

The last use involved only a parametric object, and the list **['x,-2.1,2.1]** is interpreted as a horizontal canvas width control list based on the symbol **x**.

The next example includes both an expression depending on the parameter **u** and a parametric object depending on a parameter **t**, so we must have a expression draw list **['u,umin,umax]**.

```
(%i9) qplot ([ u^3,
               [parametric, cos(t), sin(t), ['t,-%pi,%pi]]],
               ['u,-1,1],['x,-2.25,2.25],['y,-1.5,1.5])$
```

Here is approximatedly the result with the author's system:

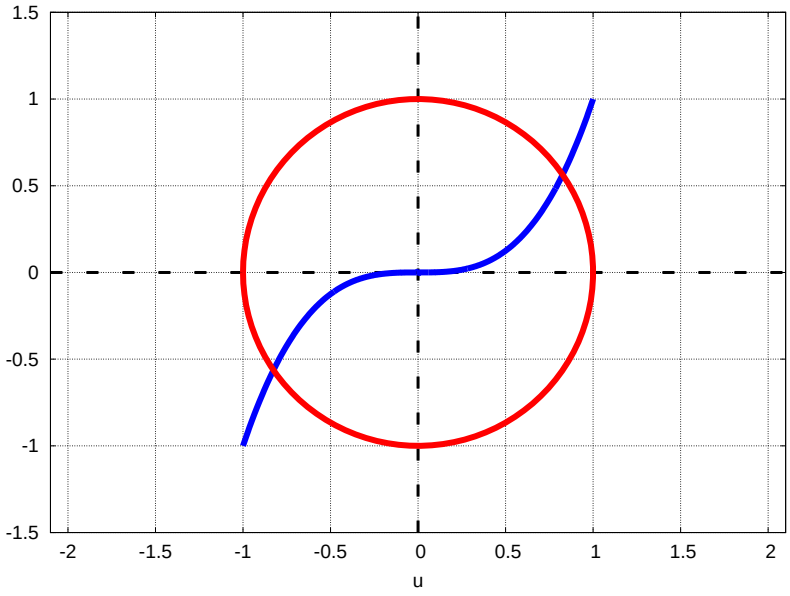

Figure 16: qplot example

To get the same plot using **plot2d** from scratch requires the code:

```
plot2d([ u^3, [parametric, cos(t), sin(t), ['t,-%pi,%pi]]],
               ['u,-1,1],['x,-2.25,2.25],['y,-1.5,1.5],
              [style,[lines,5]], [nticks,100],
            [gnuplot_preamble, "set grid; set zeroaxis lw 5;"],
             [legend,false],[ylabel, " "])$
```

Here are two **discrete** examples which draw vertical lines.

```
(%i10) qplot([discrete,[[0,-2],[0,2]]],['x,-2,2],['y,-4,4])$
(%i11) qplot( [ [discrete,[[-1,-2],[-1,2]]],
                 [discrete,[[1,-2],[1,2]]]],['x,-2,2],['y,-4,4])$
```

Here is the code (in qplot.mac) which defines the Maxima function **qplot**.

```
qplot ( exprlist, prange, [hvrange]) :=
    block([optlist, plist],
      optlist : [ [nticks,100], [legend, false],
           [ylabel, " "], [gnuplot_preamble, "set grid; set zeroaxis lw 5;"] ],
      optlist : cons ( [style,[lines,5]], optlist ),
      if length (hvrange) = 0 then plist : []
            else   plist : hvrange,
      plist : cons (prange,plist),
      plist : cons (exprlist,plist),
      plist : append ( plist, optlist ),
      apply (plot2d, plist ) )$
```

In this code, the third argument is an optional argument. The local **plist** accumulates the arguments to be passed to **plot2d** by use of **cons** and **append**, and is then passed to **plot2d** by the use of **apply**. The order of using **cons** makes sure that **exprlist** will be the first element, (and **prange** will be the second) seen by **plot2d**. In this example you can see several tools used for programming with lists.

Several choices have been made in the **qplot** code to get quick and uncluttered plots of one or more functions. One choice was to add a grid and stronger **x** and **y** axis lines. Another choice was to eliminate the key legend by using the option **[legend, false]**. If you want a key legend to appear when plotting multiple functions, you should remove that option from the code and reload **qplot.mac**.

2.1.9 Plot of a Discontinuous Function

Here is an example of a definition and plot of a discontinous function.

```
(%i12) fs(x) := if x >= -1 and x <= 1 then 3/2 else 0$
(%i13) plot2d (fs(u),['u,-2,2],['x,-3,3],['y,-.5,2],
        [style, [lines,5]],[ylabel,""],
         [xlabel,""])$
```

which produces (approximately) the plot:

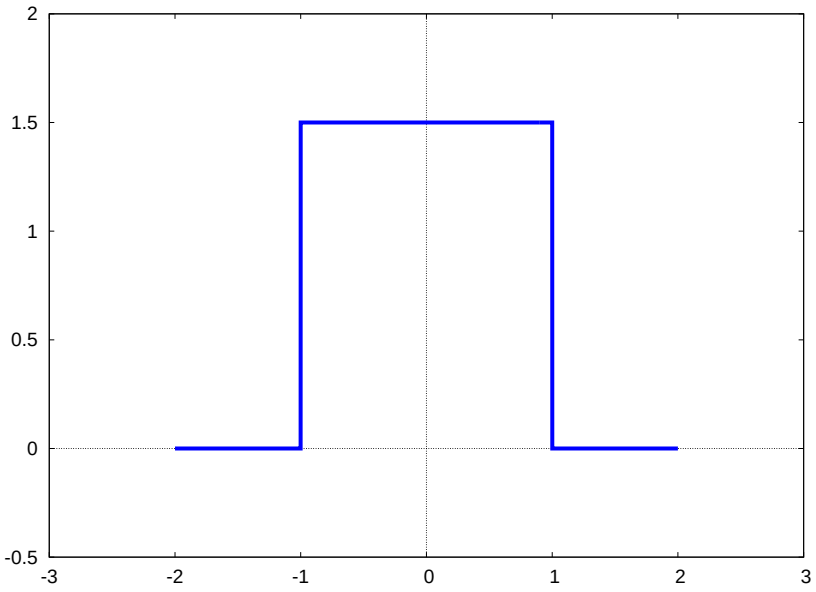

Figure 17: Plot of a Discontinuous Function

or we can use **qplot**:

```
(%i14) load(qplot);
(%o14)                          c:/work2/qplot.mac
(%i15) qplot (fs(u),['u,-2,2],['x,-3,3],['y,-.5,2],[xlabel,""])$
```

to get the same plot.

2.1.10 Multiple Plots Using the Embedded Option

The author uses the Xmaxima GUI (ver. 5.25.1) with Windows XP, and this combination allows one to get as many multiple plots as one wants using the XMaxima menu choice **Options, Plot Windows, Embedded**, and then typing plot2d commands.

A plot2d command then brings up an embedded plot which can be reduced in size using the minus icon associated with the embedded plot panel.

Another plot requires clicking the mouse at the new prompt and entering the new plot2d command.

One then has a series of plots which can be compared.

The embedded plot feature in XMaxima appears to be still “under construction”, since the various control icons are not labeled and one must experiment.

The leftmost control icon is an **X** which causes the embedded plot to vanish, apparently forever?

The next icon, reading left to right, is some sort of crossed tool icon, which may be what the help screen means by the “config menu”. The author was not successful in getting, for example, the linewidth choice to have any effect on the plot. The third icon is mysterious. The help screen was no help.

None of the above is meant to denigrate the author's favorite user interface for using Maxima (**XMaxima**), an interface which is steadily getting more useful, and is used throughout these notes, because of the relative ease of selecting and copying (with the typical M.S. Windows methods) both the Maxima input and output, and pasting the session into a text based Tex file (for the latter editing, we use **notepad++**).

2.2 Working with Files Using the Package mfiles.mac

The chapter 2 package file **mfiles.mac** (which loads **mfiles1.lisp**) has some Maxima tools for working with both text and data files. Both files are available on the author's webpage. Unless you use your initialization file to automatically load **mfiles.mac** when you start or restart Maxima, you will need to do an explicit load to use these functions.

2.2.1 Check File Existence with file_search or probe_file

To check for the existence of a file, you can use the standard Maxima function **file_search** or the **mfiles** package function **probe_file** (the latter is experimental and seems to work in a M.S. Windows version of Maxima). In the following, the file **ztemp.txt** exists in the current working directory (**c:/work2/**), and the file **ytemp.txt** does not exist in this work folder. Both functions return **false** when the file is not found. You need to supply the full file name including any extension, as a string.

The XMaxima output shown here uses **display2d:true** (the default). If you use the non-default setting (**display2d:false**), strings will appear surrounded by double-quotes, but unpleasant backslash escape characters **** will appear in the output.

```
(%i1) file_search ("ztemp.txt");
(%o1)                          c:/work2/ztemp.txt
(%i2) probe_file ("ztemp.txt");
(%o2)                         probe_file(ztemp.txt)
(%i3) load(mfiles);
(%o3)                          c:/work2/mfiles.mac
(%i4) probe_file ("ztemp.txt");
(%o4)                                  false
(%i5) probe_file ("c:/work2/ztemp.txt");
(%o5)                          c:/work2/ztemp.txt
(%i6) myf : mkp("ztemp.txt");
(%o6)                          c:/work2/ztemp.txt
(%i7) probe_file (myf);
(%o7)                          c:/work2/ztemp.txt
(%i8) file_search ("ztemp");
(%o8)                                  false
(%i9) file_search ("ytemp.txt");
(%o9)                                  false
```

Although the core Maxima function **file_search** does not need the complete path (due to our **maxima.init** file contents), our homemade functions, such as **probe_file** do need a complete path, due to recent changes in Maxima. To ease the pain of typing the full path, you can use a function which I call **mkp** ("make path") which I define in my **maxima.init** file, whose contents are:

```
/* this is c:\Documents and Settings\Edwin Woollett\maxima\maxima-init.mac  */
maxima_userdir: "c:/work2" $
maxima_tempdir : "c:/work2"$
file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$
file_search_lisp : append(["c:/work2/###.lisp"],file_search_lisp )$

bpath : "c:/work2/"$
mkp (_fname) := sconcat (bpath,_fname)$
```

We have used this function, **mkp** above, to get **probe_file** to work. The string processing function **sconcat** creates a new string from two given strings ("string concatenation"):

```
(%i10) display2d:false$
(%i11) sconcat("a bc","xy z");
(%o11) "a bcxy z"
```

Note that we used a global variable **bpath** ("base of path" or "beginning of path") instead of the global variable **maxima_userdir**. This makes it more convenient for the user to redefine **bpath** "on the fly" instead of opening and editing **maxima-init.mac**, and restarting Maxima to have the changes take effect.

We see that **file_search** is easier to use than **probe_file**.

2.2.2 Check for File Existence using ls or dir

The **mfiles** package functions **ls** and **dir** accept the use of a wildcard pathname. Both functions are experimental and are not guaranteed to work with Maxima engines compiled with a Lisp version different from GCL (Gnu Common Lisp) (which is the Lisp version used for the Windows binary used by the author).

Again, you must use the full path, and can make use of the **mkp** function as in the above examples. The last examples here refer to a folder different than the working folder **c:/work2/**.

```
(%i12) ls(mkp("*temp.txt"));
(%o12) [c:/work2/REPLACETEMP.TXT, c:/work2/temp.txt, c:/work2/ztemp.txt]
(%i13) ls("c:/work2/*temp.txt");
(%o13) [c:/work2/REPLACETEMP.TXT, c:/work2/temp.txt, c:/work2/ztemp.txt]
(%i14) dir(mkp("*temp.txt"));
(%o14)              [REPLACETEMP.TXT, temp.txt, ztemp.txt]
(%i15) dir("c:/work2/*temp.txt");
(%o15)              [REPLACETEMP.TXT, temp.txt, ztemp.txt]
(%i16) ls ("c:/work3/dirac.*");
(%o16)             [c:/work3/dirac.mac, c:/work3/dirac.tex]
(%i17) dir ("c:/work3/dirac.*");
(%o17)                       [dirac.mac, dirac.tex]
```

2.2.3 Type of File, Number of Lines, Number of Characters

The text file lisp1w.txt is a file with Windows line ending control characters which has three lines of text and no blank lines.

```
(%i18) file_search("lisp1w.txt");
(%o18)                         c:/work2/lisp1w.txt
(%i19) file_lines("lisp1w.txt");
openr: file does not exist: lisp1w.txt
#0: file_lines(fnm=lisp1w.txt)(mfiles.mac line 686)
 -- an error. To debug this try: debugmode(true);
(%i20) file_lines(mkp("lisp1w.txt"));
(%o20)                                 [3, 3]
(%i21) file_lines("c:/work2/lisp1w.txt");
(%o21)                                 [3, 3]
```

The output of the **file_lines** function returns the list
[number of non-blank lines, total number of lines].

```
(%i22) myf:mkp("lisp1w.txt");
(%o22)                         c:/work2/lisp1w.txt
(%i23) ftype(myf);
(%o23)                               windows
```

```
(%i24) file_length(myf);
(%o24)                                131
(%i25) file_info(myf);
(%o25)                       [3, 3, windows, 131]
```

The function **ftype** (file type) returns either **windows**, **unix**, or **mac** depending on the nature of the "end of line chars". The function **file_length** returns the number of characters ("chars") in the file, including the end of line chars. The function **file_info** combines the line number info, the file type, and the number of characters into one list.

2.2.4 Print All or Some Lines of a File to the Console

For a small file the packge function **print_file(file)** is useful. For a larger file **print_lines(file,start,end)** is useful.

```
(%i26) print_file (myf)$
Lisp (or LISP) is a family of computer programming
languages with a long history and a distinctive, fully
parenthesized syntax.
(%i27) myf : mkp("lisp2.txt");
(%o27)                         c:/work2/lisp2.txt
(%i28) file_info(myf);
(%o28)                       [8, 8, windows, 504]
(%i29) print_lines(myf,3,5)$
parenthesized syntax. Originally specified in 1958, Lisp is
the second-oldest high-level programming language in
widespread use today; only Fortran is older (by one year).
```

2.2.5 Rename a File using rename_file

The **mfiles.mac** package function **rename_file (oldname,newname)** is an experimental function which works as follows in a Windows version of Maxima (again we must use the complete path):

```
(%i30) file_search("foo1.txt");
(%o30)                                false
(%i31) file_search("bar1.txt");
(%o31)                         c:/work2/bar1.txt
(%i32) rename_file(mkp("bar1.txt"),mkp("foo1.txt"));
(%o32)                         c:/work2/foo1.txt
(%i33) file_search("foo1.txt");
(%o33)                         c:/work2/foo1.txt
(%i34) file_search("bar1.txt");
(%o34)                                false
```

2.2.6 Delete a File with delete_file

The **mfiles.mac** package function **delete_file (filename)** does what its name implies:

```
(%i35) file_search("bar2.txt");
(%o35)                         c:/work2/bar2.txt
(%i36) delete_file(mkp("bar2.txt"));
(%o36)                                done
(%i37) file_search("bar2.txt");
(%o37)                                false
```

2.2.7 Copy a File using copy_file

The **mfiles.mac** package function **copy_file(fsource,fdest)** will preserve the file type, but will not warn you if you are over-writing a previous file name.

```
(%i38) file_search("foo2.txt");
(%o38)                                false
(%i39) file_info(mkp("foo1.txt"));
(%o39)                          [3, 3, windows, 59]
(%i40) copy_file(mkp("foo1.txt"),mkp("foo2.txt"));
(%o40)                           c:/work2/foo2.txt
(%i41) file_info(mkp("foo2.txt"));
(%o41)                          [3, 3, windows, 59]
```

2.2.8 Change the File Type using file_convert

Here, "file type" refers to the text file types **unix**, **windows**, and **mac**, each distinguised by the use of different conventions in indicating the end of a text file line.

Syntax: **file_convert(file, newtype)** or
file_convert (oldfile, newfile,newtype)
The acceptable values of **newtype** are **windows**, **unix**, and **mac**.

For example, we can change a unix file to a windows file using **file_convert(f,windows)** which replaces the previous file, or we can change a unix file to a windows file with a new name using **file_convert(fold,fnew,windows)**.

It is easy to check the end of line characters using the Notepad2 View menu. Notepad2 also easily lets you change the end of line characters, so writing Maxima code for this task is somewhat unneeded, though instructive as a coding challenge.

```
(%i42) file_search("bar1.txt");
(%o42)                            c:/work2/bar1.txt
(%i43) file_search("bar2.txt");
(%o43)                            c:/work2/bar2.txt
(%i44) print_file(mkp("bar1.txt"))$
This is line one.
This is line two.
This is line three.
(%i45) file_info(mkp("bar1.txt"));
(%o45)                          [3, 3, windows, 59]
(%i46) file_convert(mkp("bar1.txt"),mkp("bar11u.txt"),unix);
(%o46)                         c:/work2/bar11u.txt
(%i47) file_info(mkp("bar11u.txt"));
(%o47)                           [3, 3, unix, 56]
(%i48) print_file(mkp("bar2.txt"));
This is line one.
This is line two.
This is line three.
(%o48)                            c:/work2/bar2.txt
(%i49) file_info(mkp("bar2.txt"));
(%o49)                          [3, 3, windows, 59]
(%i50) file_convert(mkp("bar2.txt"),mac);
(%o50)                            c:/work2/bar2.txt
```

```
(%i51) file_info(mkp("bar2.txt"));
(%o51)                          [3, 3, mac, 56]
(%i52) print_file(mkp("bar2.txt"));
This is line one.
This is line two.
This is line three.
(%o52)                         c:/work2/bar2.txt
```

2.2.9 Breaking File Lines with pbreak_lines or pbreak()

Four paragraphs of the Lisp entry from part of Paul Graham's web site were copied into a text file **ztemp.txt**. In Notepad2, each paragraph was one long line.

Inside Maxima, we then used the **mfiles** package function **pbreak_lines (file,nmax)** to break the lines (at a space) and print the results to the console screen of Xmaxima.

```
(%i53) file_info(mkp("ztemp.txt"));
(%o53)                       [7, 13, windows, 1082]
(%i54) print_lines(mkp("ztemp.txt"),1,1);
6. Programs composed of expressions. Lisp programs are trees ...[continues]
(%o54)                         c:/work2/ztemp.txt
(%i55) pbreak_lines(mkp("ztemp.txt"),60)$
6. Programs composed of expressions. Lisp programs are
trees of expressions, each of which returns a value. (In
some Lisps expressions can return multiple values.) This is
in contrast to Fortran and most succeeding languages, which
distinguish between expressions and statements.

It was natural to have this distinction in Fortran because
(not surprisingly in a language where the input format was
punched cards) the language was line-oriented. You could
not nest statements. And so while you needed expressions
for math to work, there was no point in making anything
else return a value, because there could not be anything
waiting for it.

This limitation went away with the arrival of
block-structured languages, but by then it was too late.
The distinction between expressions and statements was
entrenched. It spread from Fortran into Algol and thence to
both their descendants.

When a language is made entirely of expressions, you can
compose expressions however you want. You can say either
(using Arc syntax)

(if foo (= x 1) (= x 2))

or

(= x (if foo 1 2))
```

Once the line breaking text appears on the console screen, one can copy and paste into a text file for further use.

It is simpler to use the function **pbreak()** which has **nmax = 72** hardwired in the code, as well as the name **"ztemp.txt"**; this could be used in the above example as **pbreak()**, and is easy to use since you don't have to type either the text file name or the value of **nmax**.

The package function **pbreak()** uses the current definition of **bpath** and **mkp**, as well as the file name **"ztemp.txt"**.

```
(%i56) pbreak();
6. Programs composed of expressions. Lisp programs are trees of
expressions, each of which returns a value. (In some Lisps expressions
can return multiple values.) This is in contrast to Fortran and most
succeeding languages, which distinguish between expressions and
statements.

It was natural to have this distinction in Fortran because (not
surprisingly in a language where the input format was punched cards)
the language was line-oriented. You could not nest statements. And so
while you needed expressions for math to work, there was no point in
making anything else return a value, because there could not be
anything waiting for it.

This limitation went away with the arrival of block-structured
languages, but by then it was too late. The distinction between
expressions and statements was entrenched. It spread from Fortran into
Algol and thence to both their descendants.

When a language is made entirely of expressions, you can compose
expressions however you want. You can say either (using Arc syntax)

(if foo (= x 1) (= x 2))

or

(= x (if foo 1 2))
(%o56)                                done
```

Alternatively, one can employ the **mfiles** package function **break_file_lines (fold,fnew,nmax)** to dump the folded lines into created file **fnew**.

```
(%i57) break_file_lines (mkp("ztemp.txt"),mkp("ztemp1.txt"),72);
(%o57)                        c:/work2/ztemp1.txt
(%i58) print_lines(mkp("ztemp1.txt"),1,2);
6. Programs composed of expressions. Lisp programs are trees of
expressions, each of which returns a value. (In some Lisps expressions
(%o58)                        c:/work2/ztemp1.txt
(%i59) file_info(mkp("ztemp.txt"));
(%o59)                       [7, 13, windows, 1082]
(%i60) file_info(mkp("ztemp1.txt"));
(%o60)                       [20, 26, windows, 1095]
```

2.2.10 Search Text Lines for Strings with search_file

the default two arg behavior:

search_file(filename, substring)

is to return line number and line text only for lines in which **substring** is a distinct word, as defined by the package function **sword**, used by the package function **wsearch**.

Using the form

search_file (filename,substring,word)

produces exactly the same as the two arg default mode above.

Using the form

search_file (filename,substring,all)

will return line numbers and line text for all lines in which the package function **ssearch** returns an integer, ie., all lines

in which the substring appears, regardless of being a distinct word.

The simplest syntax **search_file (file, search-string)** is first demonstrated with two searches of the file **ndata1.dat**, which happens to be a purely text file.

```
(%i61) file_info (mkp("ndata1.dat"));
(%o61)                       [5, 9, windows, 336]
(%i62) print_file (mkp("ndata1.dat"))$
The calculation of the effective cross section is much simplified if only

those collisions are considered for which the impact parameter is large, so

that the field U is weak and the angles of deflection are small. The

calculation can be carried out in the laboratory system, and the center

of mass frame need not be used.
(%i63) search_file (mkp("ndata1.dat"),"is")$
c:/work2/ndata1.dat
 1  The calculation of the effective cross section is much simplified if only
 3  those collisions are considered for which the impact parameter is large, so
 5  that the field U is weak and the angles of deflection are small. The

(%i64) search_file (mkp("ndata1.dat"),"is much")$
c:/work2/ndata1.dat
 1  The calculation of the effective cross section is much simplified if only
```

We next demonstrate all three possible syntax forms with a purely text file **text1.txt**.

```
(%i65) file_info (mkp("text1.txt"));
(%o65)                       [5, 5, windows, 152]
(%i66) print_file (mkp("text1.txt"))$
is this line one? Yes, this is line one.
This might be line two.
Here is line three.
I insist that this be line four.
This is line five, isn't it?
(%i67) search_file (mkp("text1.txt"),"is")$
c:/work2/text1.txt
 1  is this line one? Yes, this is line one.
 3  Here is line three.
 5  This is line five, isn't it?

(%i68) search_file (mkp("text1.txt"),"is",word)$
c:/work2/text1.txt
 1  is this line one? Yes, this is line one.
 3  Here is line three.
 5  This is line five, isn't it?

(%i69) search_file (mkp("text1.txt"),"is",all)$
c:/work2/text1.txt
 1  is this line one? Yes, this is line one.
 2  This might be line two.
 3  Here is line three.
 4  I insist that this be line four.
 5  This is line five, isn't it?
```

2.2.11 Search for a Text String in Multiple Files with search_mfiles

The most general syntax is **search_mfiles (file or path,string,options...)** in which the options recognised are **word, all, cs, ic**, used in the same way as described above for **search_file**. The simplest syntax is **search_mfiles (file or path,string)** which defaults to case sensitive (**cs**) and isolated word (**word**) as options. An example of over-riding the default behavior (**cs** and **word**) would be **search_mfiles (file or path, string,ic, all)** and the options args can be in either order.

First an example of searching one file in the working directory.

```
(%i1) load(mfiles);
(%o1)                          c:/work2/mfiles.mac
(%i2) print_file(mkp("text1.txt"))$
is this line one? Yes, this is line one.
This might be line two.
Here is line three.
I insist that this be line four.
This is line five, isn't it?
(%i3) search_mfiles(mkp("text1.txt"),"is")$
c:/work2/text1.txt
 1  is this line one? Yes, this is line one.
 3  Here is line three.
 5  This is line five, isn't it?

(%i4) search_mfiles(mkp("text1.txt"),"Is")$
(%i5)
```

Next we use a wildcard type file name for a search in the working directory.

```
(%i5) search_mfiles(mkp("ndata*.dat"),"is")$
c:/work2/ndata1.dat
 1  The calculation of the effective cross section is much simplified if only
 3  those collisions are considered for which the impact parameter is large, so
 5  that the field U is weak and the angles of deflection are small. The
```

Next we return to a search of the single file **text1.txt**, but look for lines containing the string **"is"** whether or not it is an instance of an isolated word.

```
(%i6) search_mfiles(mkp("text1.txt"),"is",all)$
c:/work2/text1.txt
 1  is this line one? Yes, this is line one.
 2  This might be line two.
 3  Here is line three.
 4  I insist that this be line four.
 5  This is line five, isn't it?
```

We now use **search_mfiles** to look for a text string in a file **atext1.txt** which is **not** in the current working directory.

```
(%i1) load(mfiles);
(%o1)                          c:/work2/mfiles.mac
(%i2) search_mfiles ("c:/work2/temp1/atext1.txt","is");
c:/work2/temp1/atext1.txt
 2  Is this line two? Yes, this is line two.
 6  This is line six, Isn't it?

(%o2)                               done
```

If you want to search **all** files in the folder **`c:/work2/temp1`**, you use the syntax:

```
(%i3) search_mfiles ("c:/work2/temp1/","is")$
c:/work2/temp1/atext1.txt
 2  Is this line two? Yes, this is line two.
 6  This is line six, Isn't it?

c:/work2/temp1/atext2.txt
 2  Is this line two? Yes, this is line two.
 6  This is line six, Isn't it?

c:/work2/temp1/calc1news.txt
 9  The organization of chapter six is then:
 96  The Maxima output is the list of the vector curl components in the
 98   a reminder to the user of what the current coordinate system is
 102  Thus the syntax is based on lists and is similar to (although better
 105  There is a separate function to change the current coordinate system.
 112    plotderiv(..) which is useful for "automating" the plotting

c:/work2/temp1/ndata1.dat
 1  The calculation of the effective cross section is much simplified if only
 3  those collisions are considered for which the impact parameter is large, so
 5  that the field U is weak and the angles of deflection are small. The

c:/work2/temp1/stavros-tricks.txt
 34  Not a bug, but Maxima doesn't know that the beta function is symmetric:

c:/work2/temp1/text1.txt
 1  is this line one? Yes, this is line one.
 3  Here is line three.
 5  This is line five, isn't it?

c:/work2/temp1/trigsimplification.txt
 13  (1) Is there a Maxima command that indicates whether expr is a product of
 76  > (1) Is there a Maxima command that indicates whether expr is a product of
 91  Well, that is inherent in their definition.  Trigreduce replaces all
 94  is sin(x)*cos(x), since the individual terms are not products of trigs.
 95  There is no built-in function which tries to find the smallest expression,
 151  is better. If the user wants to expand the contents of sin to discover
 153  is right that Maxima avoids potentially very expensive operations in

c:/work2/temp1/wu-d.txt
 1  As a dedicated windows xp user who is delighted to have windows binaries
 3  all who are considering windows use that there is no problem with keeping previous
```

2.2.12 Replace Text in File with ftext_replace

The simplest syntax **ftext_replace(file,sold,snew)** replaces distinct substrings **sold** (separate words) by **snew**.

The four arg syntax **ftext_replace(file,sold,snew,word)** does exactly the same thing.

The four arg syntax **ftext_replace(file,sold,snew,all)** instead replaces **all** substrings **sold** by **snew**, whether or not they are distinct words.

In all cases, the text file type (unix, windows, or mac) is preserved.

The package function **ftext_replace** calls the package function **replace_file_text (fsource, fdest, sold,snew, optional-mode)** which allows the replacement to occur in a newly created file with a name **fdest**.

```
(%i4) file_info(mkp("text1w.txt"));
(%o4)                        [5, 5, windows, 152]
(%i5) print_file(mkp("text1w.txt"));
is this line one? Yes, this is line one.
This might be line two.
Here is line three.
I insist that this be line four.
This is line five, isn't it?
(%o5)                         c:/work2/text1w.txt
(%i6) ftext_replace(mkp("text1w.txt"),"is","was");
(%o6)                         c:/work2/text1w.txt
(%i7) print_file(mkp("text1w.txt"));
was this line one? Yes, this was line one.
This might be line two.
Here was line three.
I insist that this be line four.
This was line five, isn't it?
(%o7)                         c:/work2/text1w.txt
(%i8) file_info(mkp("text1w.txt"));
(%o8)                        [5, 5, windows, 156]
(%i9) ftext_replace(mkp("text1w.txt"),"was","is");
(%o9)                         c:/work2/text1w.txt
(%i10) print_file(mkp("text1w.txt"));
is this line one? Yes, this is line one.
This might be line two.
Here is line three.
I insist that this be line four.
This is line five, isn't it?
(%o10)                         c:/work2/text1w.txt
```

2.2.13 Email Reply Format Using reply_to

The package function **reply_to (name-string)** reads an email message (or part of an email message) which has been dumped into the current working directory file called **ztemp.txt** (the name chosen so the file is easy to find) and writes a version of that file to the console screen with a supplied name prefixing each line, suitable for a copy/paste into a reply email message.

It will be obvious if the message lines need breaking. If so, then use **pbreak()**, which will place the broken line message on the Xmaxima console screen, which can be copied and pasted over the original contents of **ztemp.txt**.

Once you are satisfied with the appearance of the message in **ztemp.txt**, use **reply_to("Ray")** for example, which will print out on the console screen the email message with each line prefixed by **"Ray"**. This output can then be copied from the Xmaxima screen and pasted into the email message being designed as a reply.

```
(%i11) reply_to("");
>Could you file a bug report on this?  I know about some of these issues
>and am working on them (slowly).  The basic parts work, but the corner
>cases need more work (especically since my approach fails in some cases
>where the original gave correct answers).
>
(%o11)                                done
```

or

```
(%i12) reply_to(" ray")$
 ray>Could you file a bug report on this?  I know about some of these issues
 ray>and am working on them (slowly).  The basic parts work, but the corner
 ray>cases need more work (especically since my approach fails in some cases
 ray>where the original gave correct answers).
 ray>
```

2.2.14 Reading a Data File with read_data

An important advantage of **read_data** is the ability to work correctly with all three types of text files (unix, windows, and mac).

Our first example data file has the data items separated by commas on the first line and by spaces on the second line. The data items are a mixture of integers, rational numbers, strings, and floating point numbers. None of the strings contain spaces.

The function **read_data** places the items of each line in the data file into a separate list.

```
(%i13) print_file(mkp("ndata2w.dat"))$
2 , 4.8, -3/4, "xyz", -2.8e-9
3 22.2  7/8 "abc" 4.4e10
(%i14) read_data(mkp("ndata2w.dat"));
                    3
(%o14) [[2, 4.8, - -, xyz, - 2.7999999999999998E-9],
                    4
                                                          7
                                                 [3, 22.2, -, abc, 4.4E+10]]
                                                          8
(%i15) display2d:false$
(%i16) read_data(mkp("ndata2w.dat"));
(%o16) [[2,4.8,-3/4,"xyz",-2.7999999999999998E-9],[3,22.2,7/8,"abc",4.4E+10]]
(%i17) fpprintprec:8$
(%i18) read_data(mkp("ndata2w.dat"));
(%o18) [[2,4.8,-3/4,"xyz",-2.8E-9],[3,22.2,7/8,"abc",4.4E+10]]
```

This simplest syntax mode of **read_data** does not care where the commas and spaces are, they can be randomly used as data item separators, and the data is read into lists correctly:

```
(%i19) print_file(mkp("ndata2wa.dat"))$
2 , 4.8  -3/4, "xyz" -2.8e-9
3 22.2,  7/8 "abc",  4.4e10
(%i20) read_data(mkp("ndata2wa.dat"));
(%o20) [[2,4.8,-3/4,"xyz",-2.8E-9],[3,22.2,7/8,"abc",4.4E+10]]
```

Next is a case in which the data item separator is consistently a semicolon **;** . In such a case we must include as a second argument to **read_data** the string **";"**.

```
(%i21) print_file(mkp("ndata3w.dat"))$
2.0; -3/7; (x:1,y:2,x+y); block([fpprec:24],bfloat(%pi)); foo
(%i22) read_data(mkp("ndata3w.dat"),";");
(%o22) [[2.0,-3/7,(x:1,y:2,y+x),block([fpprec:24],bfloat(%pi)),foo]]
```

(If some of the data items include semicolons, then you would not want to use semicolons as data item separators; rather you might choose a dollar sign **$** as the separator, and so indicate as the second argument.)

Our next example is a data file which includes some strings which include spaces inside the strings. The data file should use commas as data item separators, and the correct syntax to read the data is **read_data(file,",")**.

```
(%i23) print_file(mkp("ndata6.dat"));
1, 2/3, 3.4, 2.3e9, "file ndata6.dat"
"line two" , -3/4 , 6 , -4.8e-7 , 5.5
7/13, "hi there", 8, 3.3e4, -7.3
4,-3/9,"Jkl", 44.6, 9.9e-6
(%o23) "c:/work2/ndata6.dat"
(%i24) read_data(mkp("ndata6.dat"),",");
(%o24) [[1,2/3,3.4,2.3E+9,"file ndata6.dat"],["line two",-3/4,6,-4.8E-7,5.5],
        [7/13,"hi there",8,33000.0,-7.3],[4,-1/3,"Jkl",44.6,9.9E-6]]
```

The package function **read_data** ignores blank lines in the data file (as it should):

```
(%i25) print_file(mkp("ndata10w.dat"))$
2  4.8  -3/4  "xyz"  -2.8e-9

2   4.8  -3/4  "xyz"  -2.8e-9

2   4.8  -3/4  "xyz"  -2.8e-9

(%i26) read_data(mkp("ndata10w.dat"));
(%o26) [[2,4.8,-3/4,"xyz",-2.8E-9],[2,4.8,-3/4,"xyz",-2.8E-9],
        [2,4.8,-3/4,"xyz",-2.8E-9]]
(%i27) file_info (mkp("ndata10w.dat"));
(%o27) [3,6,windows,98]
```

2.2.15 File Lines to List of Strings using read_text

The package function **read_text(path)** preserves blank lines in the source file, and returns a list of strings, one for each physical line in the source file.

```
(%i28) print_file(mkp("ndata1.dat"))$
The calculation of the effective cross section is much simplified if only

those collisions are considered for which the impact parameter is large, so

that the field U is weak and the angles of deflection are small. The

calculation can be carried out in the laboratory system, and the center

of mass frame need not be used.

(%i29) read_text(mkp("ndata1.dat"));
(%o29) ["The calculation of the effective cross section is much simplified if only",
        "",
        "those collisions are considered for which the impact parameter is large, so",
        "",
        "that the field U is weak and the angles of deflection are small. The",
        "",
        "calculation can be carried out in the laboratory system, and the center",
        "","of mass frame need not be used."]
```

2.2.16 Writing Data to a Data File One Line at a Time Using with_stdout

The core Maxima function **with_stdout** can be used to write loop results to a file instead of to the screen. This can be used to create a separate data file as a byproduct of your Maxima work. The function has the syntax **with_stdout (file,expr1,expr2,...)** and writes any output generated with **print**, **display**, or **grind** (for example) to the indicated file, overwriting any pre-existing file, and creating a unix type file.

```
(%i30) with_stdout (mkp("tmp.out"),
           for i thru 10 do
             print (i,",",i^2,",",i^3))$
(%i31) print_file (mkp("tmp.out"))$
1 , 1 , 1
2 , 4 , 8
3 , 9 , 27
4 , 16 , 64
5 , 25 , 125
6 , 36 , 216
7 , 49 , 343
8 , 64 , 512
9 , 81 , 729
10 , 100 , 1000
(%i32) read_data (mkp("tmp.out"));
(%o32) [[1,1,1],[2,4,8],[3,9,27],[4,16,64],[5,25,125],[6,36,216],[7,49,343],
        [8,64,512],[9,81,729],[10,100,1000]]
(%i33) file_info (mkp("tmp.out"));
(%o33) [10,10,unix,134]
```

Notice that if you don't provide the full path to your work directory with the file name (with the Maxima function **with_stdout**), the file will be created in the **../bin/** folder of Maxima.

2.2.17 Creating a Data File from a Nested List Using write_data

The core Maxima function **write_data** can be used to write the contents of a nested list to a named file, writing one line for each sublist, overwriting the contents of any pre-existing file of that name, creating a unix type file with space separator as the default. The simplest syntax, **write_data (list, filename)** produces the default space separation of the sublist items. You can get comma separation or semicolon separation by using respectively **write_data (list,filename,comma)** or **write_data (list,filename,semicolon)**.

Note again that you need to supply the full path as well as the file name, or else the file will be created by **write_data** in **.../bin/**.

(This same function can also be used to write a Maxima matrix object to a data file.)

```
(%i34) dataL : [[0,2],[1,3],[2,4]]$
(%i35) write_data(dataL,mkp("tmp.out"))$
(%i36) file_search("tmp.out");
(%o36) "c:/work2/tmp.out"
(%i37) print_file(mkp("tmp.out"))$
0 2
1 3
2 4
(%i38) file_info(mkp("tmp.out"));
(%o38) [3,3,unix,12]
(%i39) write_data(dataL,mkp("tmp.out"),comma)$
(%i40) print_file(mkp("tmp.out"))$
0,2
1,3
2,4
(%i41) file_info(mkp("tmp.out"));
(%o41) [3,3,unix,12]
(%i42) write_data(dataL,mkp("tmp.out"),semicolon)$
(%i43) print_file(mkp("tmp.out"))$
0;2
1;3
2;4
```

Here is a simple example taken from a question from the Maxima mailing list. Suppose you compute the expression **(1 - z)/(1 + z)** for a number of values of **z**, and you want to place the numbers **z , f(z)** in a data file for later use.

The easiest way to do this in Maxima is to create a list of sublists, with each sublist being a row in your new data file. You can then use the Maxima function **write_data**, which has the syntax: **write_data (datalist, filename)**.

Let's keep the example really simple and just use five integral values of **z**, starting with an empty list we will call **dataL**. (The final **L** is useful (but not necessary) to remind us that it stands for a list.)

```
(%i44) dataL : []$
(%i45) for x thru 5 do (
          px : subst (x,z, (1-z)/(1+z)),
          dataL : cons ( [x, px], dataL ))$
(%i46) dataL;
(%o46) [[5,-2/3],[4,-3/5],[3,-1/2],[2,-1/3],[1,0]]
(%i47) dataL : reverse (dataL);
(%o47) [[1,0],[2,-1/3],[3,-1/2],[4,-3/5],[5,-2/3]]
```

```
(%i48) write_data (dataL, mkp("mydata1.dat"))$
(%i49) print_file (mkp("mydata1.dat"))$
1 0
2 -1/3
3 -1/2
4 -3/5
5 -2/3
(%i50) read_data (mkp("mydata1.dat"));
(%o50) [[1,0],[2,-1/3],[3,-1/2],[4,-3/5],[5,-2/3]]
(%i51) file_info (mkp("mydata1.dat"));
(%o51) [5,5,unix,32]
```

If you open the unix text file **mydata1.dat** using the older version of the Windows text file application Notepad, you may only see one line, which looks like:

```
1 02 -1/33 -1/24 -3/55 -2/3
```

which occurs because Maxima creates text files having Unix style line endings which older native Windows applications don't recognise.

In order to see the two columns of numbers (using a text editor), you should use the freely available Windows text editor Notepad2. (Just Google it.) Notepad2 recognises unix, mac and windows line ending control characters, and in fact has a signal (LF for unix) at the bottom of the screen which tells you what the line ending control characters are.

The alternative choice provided by the standard Maxima system to created a nested list from a data file is the function **read_nested_list**. This is not as good a choice as our package function **read_data**, as is shown here:

```
(%i52) read_nested_list (mkp("mydata1.dat"));
(%o52) [[1,0],[2,-1,\/,3],[3,-1,\/,2],[4,-3,\/,5],[5,-2,\/,3]]
(%i53) display2d:true$
(%i54) read_nested_list (mkp("mydata1.dat"));
(%o54) [[1, 0], [2, - 1, /, 3], [3, - 1, /, 2], [4, - 3, /, 5],
                                                        [5, - 2, /, 3]]
```

2.3 Least Squares Fit to Experimental Data

2.3.1 Maxima and Least Squares Fits: lsquares_estimates

Suppose we are given a list of **[x,y]** pairs which are thought to be roughly described by the relation **y = a*x^b + c**, where the three parameters are all of order **1**. We can use the data of **[x,y]** pairs to find the "best" values of the unknown parameters **[a, b, c]**, such that the data is described by the equation **y = a*x^b + c** (a three parameter fit to the data).

We are using one of the Manual examples for **lsquares_estimates**.

```
(%i1) dataL : [[1, 1], [2, 7/4], [3, 11/4], [4, 13/4]]$
(%i2) display2d:false$
(%i3) dataM : apply ('matrix, dataL);
(%o3) matrix([1,1],[2,7/4],[3,11/4],[4,13/4])
(%i4) load (lsquares);
(%o4) "C:/PROGRA~1/MA81DB~1.0/share/maxima/5.28.0-2/share/lsquares/lsquares.mac"
(%i5) fpprintprec:6$
(%i6) lsquares_estimates (dataM, [x,y], y=a*x^b+c,
               [a,b,c], initial=[3,3,3], iprint=[-1,0] );
(%o6) [[a = 1.37575,b = 0.7149,c = -0.4021]]
(%i7) myfit : a*x^b + c , % ;
(%o7) 1.37575*x^0.7149-0.4021
```

Note that we must use **load (lsquares);** to use this method. We can now make a plot of both the discrete data points and the least squares fit to those four data points.

```
(%i8) plot2d ([myfit,[discrete,dataL]],[x,0,5],
        [style,[lines,5],[points,4,2,1]],
         [legend,"myfit", "data"],
         [gnuplot_preamble,"set key bottom;"])$
```

which produces the plot

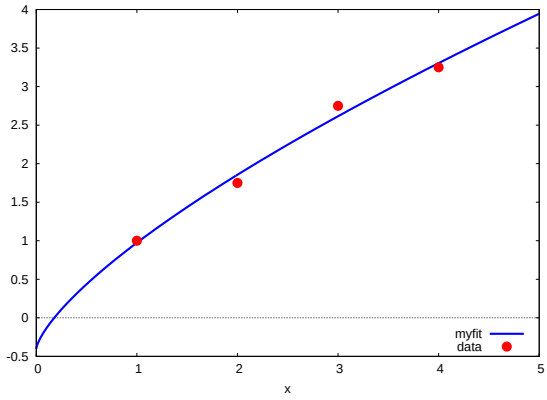

Figure 18: Three Parameter Fit to Four Data Points

2.3.2 Syntax of lsquares_estimates

The minimal syntax is

```
lsquares_estimates (data-matrix, data-variable-list, fit-eqn, param-list );
```

in which the **data-variable-list** assigns a variable name to the corresponding column of the **data-matrix**, and the **fit-eqn** is an equation which is a relation among the data variable symbols and the equation parameters which appear in **param-list**. The function returns the "best fit" values of the equation parameters in the form **[[p1 = p1val, p2 = p2val, ...]]**.

In the example above, the data variable list was **[x, y]** and the parameter list was **[a, b, c]**.

If an exact solution cannot be found, a numerical approximation is attempted using **lbfgs**, in which case, all the elements of the data matrix should be "numbers" **numberp(x) -> true**. This means that **%pi** and **%e**, for example, should be converted to explicit numbers before use of this method.

```
(%i1) expr : 2*%pi + 3*exp(-4);
                                          - 4
(%o1)                          2 %pi + 3 %e
(%i2) listconstvars:true$
(%i3) listofvars(expr);
                                [%e, %pi]
(%i4) map('numberp,%);
                               [false, false]
(%i5) fullmap('numberp,expr);
                              true
(%o5)                  false     true + false true
(%i6) float(expr);
                              6.338132223845789
(%i7) numberp(%);
(%o7)                              true
```

Optional arguments to **lsquares_estimates** are (in any order)

```
initial = [p10, p20,...], iprint = [n, m], tol = search-tolerance
```

The list **[p10, p20, ...]** is the optional list of initial values of the equation parameters, and without including your own guess for starting values this list defaults (in the code) to **[1, 1, ...]**.

The first integer **n** in the **iprint** list controls how often progress messages are printed to the screen. The default is **n = 1** which causes a new progress message printout each iteration of the search method. Using **n = -1** surpresses all progress messages. Using **n = 5** allows one progress message every five iterations.

The second integer **m** in the **iprint** list controls the verbosity, with **m = 0** giving minimum information and **m = 3** giving maximum information.

The option **iprint = [-1,0]** will hide the details of the search process.

The default value of the **search-tolerance** is **1e-3**, so by using the option **tol = 1e-8** you might find a more accurate solution.

Many examples of the use of the **lsquares** package are found in the file **lsquares.mac**, which is found in the **...share/contrib** folder. You can also see great examples of efficient programming in the Maxima language in that file.

2.3.3 Coffee Cooling Model

"Newton's law of cooling" (only approximate and not a law) assumes the rate of decrease of temperature (celsius degrees per minute) is proportional to the instantaneous difference between the temperature $\mathbf{T(t)}$ of the coffee in the cup and the surrounding ambient temperature $\mathbf{T_s}$, the latter being treated as a constant. If we use the symbol $\mathbf{r}$ for the "rate constant" of proportionality, we then assume the cooling of the coffee obeys the first order differential equation

$$\frac{\mathbf{d\,T}}{\mathbf{d\,t}} = -\mathbf{r}\,(\mathbf{T(t)} - \mathbf{T_s}) \tag{2.1}$$

Since $\mathbf{T}$ has dimension degrees Celsius, and $\mathbf{t}$ has dimension minute, the dimension of the rate constant $\mathbf{r}$ must be **1/min**.

(This attempt to employ a rough mathematical model which can be used for the cooling of a cup of coffee avoids a bottom-up approach to the problem, which would require mathematical descriptions of the four distinct physical mechanisms which contribute to the decrease of thermal energy in the system hot coffee plus cup to the surroundings: thermal radiation (net electromagnetic radiation flux, approximately black body) energy transport across the surface of the liquid and cup, collisional heat conduction due to the presence of the surrounding air molecules, convective energy transport due to local air temperature rise, and finally evaporation which is the escape of the fastest coffee molecules which are able to escape the binding surface forces at the liquid surface. If the temperature difference between the coffee and the ambient surroundings is not too large, experiment shows that the simple relation above is roughly true.)

This differential equation is easy to solve "by hand" , since we can write

$$\frac{\mathbf{d\,T}}{\mathbf{d\,t}} = \frac{\mathbf{d\,(T - T_s)}}{\mathbf{d\,t}} = \frac{\mathbf{d\,y}}{\mathbf{d\,t}} \tag{2.2}$$

and then divide both sides by $\mathbf{y = (T - T_s)}$, multiply both sides by $\mathbf{d\,t}$, and use $\mathbf{d\,y/y = d\,ln(y)}$ and finally integrate both sides over corresponding intervals to get $\mathbf{ln(y) - ln(y_0) = ln(y/y_0) = -r\,t}$, where $\mathbf{y_0 = T(0) - T_s}$ involves the initial temperature at $\mathbf{t = 0}$. Since

$$\mathbf{e^{ln(A)} = A}, \tag{2.3}$$

by equating the exponential of the left side to that of the right side, we get

$$\mathbf{T(t) = T_s + (T(0) - T_s)\, e^{-rt}}. \tag{2.4}$$

Using **ode2**, **ic1**, **expand**, and **collectterms**, we can also use Maxima just for fun:

```
(%i1) de : 'diff(T,t) + r*(T - Ts);
                                dT
(%o1)                           -- + r (T - Ts)
                                dt
(%i2) gsoln : ode2(de,T,t);
                                - r t    r t
(%o2)                    T = %e      (%e    Ts + %c)
(%i3) de, gsoln, diff, ratsimp;
(%o3)                                0
(%i4) ic1 (gsoln, t = 0, T = T0);
                              - r t          r t
(%o4)                   T = %e      (T0 + (%e    - 1) Ts)
(%i5) expand (%);
                              - r t        - r t
(%o5)                   T = %e      T0 - %e      Ts + Ts
(%i6) Tcup : collectterms ( rhs(%), exp(-r*t) );
                               - r t
(%o6)                         %e      (T0 - Ts) + Ts
(%i7) Tcup, t = 0;
(%o7)                                  T0
```

We arrive at the same solution as found "by hand". We have checked the particular solution for the initial condition and checked that our original differential equation is satisfied by the general solution.

2.3.4 Experiment Data for Coffee Cooling

Let's take some "real world" data for this problem (p. 21, An Introduction to Computer Simulation Methods, 2nd ed., Harvey Gould and Jan Tobochnik, Addison-Wesley, 1996) which is in a data file **`c:\work2\coffee.dat`** on the author's Window's XP computer (data file available with this chapter on the author's webpage).

This file contains three columns of tab separated numbers, column one being the elapsed time in minutes, column two is the Celsius temperature of the system glass plus coffee for black coffee, and column three is the Celsius temperature for the system glass plus creamed coffee. The glass-coffee temperature was recorded with an estimated accuracy of $\mathbf{0.1^\circ C}$. The ambient temperature of the surroundings was $\mathbf{17\,^\circ C}$. The function **`read_data`** automatically replaces tabs (ascii(9)) in the data by spaces (ascii(32)) as each line is read in.

We need to remind the reader that we are using a function **`mkp`** to create a complete path to a file name. This function was discussed at the beginning of the section on file manipulation methods. For convenience, we repeat some of that discussion here:

To ease the pain of typing the full path, you can use a function which I call **`mkp`** ("make path") which I define in my **`maxima.init`** file, whose contents are:

```
/* this is c:\Documents and Settings\Edwin Woollett\maxima\maxima-init.mac  */
maxima_userdir: "c:/work2" $
maxima_tempdir : "c:/work2"$
file_search_maxima : append(["c:/work2/###.{mac,mc}"],file_search_maxima )$
file_search_lisp : append(["c:/work2/###.lisp"],file_search_lisp )$

bpath : "c:/work2/"$
mkp (_fname) := sconcat (bpath,_fname)$
```

We will use this function, **mkp**, below for example with **print_file** and **read_data**. The string processing function **sconcat** creates a new string from two given strings ("string concatenation"):

```
(%i8) display2d:false$
(%i9) sconcat("a bc","xy z");
(%o9) "a bcxy z"
```

Note that we used a global variable **bpath** ("base of path" or "beginning of path") instead of the global variable **maxima_userdir**. This makes it more convenient for the user to redefine **bpath** "on the fly" instead of opening and editing **maxima-init.mac**, and restarting Maxima to have the changes take effect.

```
(%i10) file_search("coffee.dat");
(%o10)                          c:/work2/coffee.dat
(%i11) (display2d:false,load(mfiles));
(%o11) "c:/work2/mfiles.mac"
(%i12) print_file(mkp("coffee.dat"))$
0         82.3         68.8
2         78.5         64.8
4         74.3         62.1
6         70.7         59.9
8         67.6         57.7
10         65.0         55.9
12         62.5         53.9
14         60.1         52.3
16         58.1         50.8
18         56.1         49.5
20         54.3         48.1
22         52.8         46.8
24         51.2         45.9
26         49.9         44.8
28         48.6         43.7
30         47.2         42.6
32         46.1         41.7
34         45.0         40.8
36         43.9         39.9
38         43.0         39.3
40         41.9         38.6
42         41.0         37.7
44         40.1         37.0
```

We now use **read_data** which will create a list of sublists, one sublist per row.

```
(%i13) fpprintprec:6$
(%i14) cdata : read_data(mkp("coffee.dat"));
(%o14) [[0,82.3,68.8],[2,78.5,64.8],[4,74.3,62.1],[6,70.7,59.9],[8,67.6,57.7],
       [10,65.0,55.9],[12,62.5,53.9],[14,60.1,52.3],[16,58.1,50.8],
       [18,56.1,49.5],[20,54.3,48.1],[22,52.8,46.8],[24,51.2,45.9],
       [26,49.9,44.8],[28,48.6,43.7],[30,47.2,42.6],[32,46.1,41.7],
       [34,45.0,40.8],[36,43.9,39.9],[38,43.0,39.3],[40,41.9,38.6],
       [42,41.0,37.7],[44,40.1,37.0]]
```

We now use **makelist** to create a (time, temperature) list based on the black coffee data and then based on the white (creamed coffee) data.

```
(%i15) black_data : makelist( [first(cdata[i]),second(cdata[i])],
                        i,1,length(cdata));
(%o15) [[0,82.3],[2,78.5],[4,74.3],[6,70.7],[8,67.6],[10,65.0],[12,62.5],
       [14,60.1],[16,58.1],[18,56.1],[20,54.3],[22,52.8],[24,51.2],[26,49.9],
       [28,48.6],[30,47.2],[32,46.1],[34,45.0],[36,43.9],[38,43.0],[40,41.9],
       [42,41.0],[44,40.1]]
```

```
(%i16) white_data : makelist( [first(cdata[i]),third(cdata[i])],
                        i,1,length(cdata));
(%o16) [[0,68.8],[2,64.8],[4,62.1],[6,59.9],[8,57.7],[10,55.9],[12,53.9],
       [14,52.3],[16,50.8],[18,49.5],[20,48.1],[22,46.8],[24,45.9],[26,44.8],
       [28,43.7],[30,42.6],[32,41.7],[34,40.8],[36,39.9],[38,39.3],[40,38.6],
       [42,37.7],[44,37.0]]
```

2.3.5 Least Squares Fit of Coffee Cooling Data

We now use **lsquares_estimates** to use a least squares fit with each of our data sets to our phenomenological model, that is finding the "best" value of the cooling rate constant **r** that appears in Eq. (2.4). The function **lsquares_estimates(data_matrix, eqnvarlist,eqn,paramlist)** is available after using **load(lsquares)**.

To save space in this chapter we use **display2d:false** to surpress the default two dimensional display of a Maxima **matrix** object.

```
(%i17) black_matrix : apply ( 'matrix, black_data);
(%o17) matrix([0,82.3],[2,78.5],[4,74.3],[6,70.7],[8,67.6],[10,65.0],[12,62.5],
              [14,60.1],[16,58.1],[18,56.1],[20,54.3],[22,52.8],[24,51.2],
              [26,49.9],[28,48.6],[30,47.2],[32,46.1],[34,45.0],[36,43.9],
              [38,43.0],[40,41.9],[42,41.0],[44,40.1])
(%i18) white_matrix : apply ( 'matrix, white_data);
(%o18) matrix([0,68.8],[2,64.8],[4,62.1],[6,59.9],[8,57.7],[10,55.9],
               [12,53.9],[14,52.3],[16,50.8],[18,49.5],[20,48.1],[22,46.8],
               [24,45.9],[26,44.8],[28,43.7],[30,42.6],[32,41.7],[34,40.8],
               [36,39.9],[38,39.3],[40,38.6],[42,37.7],[44,37.0])
```

We now load **lsquares.mac** and calculate the "best fit" values of the cooling rate constant **r** for both cases. For the black coffee case, **T0 = 82.3 deg C** and **Ts = 17 deg C** and we surpress the units.

```
(%i19) load(lsquares);
(%o19) "C:/PROGRA~1/MAXIMA~1.1-G/share/maxima/5.25.1/share/contrib/lsquares.mac"
(%i20) black_eqn : T = 17 + 65.3*exp(-r*t);
(%o20) T = 65.3*%e^-(r*t)+17
(%i21) lsquares_estimates ( black_matrix, [t,T], black_eqn, [r],
                            iprint = [-1,0] );
(%o21) [[r = 0.02612]]
(%i22) black_fit : rhs ( black_eqn ), %;
(%o22) 65.3*%e^-(0.02612*t)+17
```

Thus **rblack** is roughly **0.026 min^(-1)**.

For the white coffee case, **T0 = 68.8 deg C** and **Ts = 17 deg C**.

```
(%i23) white_eqn : T = 17 + 51.8*exp(-r*t);
(%o23) T = 51.8*%e^-(r*t)+17
(%i24) lsquares_estimates ( white_matrix, [t,T], white_eqn, [r],
                           iprint = [-1,0] );
(%o24) [[r = 0.02388]]
(%i25) white_fit : rhs ( white_eqn ), %;
(%o25) 51.8*%e^-(0.02388*t)+17
```

Thus **rwhite** is roughly **0.024 min^(-1)**, a slightly smaller value than for the black coffee (which is reasonable since a black body is a better radiator of thermal energy than a white surface).

A prudent check on mathematical reasonableness can be made by using, say, the two data points for $\mathbf{t = 0}$ and $\mathbf{t = 24\,min}$ to solve for a rough value of **r**. For this rough check, the author concludes that **rblack** is roughly `0.027 min^(-1)` and **rwhite** is roughly `0.024 min^(-1)`.

We can now plot the temperature data against the best fit model curve, first for the black coffee case.

```
(%i26) plot2d( [ black_fit ,[discrete,black_data] ],
           [t,0,50], [style, [lines,5], [points,2,2,6]],
           [ylabel," "] ,
           [xlabel," Black Coffee T(deg C) vs. t(min) with r = 0.026/min"],
           [legend,"black fit","black data"] )$
```

which produces the plot

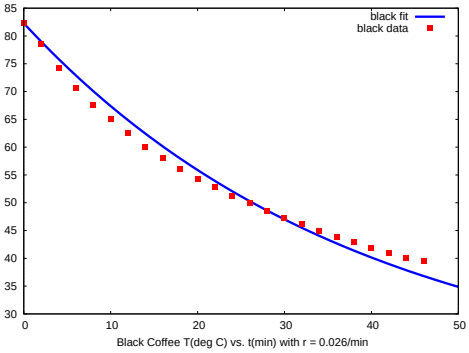

Figure 19: Black Coffee Data and Fit

and next plot the white coffee data and fit:

```
(%i27) plot2d( [ white_fit ,[discrete, white_data] ],
           [t,0,50], [style, [lines,5], [points,2,2,6]],
           [ylabel," "] ,
           [xlabel," White Coffee T(deg C) vs. t(min) with r = 0.024/min"],
           [legend,"white fit","white data"] )$
```

which yields the plot

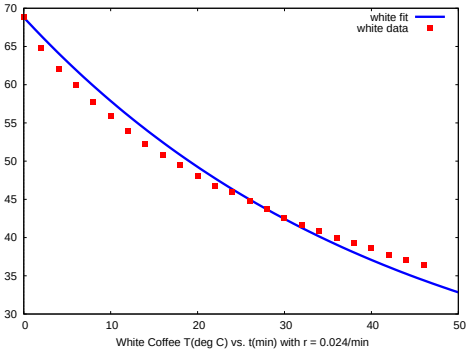

Figure 20: White Coffee Data and Fit

Cream at Start or Later?

Let's use the above approximate values for the cooling rate constants to find the fastest method to use to get the temperature of hot coffee down to a drinkable temperature. Let's assume we start with a glass of very hot coffee, $\mathbf{T_0 = 90°C}$, and want to compare two methods of getting the temperature down to $\mathbf{75°C}$, which we assume is low enough to start sipping. We will assume that adding cream lowers the temperature of the coffee by $\mathbf{5°C}$ for both options we explore. Option 1 (white option) is to immediately add cream and let the creamed coffee cool down from $\mathbf{85°C}$ to $\mathbf{75°C}$. We first write down a general expression as a function of **T0** and **r**, and then substitute values appropriate to the white coffee cooldown.

```
(%i28) T : 17 + (T0 -17)*exp(-r*t);
(%o28) %e^-(r*t)*(T0-17)+17
(%i29) T1 : T, [T0 = 85, r = 0.02388];
(%o29) 68*%e^-(0.02388*t)+17
(%i30) t1 : find_root(T1 - 75,t,2,10);
(%o30) 6.661
```

The "white option" requires about $\mathbf{6.7\,min}$ for the coffee to be sippable.

Option 2 (the black option) lets the black coffee cool from $\mathbf{90°C}$ to $\mathbf{80°C}$, and then adds cream, immediately getting the temperature down from $\mathbf{80°C}$ to $\mathbf{75°C}$

```
(%i31) T2 : T, [T0 = 90, r = 0.02612];
(%o31) 73*%e^-(0.02612*t)+17
(%i32) t2 : find_root(T2 - 80,t,2,10);
(%o32) 5.6403
```

The black option (option 2) is the fastest method to cool the coffee, taking about $\mathbf{5.64\,min}$ which is about $\mathbf{61\,sec}$ less than the white option 1.