

# Maxima by Example: Ch.5: 2D Plots and Graphics using qdraw \*

Edwin L. Woollett

January 29, 2009

## Contents

<b>5</b>	<b>2D Plots and Graphics using qdraw</b>	<b>3</b>
5.1	Quick Plots for Explicit Functions: ex(...) . . . . .	3
5.2	Quick Plots for Implicit Functions: imp(...) . . . . .	10
5.3	Contour Plots with contour(...) . . . . .	12
5.4	Density Plots with qdensity(...) . . . . .	14
5.5	Explicit Plots with Greater Control: ex1(...) . . . . .	17
5.6	Explicit Plots with ex1(...) and Log Scaled Axes . . . . .	19
5.7	Data Plots with Error Bars: pts(...) and errorbars(...) . . . . .	21
5.8	Implicit Plots with Greater Control: imp1(...) . . . . .	27
5.9	Parametric Plots with para(...) . . . . .	29
5.10	Polar Plots with polar(...) . . . . .	31
5.11	Geometric Figures: line(...) . . . . .	32
5.12	Geometric Figures: rect(...) . . . . .	34
5.13	Geometric Figures: poly(...) . . . . .	35
5.14	Geometric Figures: circle(...) and ellipse(...) . . . . .	38
5.15	Geometric Figures: vector(..) . . . . .	40
5.16	Geometric Figures: arrowhead(..) . . . . .	43
5.17	Labels with Greek Letters . . . . .	43
5.17.1	Enhanced Postscript Methods . . . . .	43
5.17.2	Windows Fonts Methods with jpeg Files . . . . .	47
5.17.3	Using Windows Fonts with the Gnuplot Console Window . . . . .	48
5.18	Even More with more(...) . . . . .	49
5.19	Programming Homework Exercises . . . . .	50
5.19.1	General Comments . . . . .	50
5.19.2	XMaxima Tips . . . . .	51
5.19.3	Suggested Projects . . . . .	51
5.20	Acknowledgements . . . . .	52

---

\*This version uses Maxima 5.17.1. This is a live document. Check <http://www.csulb.edu/~woollett/> for the latest version of these notes. Send comments and suggestions to [woollett@charter.net](mailto: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.

## 5 2D Plots and Graphics using qdraw

### 5.1 Quick Plots for Explicit Functions: ex(...)

This chapter provides an introduction to a new graphics interface developed by the author of the Maxima by Example tutorial notes. The **qdraw** package ( qdraw.mac: available for download on the Maxima by Example webpage ) is an interface to the **draw** package function **draw2d**; to obtain a plot you must load **draw** as well as **qdraw**. You can just use `load(qdraw)` if you have the file in your work folder and have set up your file search as described in Chap. 1. Otherwise just put qdraw.mac into your `...maxima...share\draw` folder where it will be found.

The primary motivation for the **qdraw** package is to provide "quick" (hence the "q" in "qdraw") plotting software which provides the kinds of plotting defaults which are of interest to students and researchers in the physical sciences and engineering. There are two "quick" plotting functions you can use with **qdraw**: **ex(...)** and **imp(...)**.

An entry like

```
(%i1) load(draw)$  
(%i2) load(qdraw)$  
(%i3) qdraw( ex( [x,x^2,x^3],x,-3,3 ) )$
```

will produce a plot of the three explicit functions of  $x$  in the first argument list, using line width = 3, an automatic rotating series of default colors, clearly visible  $x$  and  $y$  axes, and also a "grid" as well. The **ex** function passes its arguments on to a series of calls to **draw2d**'s **explicit** function.

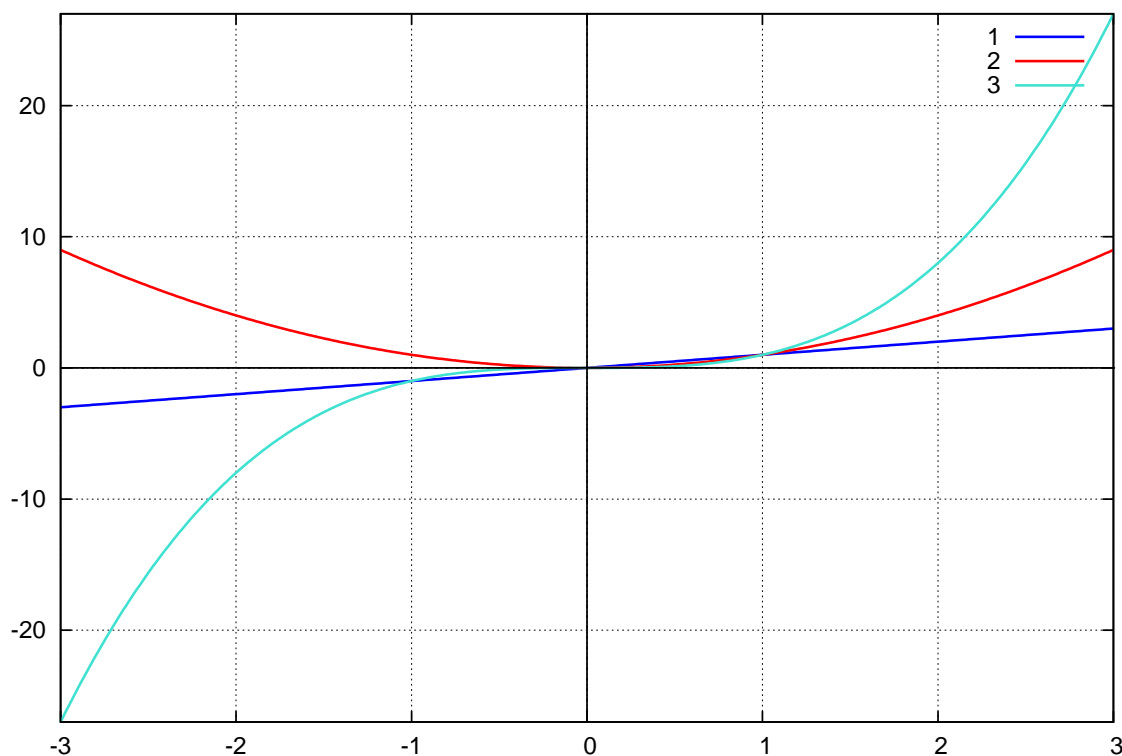


Figure 1: Using `ex()` for  $x, x^2, x^3$

Since  $3^3 = 27$ , **draw2d** extends the vertical axis to  $\pm 27$  by default.

You can control the vertical range of the "canvas" with the **yr(...)** function, which passes its arguments to a **draw2d** entry `yrange = [y1,y2]`.

```
(%i4) qdraw( ex([x,x^2,x^3],x,-3,3 ),yr(-2,2) )$
```

which produces the plot:

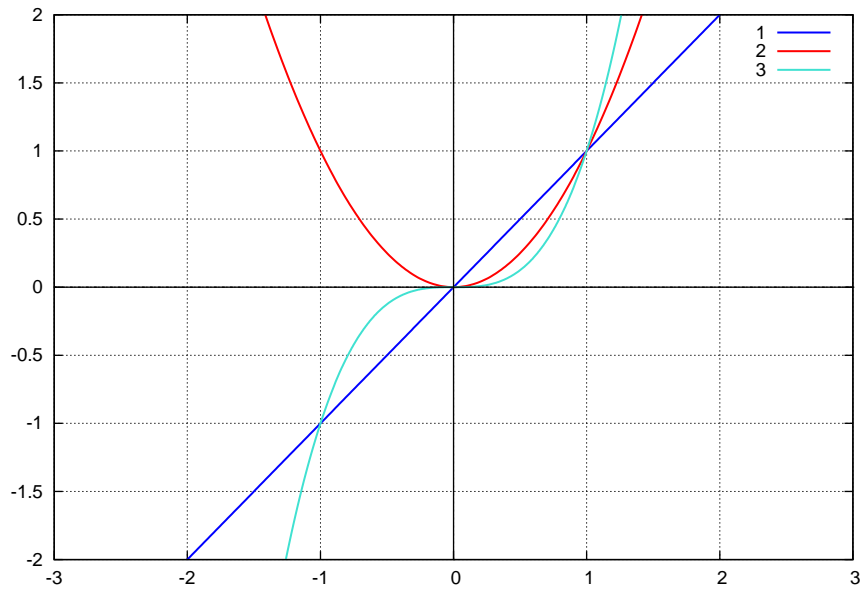


Figure 2: Adding  $yr(-2,2)$

You can make the lines thinner or thicker than the default line width (3) by using the **lw(n)** option, which only affects the quick plotting functions **ex(...)** and **imp(...)**, as in

```
(%i5) qdraw(ex([x,x^2,x^3],x,-3,3 ),yr(-2,2) , lw(6) )$
```

to get:

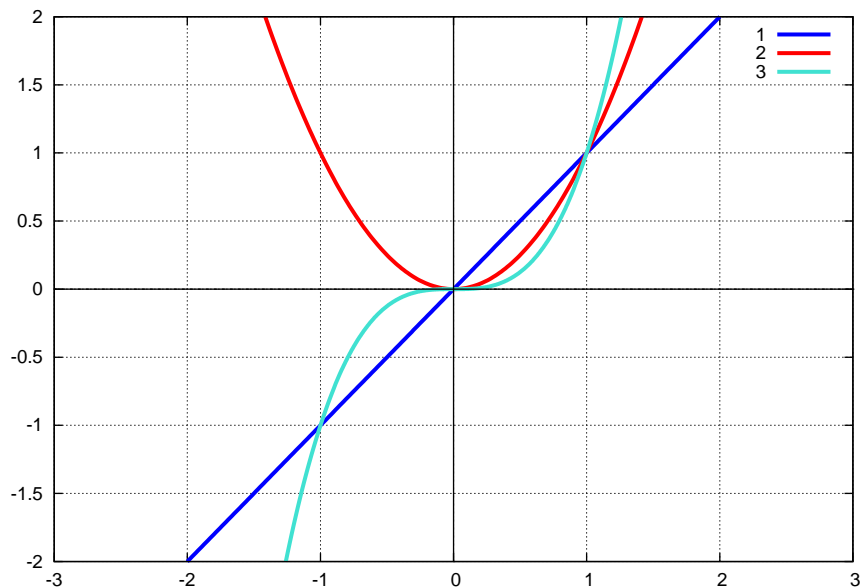


Figure 3: Adding  $lw(6)$

You can place the plot "key" (legend) at the bottom right by using the **key(bottom)** option, as in:

```
(%i6) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),lw(6),key(bottom) )$
```

to get:

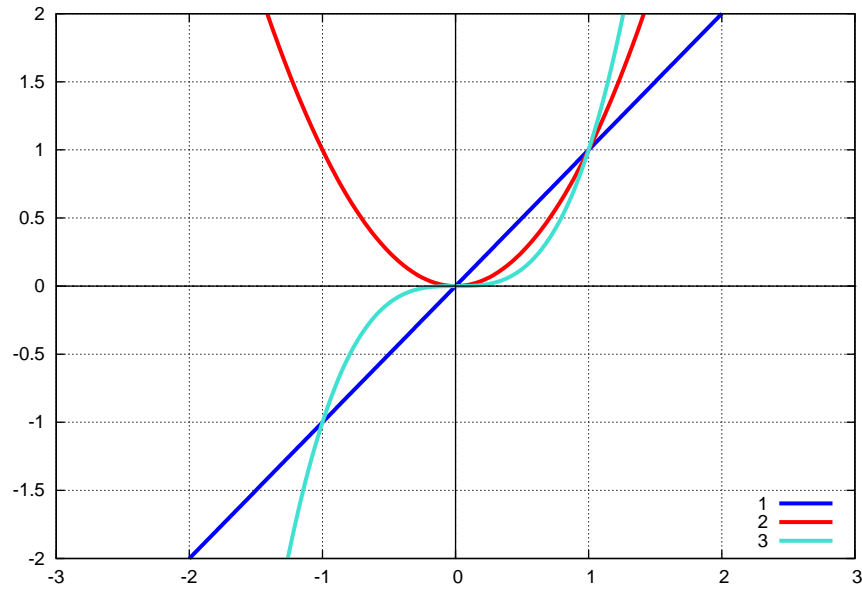


Figure 4: Adding *key(bottom)*

You can remove the default grid and xy axes by adding **cut(grid,xyaxes)** as in:

```
(%i7) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),  
lw(6),key(bottom), cut(grid, xyaxes) )$
```

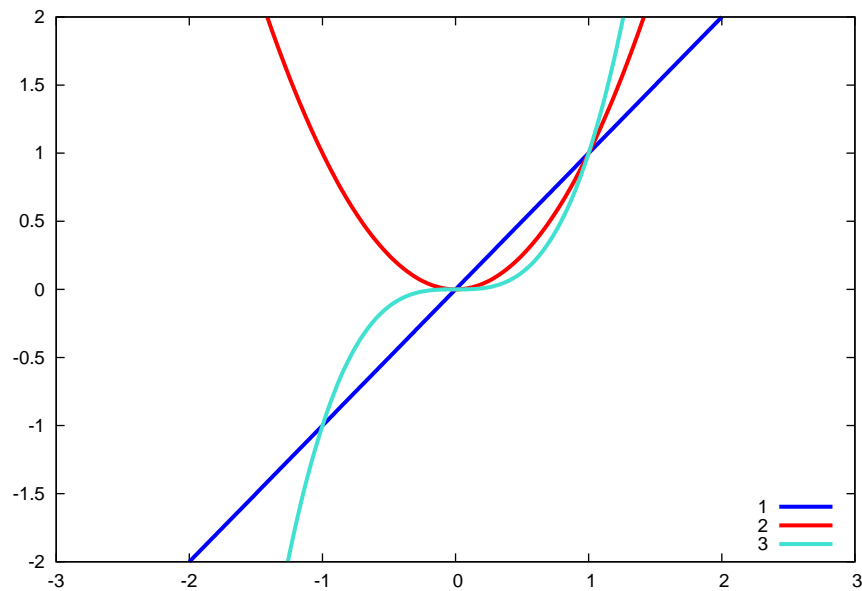


Figure 5: Adding *cut(grid,xyaxes)*

You can remove the grid, axes, the key, and all the borders using **cut(all)**, as in:

```
(%i8) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2),
             lw(6), cut( all ) )$
```

which results in a "clean" canvas:

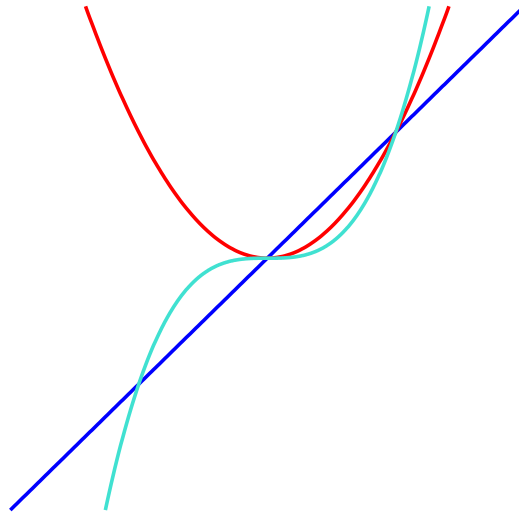


Figure 6: Adding *cut(all)*

Restoring the (default) grid and axes, we can place points (default size 3 and color black) at the intersection points using the **pts(...)** option, which passes a points list to **draw2d**'s **points** function:

```
(%i9) qdraw( ex([x,x^2,x^3],x,-3,3 ),yr(-2,2),lw(6),
             key(bottom),pts( [ [-1,-1],[0,0],[1,1] ] ) )$
```

which produces:

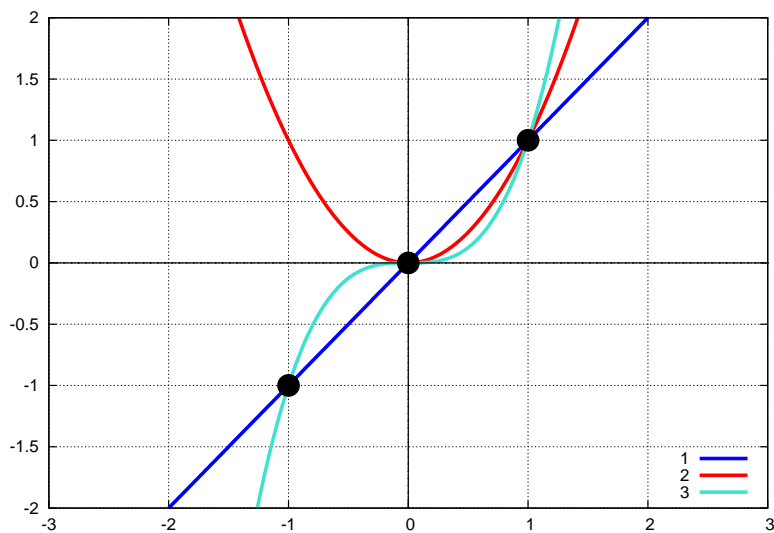


Figure 7: Adding *pts(ptlist)*

We can override the default size and color of those points by including inside the **pts** function the optional **ps(n)** and **pc(c)** arguments, as in:

```
(%i10) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
               pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta) ) )$
```

which produces:

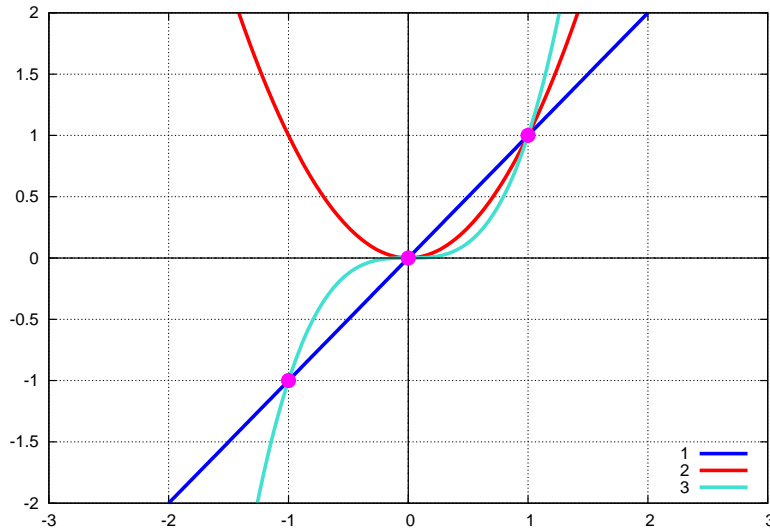


Figure 8: Adding  $pts(ptlist, ps(2), pc(magenta))$

We can include a key entry for the points using the **pk(string)** option for the **pts** function, as in:

```
(%i11) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
               pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta),pk("intersections") ) )$
```

which produces:

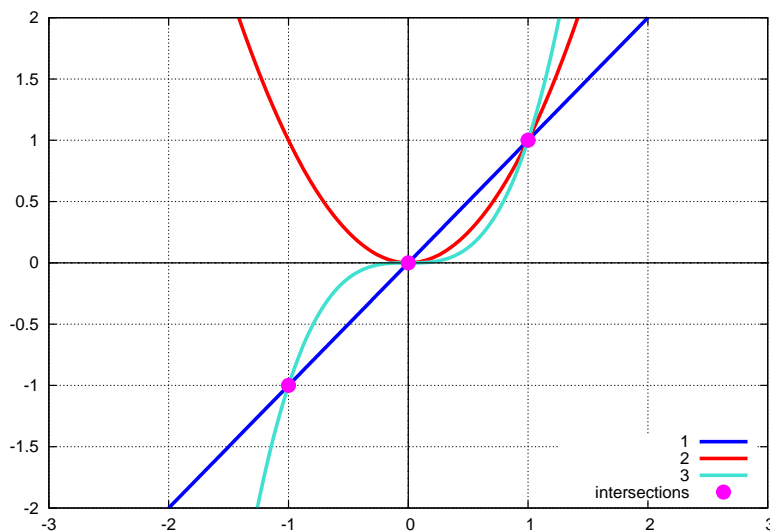


Figure 9:  $pts(ptlist, ps(2), pc(magenta), pk("intersections"))$

The "eps" file ch5p9.eps used to get the last figure in the Tex file which is the source of this pdf file was produced using the **pic**(type, filename) option to **qdraw**, as in:

```
(%i12) qdraw( ex( [x,x^2,x^3],x,-3,3 ),yr(-2,2), lw(6),key(bottom),
             line(-3,0,3,0,lw(2)),line(0,-2,0,2,lw(2)),
             pts([ [-1,-1],[0,0],[1,1] ],ps(2),pc(magenta),pk("intersections")),
             pic(eps,"ch5p9") )$
```

We have discussed, at the end of Chapter 1, Getting Started, how we insert such an "eps" file into our Tex file in order to get the figures you see here.

The extra, optional, arguments we have included inside **qdraw** can be entered in any order; in fact, all arguments to **qdraw** are optional and can be entered in any order. For example

```
(%i13) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
             key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
             pts([ [-1,-1],[0,0],[1,1] ]) )$
```

which adds  $\sin(3x)e^{x/3}$  with a separate **ex(...)** argument to **qdraw**, and produces

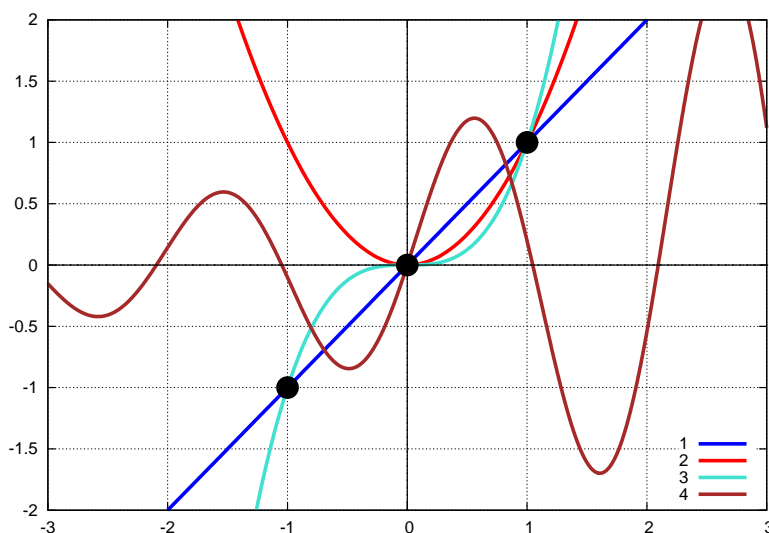


Figure 10: Adding  $\sin(3x)e^{x/3}$

We next add a label "qdraw at work" to our plot.

Using Windows, the font size must be adjusted only after getting the plot drawn in a Gnuplot window by right clicking the icon in the upper left hand corner, and selecting Options, Choose Font,... If you increase the windows graphics font from the default value of 10 to 20, say, you will see a dramatic increase in the size of the label, but also in the size of the x and y axis coordinate numbers, and also a large increase in size of any features of the graphics which used a call to draw2d's **points** function (such as **qdraw**'s **pts** function).

This behavior seems to be related to the limitations of the present incarnation of the adaptation of Gnuplot to the Windows system, and hopefully will be addressed in the future by the volunteers who work on Gnuplot software.

Our illustration of the use of labels will simply be what one gets by sending the graphics object to a graphics file "ch5p11.eps" and including that file in our Tex/pdf file. In Maxima, we use the code:

```
(%i14) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
             key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
             pts([ [-1,-1],[0,0],[1,1] ]) ,
             label(["qdraw at work",-2.9,1.5]),
             pic(eps,"ch5p11",font("Times-Bold",20) ) );
```



The resulting plot is then

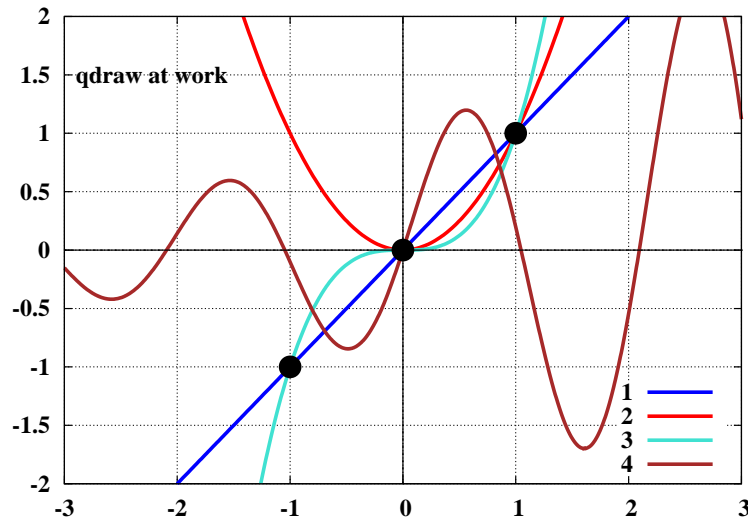


Figure 11: Adding a label at (-2.9,1.5)

If you look in the html Maxima manual under the index item "font", which takes you to a subsection of the draw package documentation, you will find a listing of the available postscript fonts, which one can use with the **pic(eps, filename, font( name, size ) )** function call. These options have the names Times-Roman, Times-Bold, Helvetica, Helvetica-Bold, Courier, Courier-Bold, also -Italic options.

If you want to save the graphics as a jpeg file, the font name should be a string containing the path to the desired font file. Using the Windows XP operating system, the available windows fonts are in the folder `c:\windows\fonts\`. Here is Maxima code to get a jpeg graphics file based on our present drawing:

```
(%i15) qdraw( yr(-2,2),lw(6), ex( [x,x^2,x^3],x,-3,3 ),
             key(bottom), ex(sin(3*x)*exp(x/3),x,-3,3),
             pts([ [-1,-1],[0,0],[1,1] ]),
             label(["qdraw at work",-2.9,1.5]),
             pic(jpg,"ch5p11",font("c:/windows/fonts/timesbd.ttf",20) ) );
```

The resulting jpeg file has thicker lines and bolder labels, so some experimentation may be called for to get the desired result. The font file requested corresponds to times roman bold. The font file extension "tff" stands for "true type fonts". If you look in the windows, fonts folder you can find other interesting choices.

## 5.2 Quick Plots for Implicit Functions: `imp(...)`

The quick plotting function **imp(...)** has the syntax

```
imp( eqnlist, x, x1, x2, y, y1, y2 )  
or   imp( eqn,      x, x1, x2, y, y1, y2 ) .
```

If the equation(s) are actually functions of (u,v) then  $x \rightarrow u$  and  $y \rightarrow v$ . The numbers (x1,x2) determine the horizontal canvas extent, and the numbers (y1,y2) determine the vertical canvas extent. Here is an example using the single equation form:

```
(%i16) qdraw( imp( sin(2*x)*cos(y)=0.4, x,-3,3, y,-3,3 ) ,  
              cut(key) );
```

which produces the "implicit plot":

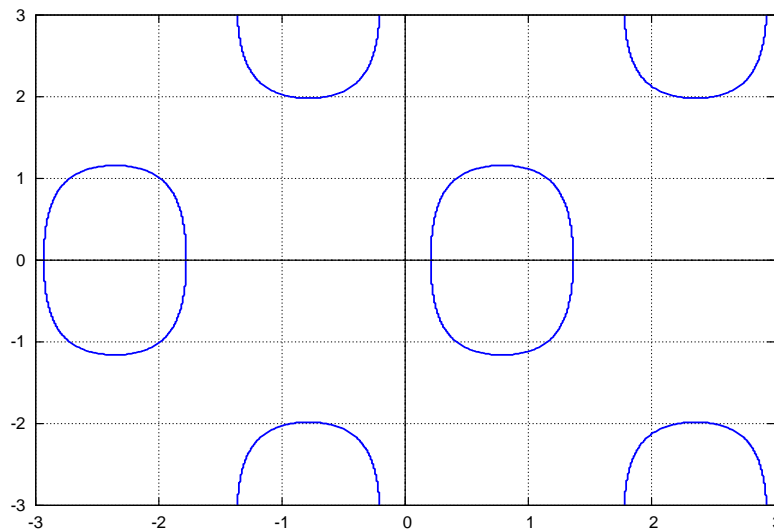


Figure 12: Implicit plot of  $\sin(2x)\cos(y)$

which uses the default line width = 3, the first of the default rotating colors (blue), and, of course, the default axes and grid. To remove the default key, we have used the **cut** function. Since the left hand side of this equation will periodically return to the same numerical value in both the x and the y directions, there is no "limit" to the solutions obtained by setting the left hand side equal to some numerical value between zero and one.

This looks like one piece of a contour plot for the given function. We can add more contour lines using the **imp** function by using the `list_of_equations` form:

```
(%i17) qdraw( imp( [sin(2*x)*cos(y)=0.4,  
                  sin(2*x)*cos(y)=0.7,  
                  sin(2*x)*cos(y)=0.9] , x,-3,3, y,-3,3 ) ,  
              cut(key) );
```

The resulting plot with the default rotating color set is shown on the top of the next page.

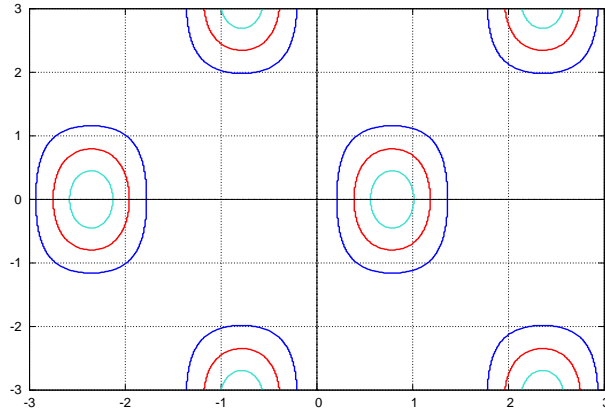


Figure 13: contour plot of  $\sin(2x)\cos(y)$  using `imp()`

Of course if we define `g`, say, to be the expression  $\sin(2x)\cos(y)$  first, we can use that binding to simplify our call to **`imp(...)`** :

```
(%i18) g : sin(2*x)*cos(y)$
(%i19) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-3,3,y,-3,3 ) ,
              cut(key) );
```

to achieve the same plot.

We can also use symbols like `%pi`, which will evaluate to a real number, in our horizontal and vertical limit slots, as in:

```
(%i20) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-%pi,%pi,y,-%pi,%pi ) ,
              cut(key) );
```

We need to arrange that the horizontal canvas width is about 1.4 time the vertical canvas height in order that geometrical shapes look closer to reality. For the present plot we simply change the numerical values of the **`imp(...)`** function (`x1,x2`) parameters:

```
(%i21) qdraw( imp( [g = 0.4,g = 0.7,g = 0.9] ,x,-4.2,4.2,y,-3,3 ) ,
              cut(key) );
```

which produces a slightly different looking plot:

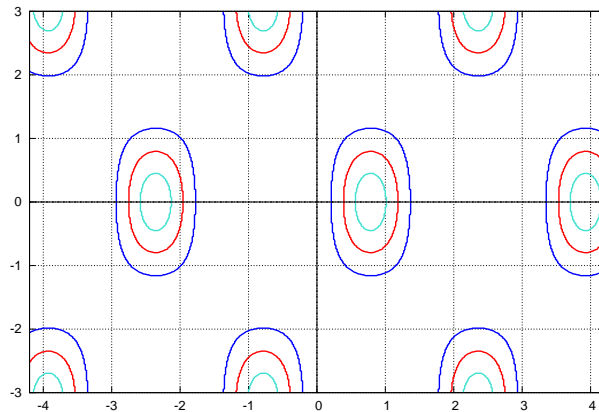


Figure 14: using  $(x1,x2) = (-4.2,4.2)$

### 5.3 Contour Plots with `contour(...)`

Since we are talking about contour plots, this is a natural place to give some examples of the **qdraw** package's **contour(...)** function which has two forms:

```
contour( expr,x,x1,x2,y,y1,y2,cvals( v1,v2,...),options )
contour( expr,x,x1,x2,y,y1,y2, crange(n,min,max), options ) .
```

where `expr` is assumed to be a function of  $(x,y)$  and the first form allows the setting of `expr` to the supplied numerical values, while the second form allows one to supply the number of contours (`n`), the minimum value for a contour (`min`) and the maximum value for a contour (`max`). If we use the most basic `cvals(...)` form (ignoring options):

```
(%i22) qdraw( contour(g, x,-4.2,4.2, y,-3,3, cvals(0.4,0.7,0.9) ) );
```

we get a "plain jane" contour plot having line width 1, the key, grid, and xy-axes removed, in "black": Since

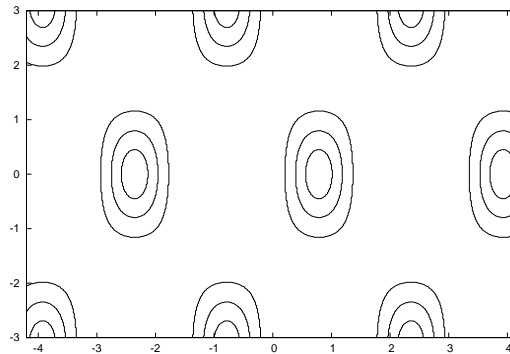


Figure 15: simplest default contour example

the quick plot functions **ex** and **imp** both use the rotating default colors which cannot be turned off, we would have to use the **imp1** function (which we have not yet discussed) with some of its options, to get the same results as the default use of **contour** produces. The available "options" which can be used in any order but after the required first eight arguments, are **lw(n)**, **lc(color)**, and **add(options)**, where the "add options" are any or all of the set `[grid,xaxis,yaxis,xyaxes]`.

Thus the following invocation of **contour**:

```
(%i23) qdraw( contour(g,x,-4.2,4.2,y,-3,3,cvals(0.4,0.7,0.9),
    lw(2),lc(brown) ), ipgrid(15) );
```

produces:

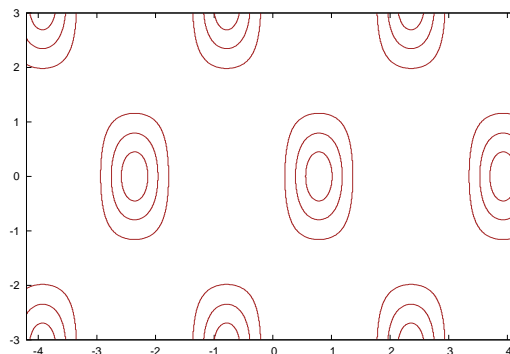


Figure 16: adding `lw(2)`, `lc(brown)`

We also added the separate **qdraw** function **ipgrid** with argument 15 to over-ride the **qdraw** default value of the **draw2d** parameter `ip_grid_in` . The **draw2d** default for this parameter is 5, which results in some "jag-gies" in implicit plots. The default value inside the **qdraw** package is 10, which generally produces smoother plots, but the drawing process takes more time, of course. For our example here, we increased this parameter from 10 to 15 to get a smoother plot at the price of increased drawing time.

Here is an example of using the second, "crange", form of **contour**:

```
(%i24) qdraw( contour(g,x,-4.2,4.2,y,-3,3,crange(4,0.2,0.9),
    lw(2),lc(brown) ), ipgrid(15) )$
```

which produces the plot:

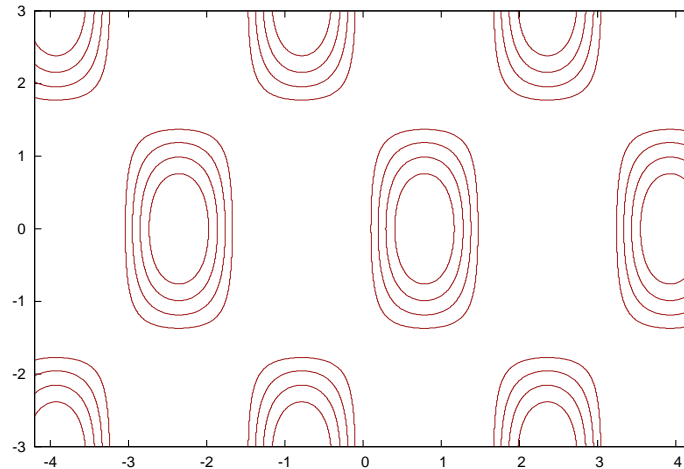


Figure 17: using `crange(4,.2,.9)`

A final example illustrates the **contour** option **add**:

```
(%i25) qdraw( contour(sin(x)*sin(y),x,-2,2,y,-2,2,crange(4,0.2,0.9),
    lw(3),lc(blue),add(xyaxes) ), ipgrid(15) )$
```

with the plot:

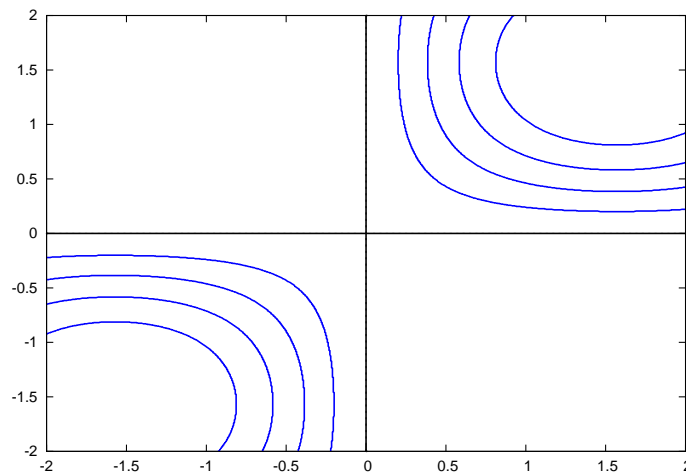


Figure 18: using `add( xyaxes )`

## 5.4 Density Plots with `qdensity(...)`

A type of plot closely related to the contour plot is the density plot, which paints small regions of the graphics window with a variable color chosen to highlight regions where the function of two variables takes on large values. A completely separate density plotting function, **qdensity**, is supplied in the `qdraw` package. The **qdensity** function is completely independent of the default conventions and syntax associated with the function **qdraw**.

The syntax of **qdensity** is:

`qdensity(expr, [x, x1, x2, dx], [y, y1, y2, dy], options palette(p), pic(...))`, where the two optional arguments are `palette(p)` and `pic(type, filename)`. The `x` interval `(x1, x2)` is divided into subintervals of size `dx`, and likewise the `y` interval `(y1, y2)` is divided into subintervals of size `dy`.

If the `palette(p)` option is not present, a default "shades of blue" density plot is drawn (which corresponds to `palette = [1, 3, 8]`). To use the `palette` option, the argument "p" can be either `blue`, `gray`, `color`, or a three element list `[n1, n2, n3]`, where `(n1, n2, n3)` are positive integers which select functions to apply respectively to red, green, and blue.

To use the `pic(...)` option, the type is `eps`, `eps_color`, `jpg`, or `png`, and the filename is a string like "case5a". As usual, use "x" and "y" if `expr` depends explicitly on `x` and `y`, or use "u" and "v" if `expr` depends explicitly on `u` and `v`, etc.

A simple function of two variables to try is  $f(x, y) = xy$ , which increases from zero at the origin to 1 at (1,1).

```
(%i26) qdensity(x*y, [x, 0, 1, 0.2], [y, 0, 1, 0.2])$
```

This produces the density plot:

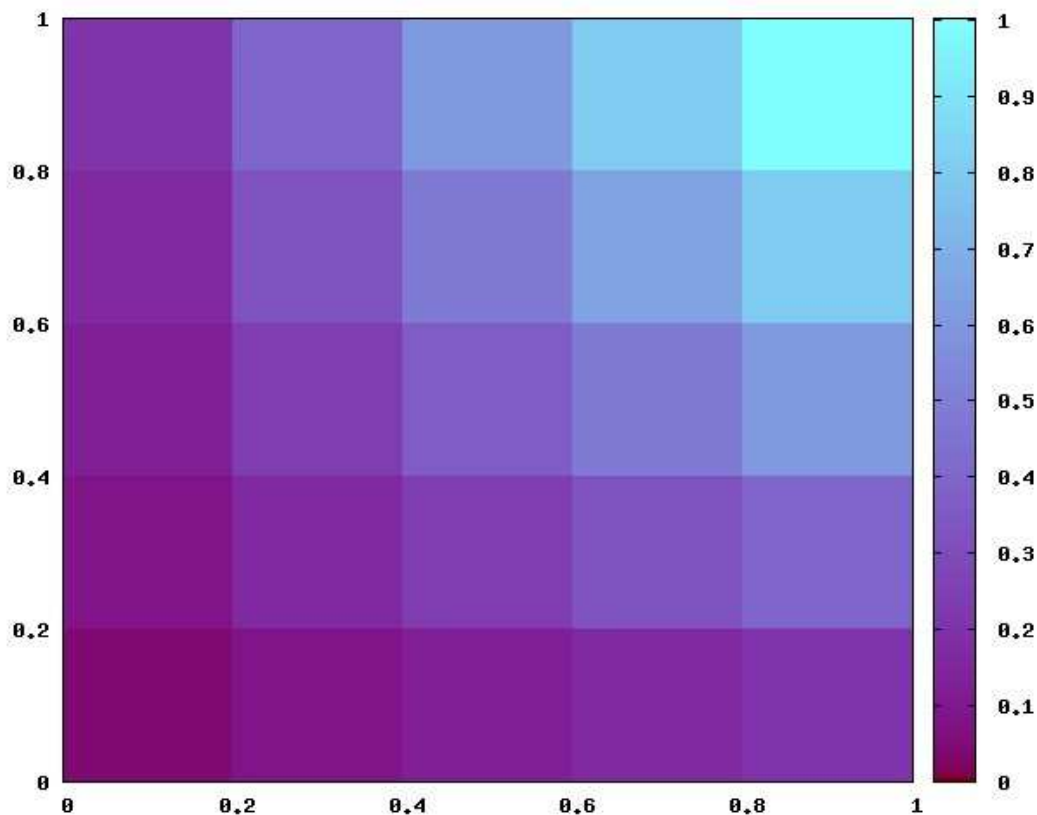


Figure 19: default palette density plot

If we use the gray palette option

```
(%i27) qdensity(x*y,[x,0,1,0.2],[y,0,1,0.2],  
               palette(gray) )$
```

we get

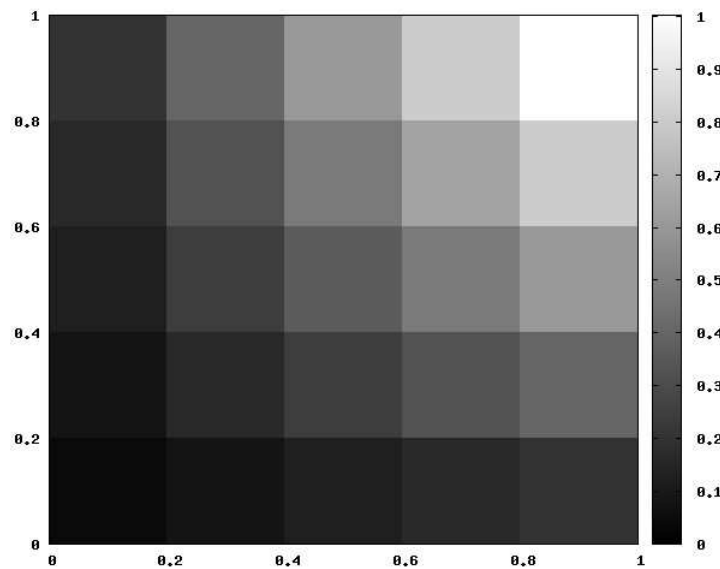


Figure 20: palette(gray) option

while if we use palette(color), we get

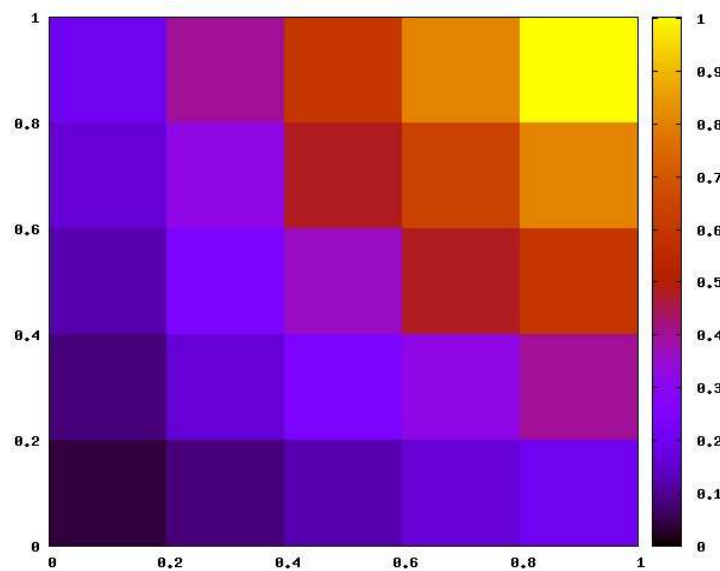


Figure 21: palette(color) option

To get a finer sampling of the function, you should decrease the values of  $dx$  and  $dy$  to  $0.05$  or less. Using the default palette choice with the interval choice  $0.05$ ,

```
(%i28) qdensity(x*y,[x,0,1,0.05],[y,0,1,0.05])$
```

yields a refined density plot with  $20 \times 20 = 400$  painted rectangular panels.

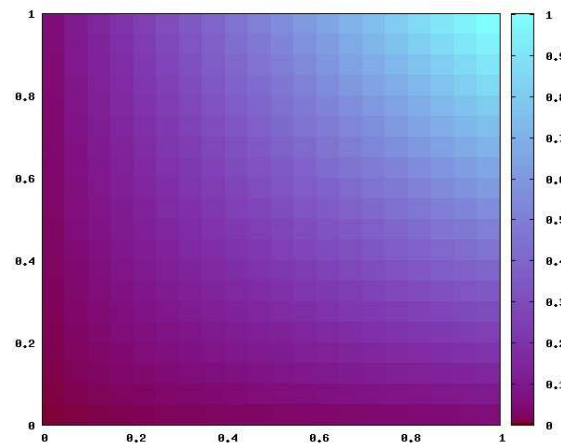


Figure 22: interval set to 0.05

A more interesting function to look at is  $f(x,y) = \sin(x) \sin(y)$ .

```
(%i29) qdensity(sin(x)*sin(y),[x,-2,2,0.05],[y,-2,2,0.05])$
```

which yields

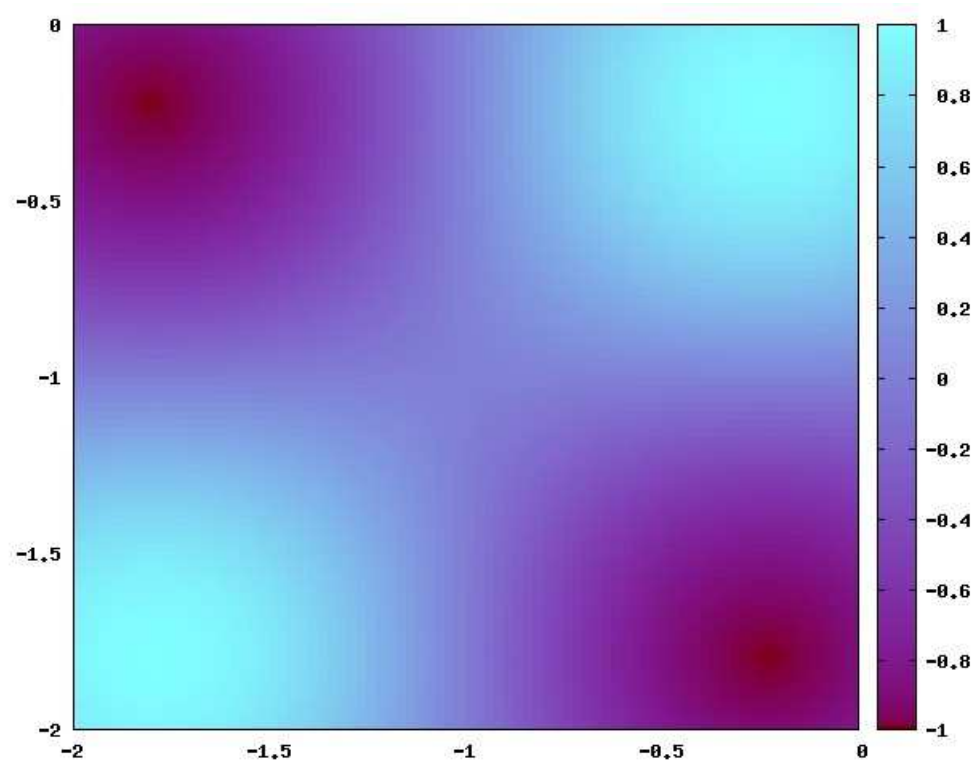


Figure 23:  $\sin(x) \sin(y)$



## 5.5 Explicit Plots with Greater Control: `ex1(...)`

If we are willing to deal with one explicit function or expression at a time, we get more control over the plot elements if we use the `qdraw` function `ex1(...)`, which has the syntax:

```
ex1( expr, x, x1,x2, lc(c), lw(n),lk(string) ) .
```

As usual, if the expression `expr` is actually a function of `u`, then  $x \rightarrow u$ . The first four arguments are required and must be in the first four slots. The last three arguments are all optional and can be in any order.

Let's illustrate the use of `ex1(...)` by displaying a simple curve and the tangent and normal at one point of the curve. We will use the curve  $y = x^2$  with the "slope"  $dy/dx = 2x$ , and construct the tangent line tangent at the point  $(x_0, y_0)$ :

$$(y - y_0) = m(x - x_0)$$

where  $m$  is the slope at  $(x_0, y_0)$ . As we discuss in the next chapter, the normal line through the same point is

$$(y - y_0) = (-1/m)(x - x_0).$$

For the point  $x_0 = 1, y_0 = 1, m = 2$ , the tangent line is  $y = 2x - 1$  and the normal line is  $y = -x/2 + 3/2$ .

```
(%i30) qdraw( xr(-1.4,2.8),yr(-1,2),
             ex1(x^2,x,-2,2,lw(5),lc(brown),lk("X^2")),
             ex1(2*x-1,x,-2,2,lw(2),lc(blue),lk("TANGENT")),
             ex1(-x/2 + 3/2,x,-2,2,lw(2),lc(magenta),lk("NORMAL")) ,
             pts( [ [1,1] ],ps(2),pc(red) ) )$
```

Note that we were careful to force the x-range to be about 1.4 times as great as the y-range (to get the correct geometry of the tangent and normal lines). The resulting plot is:

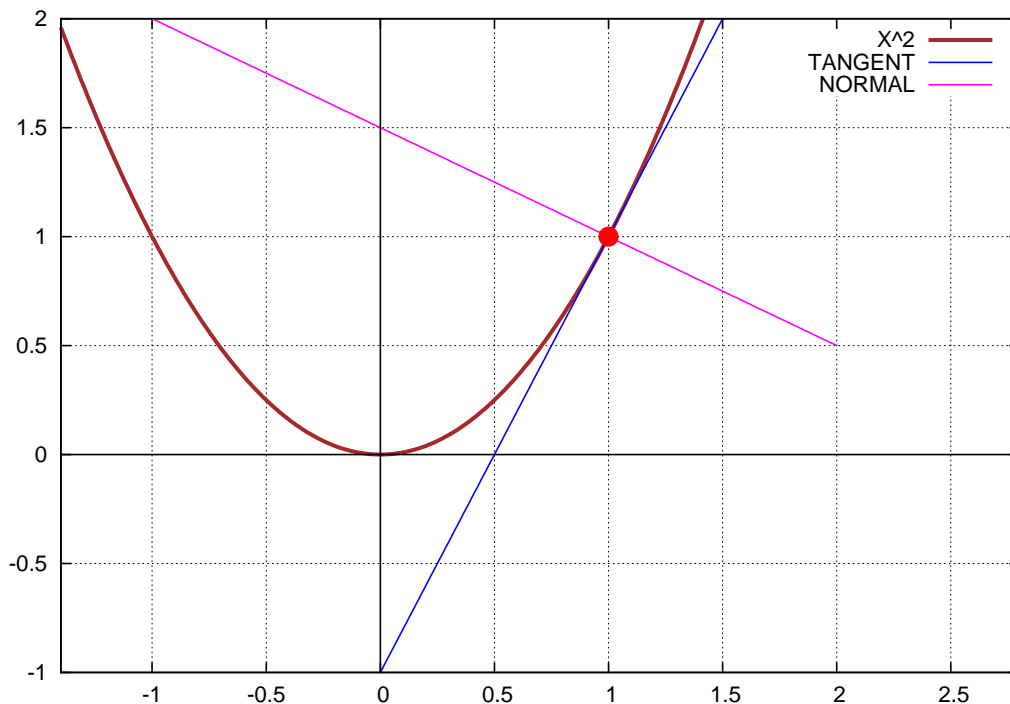


Figure 24: plot using `ex1(...)`

Here we use **ex1** to plot the first few Bessel functions of the first kind  $J_n(x)$  for integral  $n$  and real  $x$ ,

```
(%i31) qdraw( ex1(bessel_j(0,x),x,0,20,lc(red),lw(6),lk("bessel_j ( 0, x)")),
             ex1(bessel_j(1,x),x,0,20,lc(blue),lw(5),lk("bessel_j ( 1, x)")),
             ex1(bessel_j(2,x),x,0,20,lc(brown),lw(4),lk("bessel_j ( 2, x)")),
             ex1(bessel_j(3,x),x,0,20,lc(green),lw(3),lk("bessel_j ( 3, x)")) )$
```

which produces the plot:

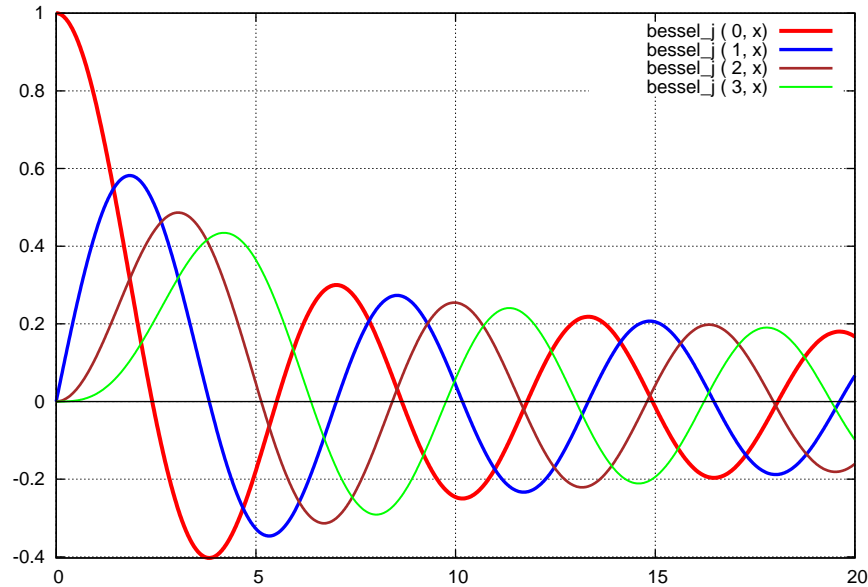


Figure 25:  $J_n(x)$

Here is a plot of  $J_0(\sqrt{x})$ :

```
(%i32) qdraw(line(0,0,50,0,lc(red),lw(2)),
             ex1(bessel_j(0, sqrt(x)),x,0,50,lc(blue),
             lw(7),lk("J0( sqrt(x) )")) )$
```

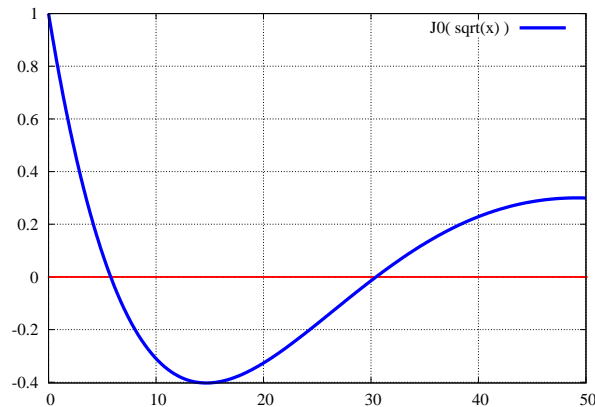


Figure 26:  $J_0(\sqrt{x})$

We chose to emphasize the axis  $y = 0$  with a red line supplied by another of the **qdraw** functions, **line**, which we will discuss later in the section on geometric figures. Placing the **line** element before **ex1(..)** causes the curve to write "over" the line, rather than the reverse.

## 5.6 Explicit Plots with ex1(...) and Log Scaled Axes

The name "log plot" usually refers to a plot of  $\ln(y)$  vs  $x$  using linear graph paper, which is equivalent to a plot of  $y$  vs  $x$  on graph paper which uses a "logarithmic scale" on the vertical axis. Given an expression  $g$  depending on  $x$ , you can either use the syntax `qdraw( ex1( log(g), x, x1, x2 ), other options )` to generate such a "log plot" or `qdraw( ex1(g, x, x1, x2), log(y), other options )`.

Let's show the differences using the function  $f(x) = x e^{-x}$ , but using an expression called  $g$  rather than a Maxima function.

```
(%i33) g : x*exp(-x)$
(%i34) qdraw( ex1( log(g), x, 0.001, 10, lc(red) ), yr(-8, 0) )$
```

which displays the plot

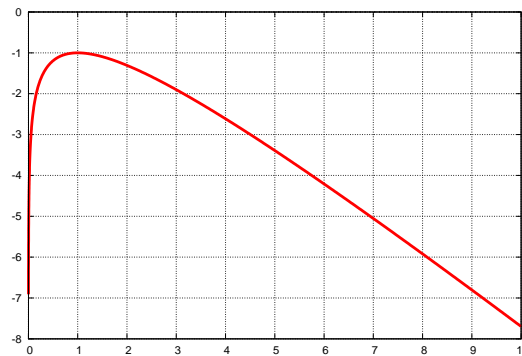


Figure 27: Linear Graph Paper Plot of  $\ln(g)$

The numbers on the vertical axis correspond to values of  $\ln(g)$ . Since  $g$  is singular at  $x = 0$ , we have avoided that region by using  $x_1 = 0.001$ .

The second way to get a "log plot" of  $g$  is to request "semi-log" graph paper which has the vertical axis marked using a logarithmic scale for the values of  $g$ . Using the **log(y)** option of the **qdraw** function, we use:

```
(%i35) qdraw( ex1(g, x, 0.001, 10, lc(red) ),
              yr(0.0001, 1), log(y) )$
```

The **yr(y1,y2)** option takes into account the numerical limits of  $g$  over the  $x$  interval requested. The minimum value of  $g$  is 0.005 which occurs at  $x = 10$ . The maximum value of  $g$  is about 0.37. The resulting plot is:

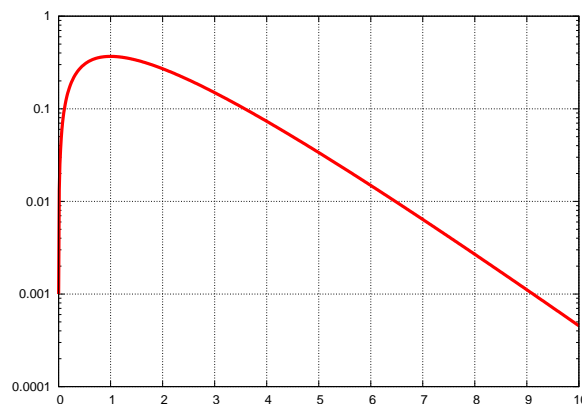


Figure 28: Log Paper Plot of  $g$

The name "log-linear plot" can be used to mean "x axis marked with a log scale, y axis marked with a linear scale". Using the same function, we generate this plot by using the **log(x)** option to **qdraw**:

```
(%i36) qdraw( exl(g, x, 0.001,10,lc(red),lw(7) ),
              yr(0,0.4), log(x) )$
```

This generates the plot

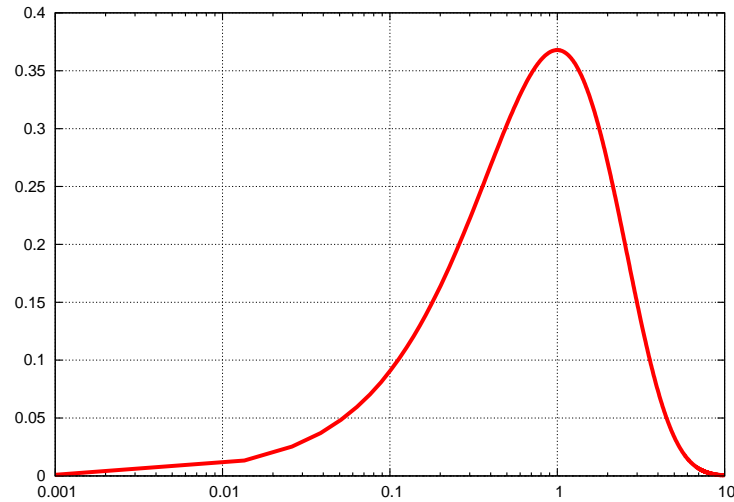


Figure 29: Log-Linear Plot of  $g$

Scientists and engineers normally like to use a log scaled axis for a variable which varies over many powers of ten, which is not the case for our example.

Finally, we can request "log-log paper" which has both axes marked with a log scale, by using the **log(xy)** option to **qdraw**.

```
(%i37) qdraw( exl(g, x, 0.001,10,lc(red) ),
              yr(0.0001,1), log(xy) )$
```

which produces

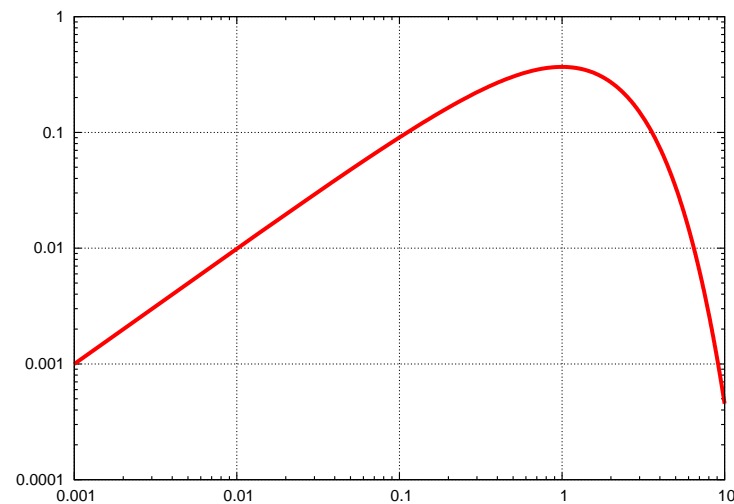


Figure 30: Log-Log Plot of  $g$

## 5.7 Data Plots with Error Bars: `pts(...)` and `errorbars(...)`

In Chapter One of *Maxima by Example*, Section 1.5.8, we created a data file called "fit1.dat", which can be downloaded from the author's webpage. We will use that data file, together with "fit2.dat", also available, to illustrate making simple data plots using the **qdraw** functions **pts(...)** and **errorbars(...)**. The syntax of **pts(...)** is:

```
pts( pointlist, pc(c), ps(s), pt(t), pj(lw), pk(string) )
```

The only required argument is the first argument "pointlist" which has the form:

```
[ [x1,y1], [x2,y2], [x3,y3], ... ] .
```

The remaining arguments are all optional and may be entered in any order following the first required argument.

The optional argument `pc(c)` overrides the default color (black), for example, `pc(red)`.

The optional argument `ps(s)` overrides the default size (3), and an example is `ps(2)`.

The optional argument `pt(t)` overrides the default type (7, which is the integer used for `filled_circle`: see the Maxima manual index entry for "point\_type"); an example would be `pt(8)`, which would use an open "up\_triangle" instead of a filled circle.

The optional argument `pj(lw)`, if present, will cause the points provided by the nested list "pointlist" to be joined using a line whose width is given by the argument of `pj`; an example is `pj(2)` which would set the line width to the value 2.

The optional argument `pk(string)` provides text for a key entry for the set of points represented by pointlist; as example is `pk("case x^2")`.

Before making the data plot, let's look at the data file contents from inside Maxima:

```
(%i38) printfile("fit1.dat")$
1 1.8904
2 3.0708
3 3.9215
4 5.1813
5 5.9443
6 7.0156
7 7.8441
8 8.8806
9 9.8132
10 11.129
```

We next use Maxima's **read\_nested\_list** function to create a list of data points from the data file.

```
(%i39) plist : read_nested_list("fit1.dat");
(%o39) [[1, 1.8904], [2, 3.0708], [3, 3.9215], [4, 5.1813], [5, 5.9443],
[6, 7.0156], [7, 7.8441], [8, 8.880599999999999], [9, 9.8132], [10, 11.129]]
```

The most basic plot of this data uses the **pts(...)** function defaults:

```
(%i40) qdraw( pts(plist) )$
```

which produces:

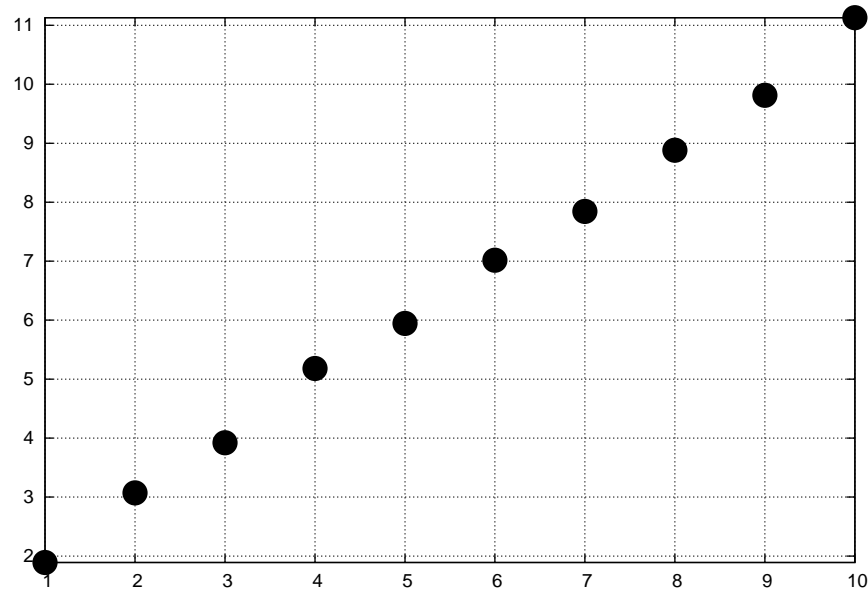


Figure 31: Using pts(...) Defaults

We can use the **qdraw** functions **xr(...)** and **yr(...)** to override the default range selected by **draw2d**, and decrease the point size:

```
(%i41) qdraw( pts(plist, ps(2)), xr(0,12), yr(0,15) )$
```

with the result:

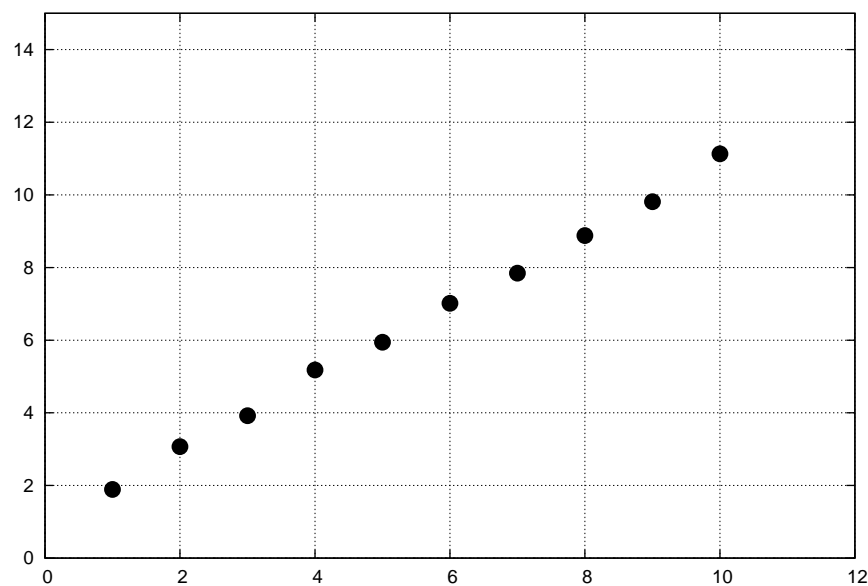


Figure 32: Adding ps(2), xr(..), yr(..)

Now we add color and a key string, as well as simple error bars corresponding to an assumed uncertainty of the y value of plus or minus 1 for all the data points.

```
(%i42) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
              key(bottom), errorbars( plist, 1) )$
```

which looks like:

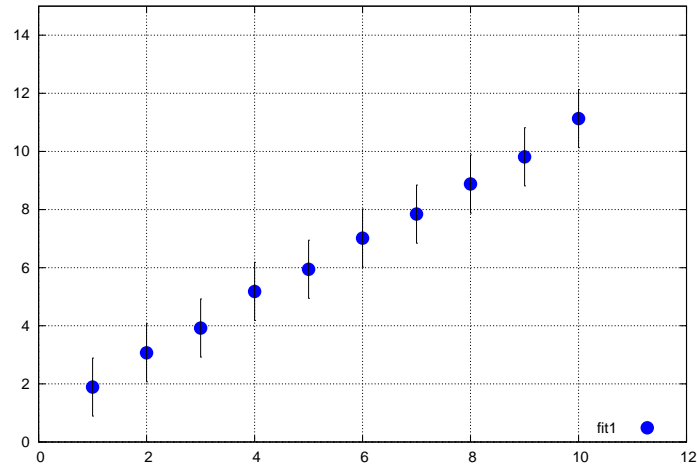


Figure 33: Adding pc(blue) and Simple Error Bars

The default error bar line width of 1 is almost too small to see, so we thicken the error bars and add color

```
(%i43) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
              key(bottom), errorbars( plist, 1, lw(3),lc(red) ) )$
```

with the result:

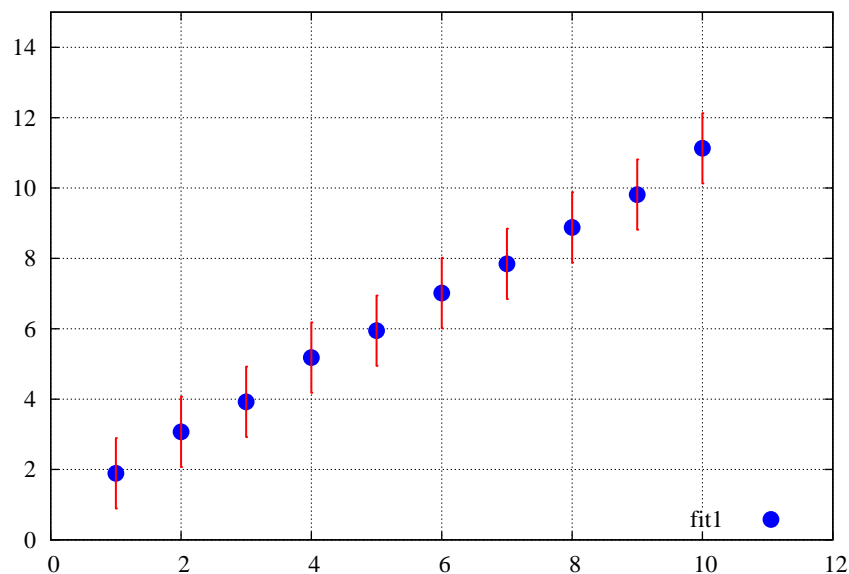


Figure 34: Adding lw(3), lc(red) to errorbars(...)

The difference in the fonts is due to my using `pic(eps, "ch5p27h", font("Times-Roman",18) )` to create the eps graphic instead of just `pic(eps, "ch5p27h" )` as another argument to **qdraw**.

If the data set has individual uncertainties in the  $y$  value, we create a list `dy1`, say, of the values `dy1`, `dy2`, `dy3`, ... and use the syntax:

```
errorbars( pointlist, dylist, lw(n), lc(c) )
```

Here is an example:

```
(%i44) dy1 : [0.2,0.3,0.5,1.5,0.8,1,1.4,1.8,2,2];
(%o44)      [0.2, 0.3, 0.5, 1.5, 0.8, 1, 1.4, 1.8, 2, 2]
(%i45) map(length,[plist,dy1] );
(%o45)      [10, 10]
(%i46) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
              key(bottom), errorbars( plist, dy1, lw(3),lc(red) ) ) )$
```

with the result

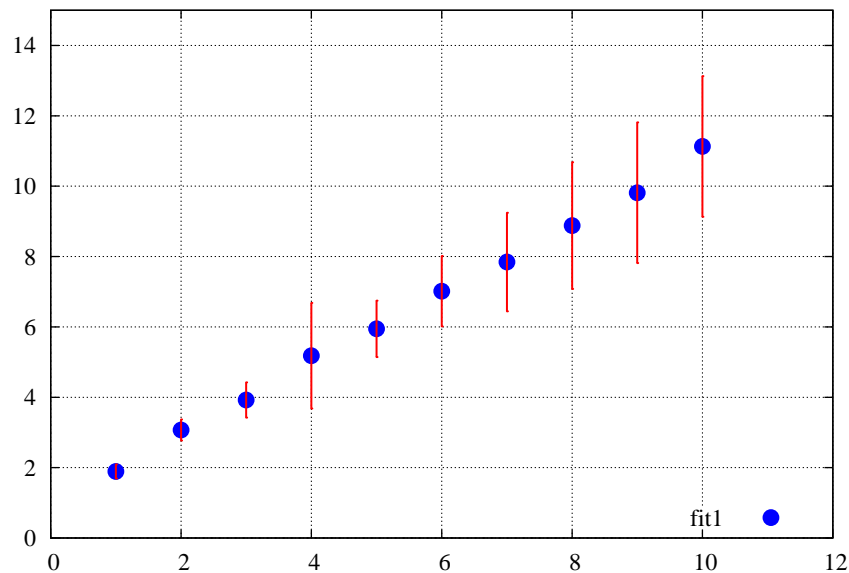


Figure 35: Using a list of  $dy$  values with `errorbars(..)`

We now repeat the least squares fit of this data which we carried out in Chapter 1. See our discussion there for an explanation of what we are doing here.

```
(%i47) display2d:false$
(%i48) pmatrix : apply( 'matrix, plist );
(%o48) matrix([1,1.8904],[2,3.0708],[3,3.9215],[4,5.1813],[5,5.9443],
              [6,7.0156],[7,7.8441],[8,8.880599999999999],[9,9.8132],
              [10,11.129])
(%i49) load(lsquares);
(%o49) "C:/PROGRA~1/MAXIMA~4.0/share/maxima/5.15.0/share/contrib/lsquares.mac"
(%i50) soln : (lsquares_estimates(pmatrix,[x,y],y=a*x+b,
                                [a,b]), float(%%) );
(%o50) [[a = 0.99514787679748,b = 0.99576667381004]]
(%i51) [a,b] : (fpprintprec:5, map( 'rhs, soln[1] ) )$
(%i52) [a,b];
(%o52) [0.995,0.996]
(%i53) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
              key(bottom), errorbars( plist, dy1, lw(3),lc(red) ),
              ex1( a*x + b,x,0,12, lc(brown),lk("linear fit") ) ) )$
```



We use the **qdraw** function **ex1(...)** to add the line  $f(x) = ax + b$  to the data plot. The resulting plot with the least squares fit added is then:

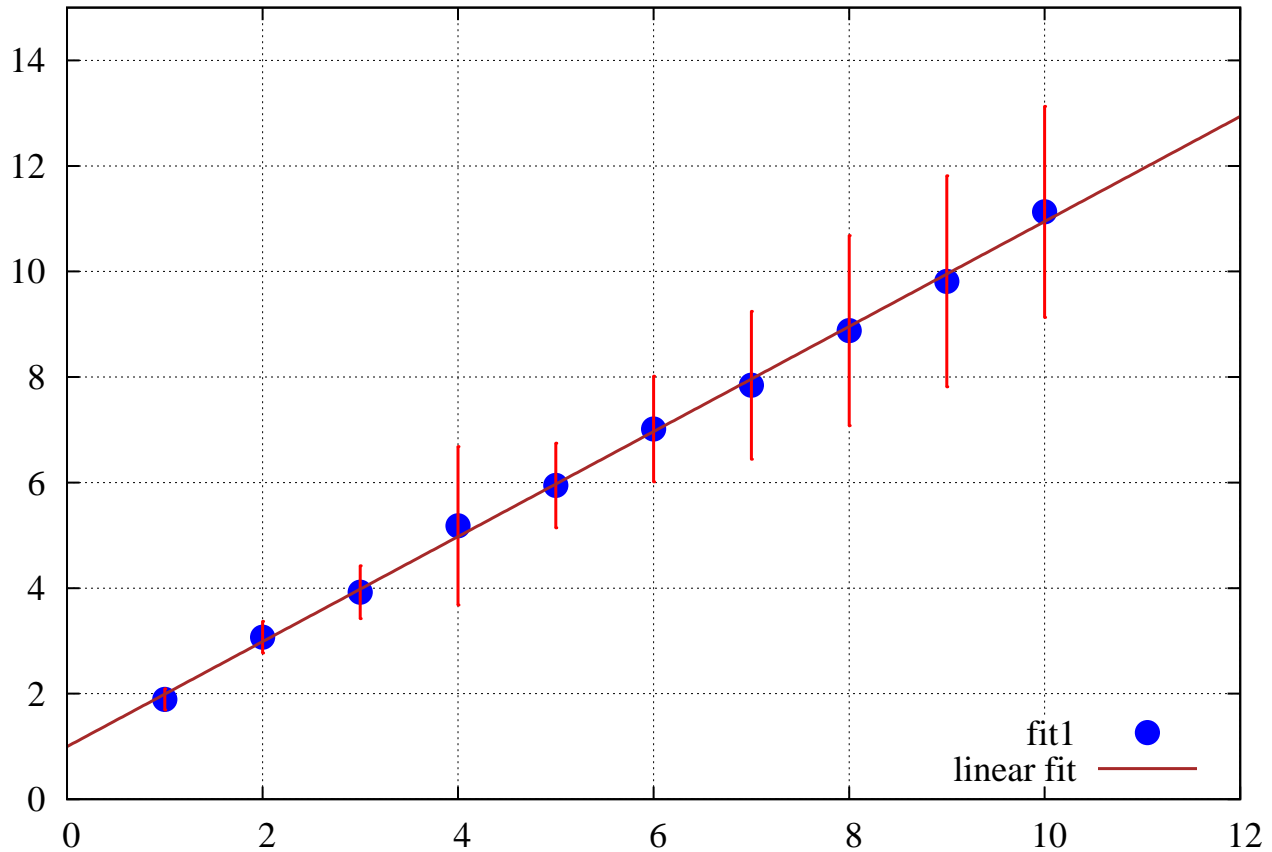


Figure 36: Adding the Linear Fit Line

Now we add the data in the file "fit2.dat":

```
(%i54) printfile("fit2.dat");
1 0.9452
2 1.5354
3 1.9608
4 2.5907
5 2.9722
6 3.5078
7 3.9221
8 4.4403
9 4.9066
10 5.5645
(%o54) "fit2.dat"
(%i55) p2list: read_nested_list("fit2.dat");
(%o55) [[1,0.945],[2,1.5354],[3,1.9608],[4,2.5907],[5,2.9722],[6,3.5078],
[7,3.9221],[8,4.4403],[9,4.9066],[10,5.5645]]
(%i56) qdraw( pts(plist,pc(blue),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
key(bottom), errorbars( plist, dyl, lw(3),lc(red) ),
ex1( a*x + b,x,0,12, lc(brown),lk("linear fit 1") ),
pts(p2list, pc(magenta),pk("fit2"),ps(2)),
errorbars( p2list,0.5,lw(3) ) )$
```

Here is the plot with the second data set:

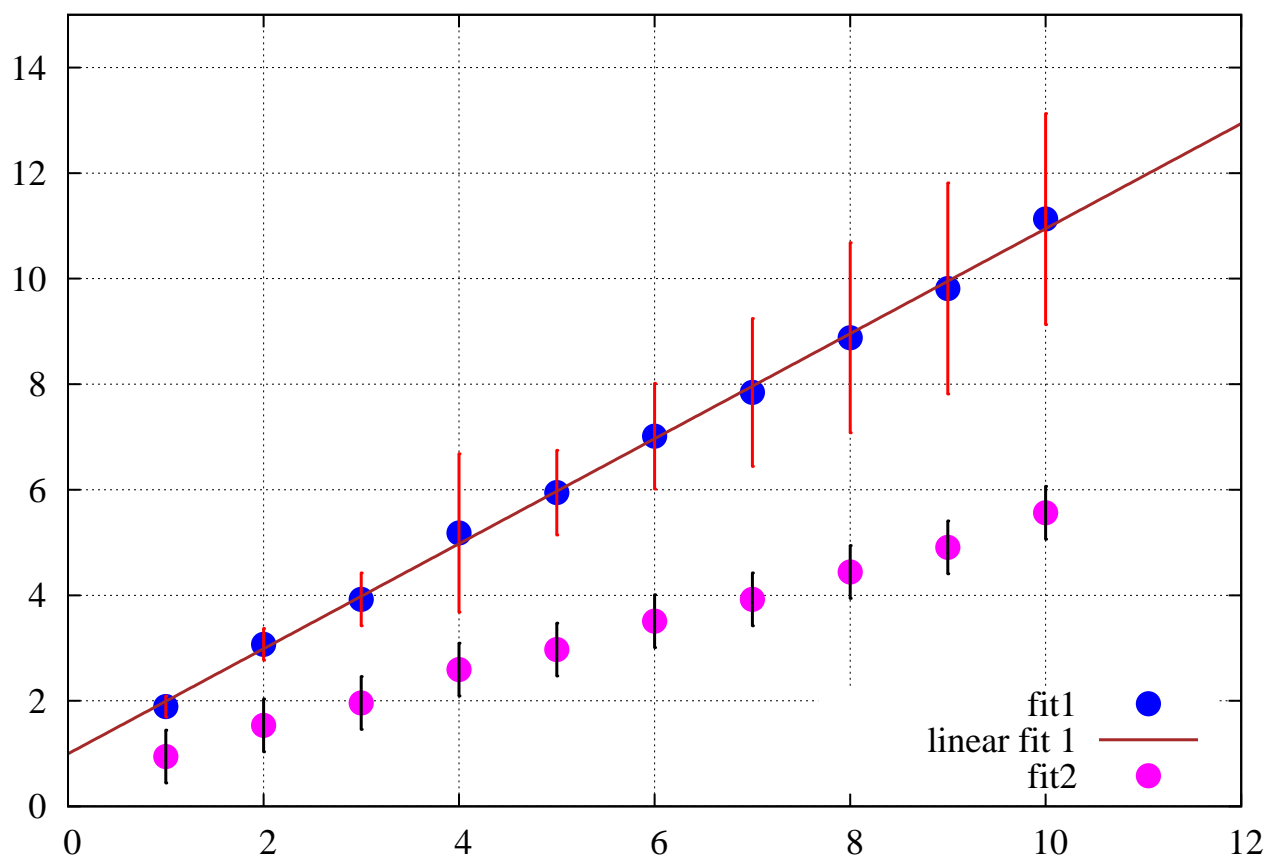


Figure 37: Adding the Second Set of Data

We could then find the least squares fit to the data set 2 and again use the function **ex1(...)** to add that fit to our plot, and add any other features desired.

## 5.8 Implicit Plots with Greater Control: `imp1(...)`

If we are willing to deal with one implicit equation of two variables at a time, we get more control over the plot elements if we use the `qdraw` function `imp1(...)`, which has the syntax:

```
imp1( eqn, x, x1,x2, y, y1,y2, lc(c), lw(n),lk(string) ) .
```

As usual, if the equation `eqn` is actually a function of the pair of variables  $u$  and  $v$ , then let  $x \rightarrow u$ , and  $y \rightarrow v$ . The first seven arguments are required and must be in the first seven slots. The last three arguments are all optional and can be in any order.

Let's illustrate the use of `imp1(...)` by displaying a translated and rotated ellipse, together with the rotated  $x$  and  $y$  axes. In the following, `eqn1` describes the rotated ellipse, `eqn2` describes the rotated  $x$  axis, and `eqn3` describes the rotated  $y$  axis. The angle of rotation is about  $63.4^\circ$  (counter clockwise), which corresponds to  $\tan \phi = 2$ . Notice that we take care to get the  $x$ -axis range about 1.4 times the  $y$ -axis range, in order to get the geometry approximately right.

```
(%i1) eqn1 : 5*x^2 + 4*x*y + 8*y^2 - 16*x + 8*y - 16 = 0$
(%i2) eqn2 : y+1 = 2*(x-2)$
(%i3) eqn3 : y+1 = -(x-2)/2$
(%i4) qdraw( imp1(eqn1,x,-2,6.4,y,-4,2,lc(red),lw(6),lk("ELLIPSE")),
              imp1(eqn2,x,-2,6.4,y,-4,2,lc(blue),lw(4),lk("ROT X AXIS")),
              imp1(eqn3,x,-2,6.4,y,-4,2,lc(brown),lw(4),lk("ROT Y AXIS")),
              pts([ [2,-1] ],ps(2),pc(magenta),pk("TRANSLATED ORIGIN")) ) )$
```

We get the following figure, if we increase the font size,

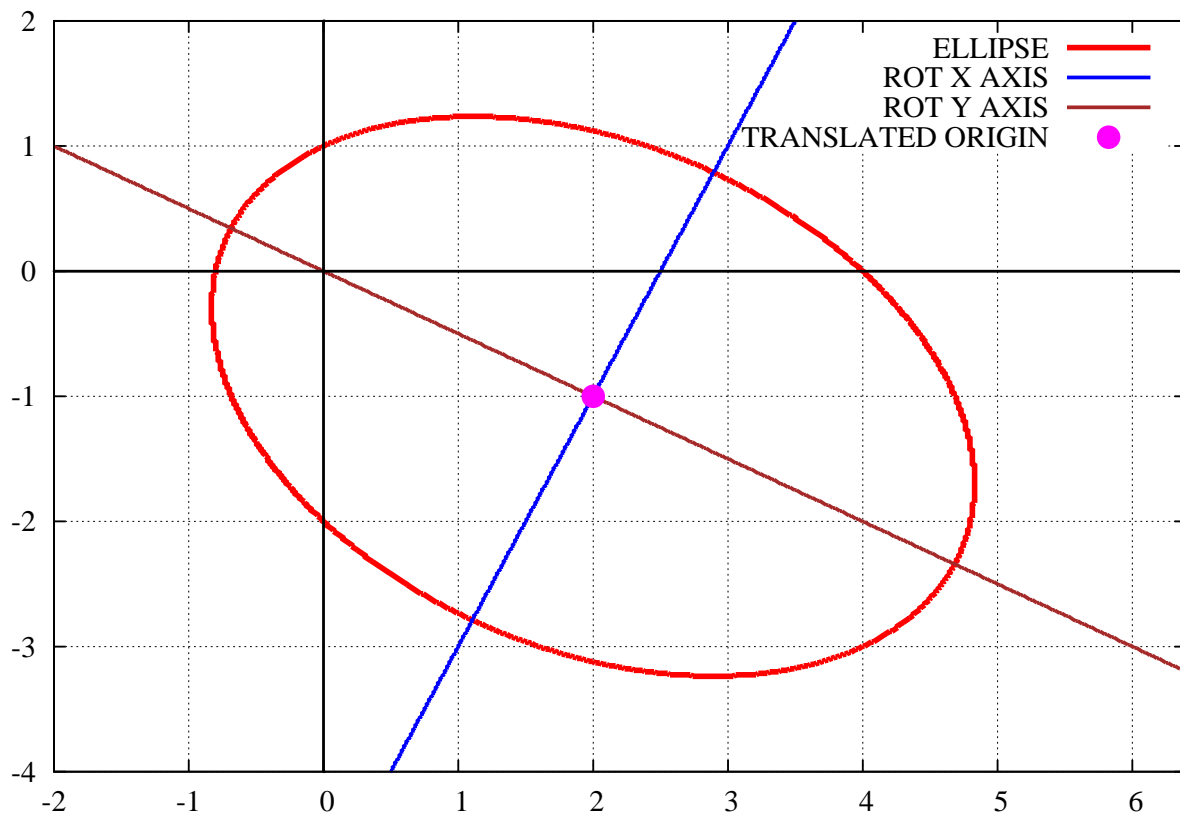


Figure 38: Rotated and Translated Ellipse

As a second example with **imp1** we make a simple plot based on the equation  $y^3 = x^2$ .

```
(%i5) qdraw( imp1(y^3=x^2,x,-3,3,y,-1,3,lw(10),lc(dark-blue)) )$
```

which produces the plot:

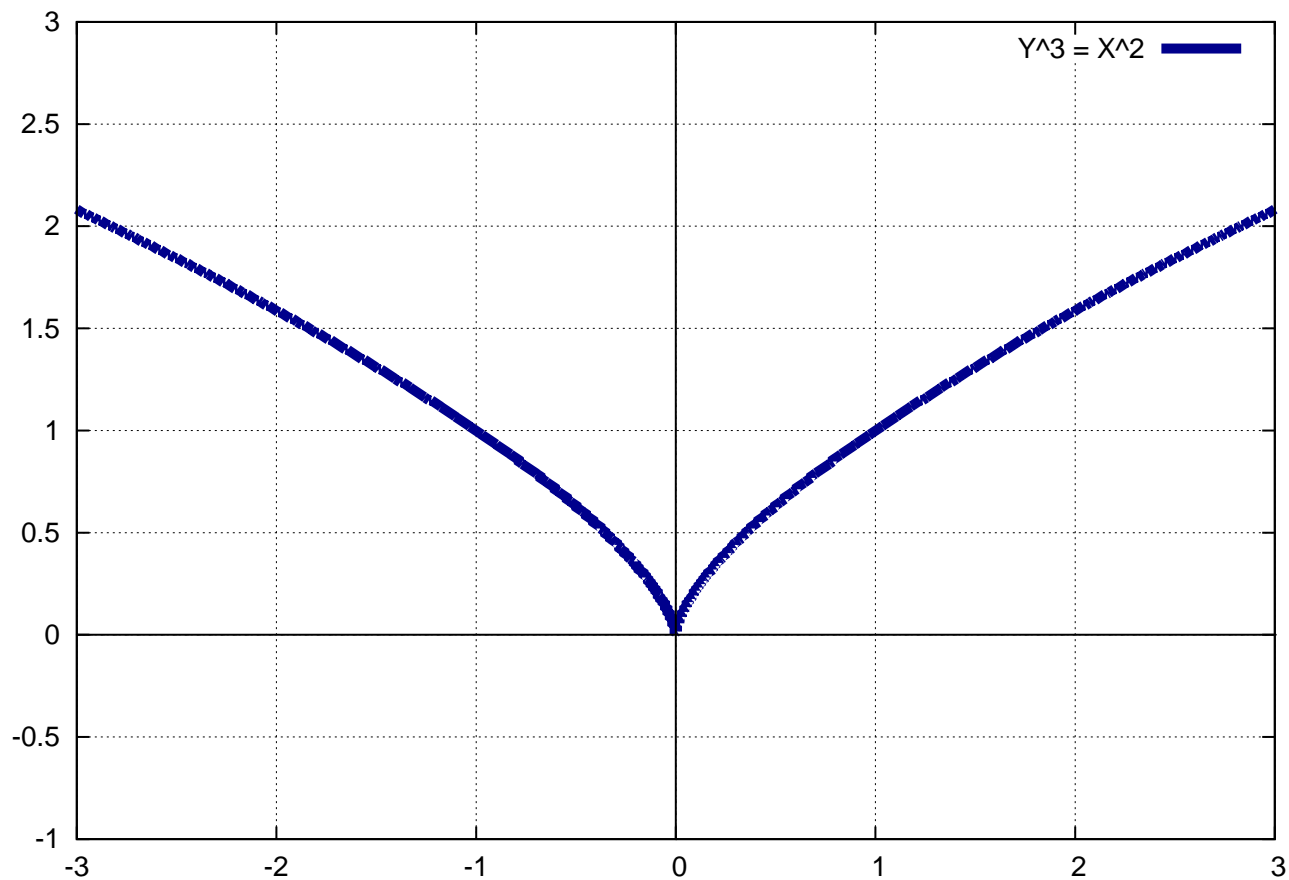


Figure 39: Implicit Plot of  $y^3 = x^2$

Notice that you can use hyphenated color choices (see Maxima color index) without the double quotes, or with the double quotes.

## 5.9 Parametric Plots with para(...)

The **qdraw** function **para** can be used to draw parametric plots and has the syntax

```
para(xofu,yofu,u,u1,u2,lw(n),lc(c),lk(string) )
```

where, as usual, the line width, line color, and key string are optional and can be in any order. The parameter "u" can, of course, be any symbol.

A simple example, in which we use "t" for the parameter, and let the x coordinate corresponding to some value of t be  $\sin(t)$ , and let the y coordinate corresponding to that same value of t be  $\sin(2t)$  is:

```
(%i6) qdraw(xr(-1.5,2),yr(-2,2),  
           para(sin(t),sin(2*t),t,0,2*pi) ,  
           pts( [ [sin(%pi/8),sin(%pi/4)] ],ps(2),pc(blue),pk("t = pi/8")),  
           pts( [ [1,0] ],ps(2),pc(red),pk("t = pi/2")) )$
```

produces the plot:

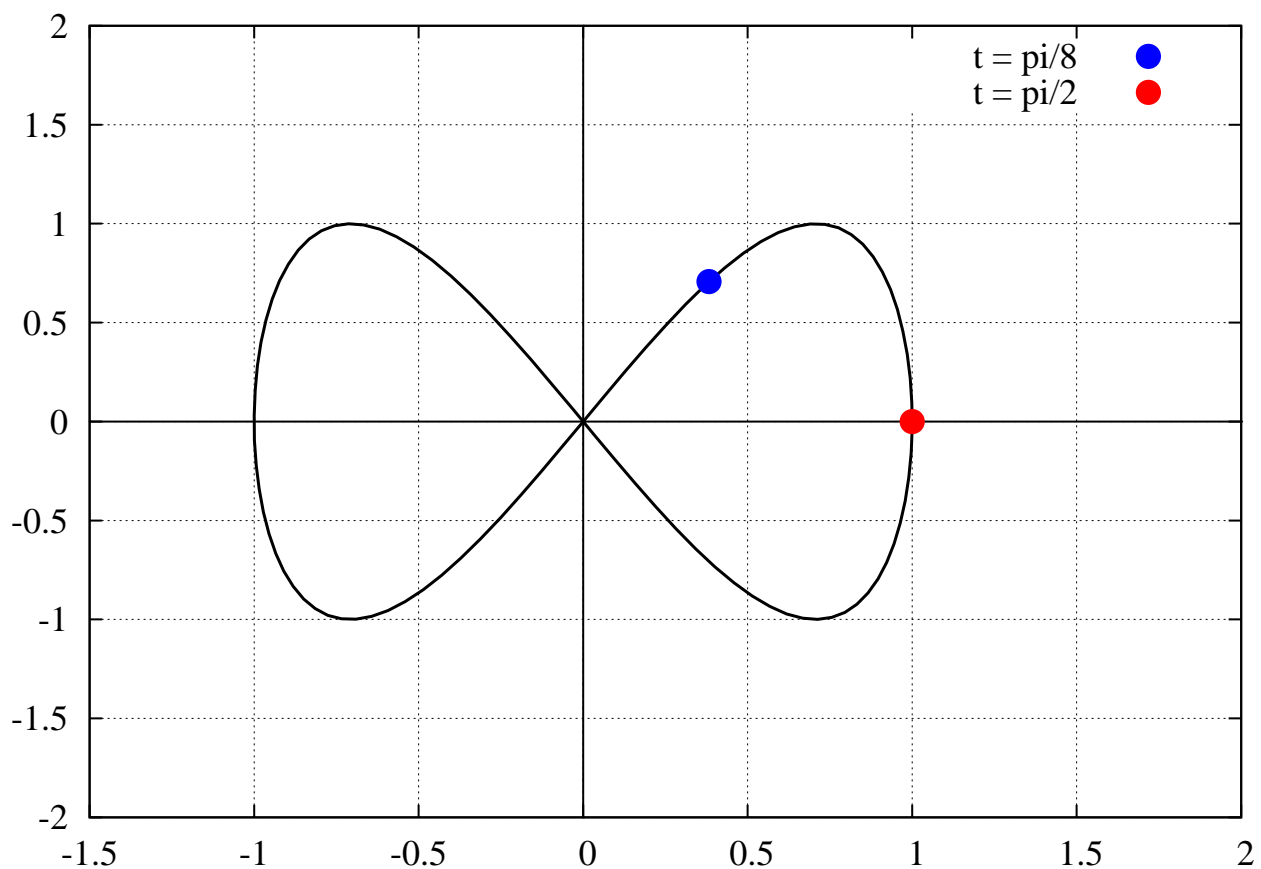


Figure 40:  $x = \sin(t)$ ,  $y = \sin(2t)$

A second example of a parametric plot has  $u$  as the parameter,  $x = 2\cos(u)$ , and  $y = u^2$ :

```
(%i7) qdraw(xr(-3,4),yr(-1,40), para(2*cos(u),u^2,u,0,2*pi) ,
      pts([ [2,0] ],ps(2),pc(blue),pk("u = 0")),
      pts([ [0,(pi/2)^2] ],ps(2),pc(red),pk("u = pi/2")),
      pts([ [-2,pi^2] ],ps(2),pc(green),pk("u = pi")),
      pts([ [0,(3*pi/2)^2] ],ps(2),pc(magenta),pk("u = 3*pi/2")) )$
```

which yields the plot:

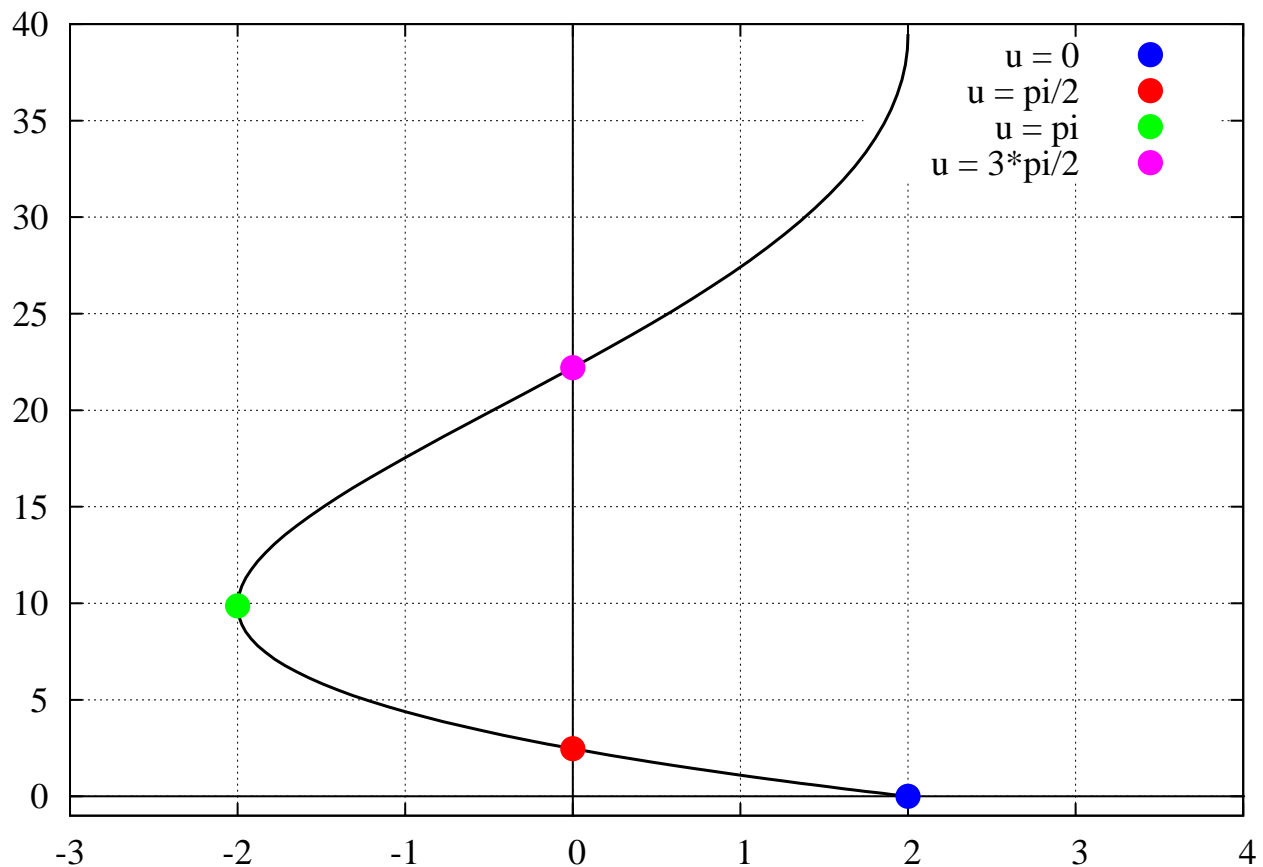


Figure 41:  $x = 2\cos(u)$ ,  $y = u^2$

## 5.10 Polar Plots with polar(...)

A "polar plot" plots the points  $(x = r(\theta) \cos(\theta), y = r(\theta) \sin(\theta))$ , where the function  $r(\theta)$  is supplied.

The **qdraw** function **polar** has the syntax:

```
polar( roftheta, theta, th1,th2, lc(c), lw(n), lk(string) )
```

where theta, th1, and th2 are in radians, and the last three arguments are optional.

A simple example is provided by the hyperbolic spiral  $r(\theta) = 10/\theta$ .

```
(%i8) qdraw( polar(10/t,t,1,3*pi,lc(brown),lw(5)),nticks(200),
            xr(-4,6),yr(-3,9),key(bottom) ,
            pts( [[10*cos(1),10*sin(1)],[5*cos(2),5*sin(2)]] ,ps(3),pc(red),pk("t = 1 rad")),
            pts([[5*cos(2),5*sin(2)]],ps(3),pc(blue),pk("t = 2 rad") ),
            line(0,0,5*cos(2),5*sin(2)) )$
```

which looks like:

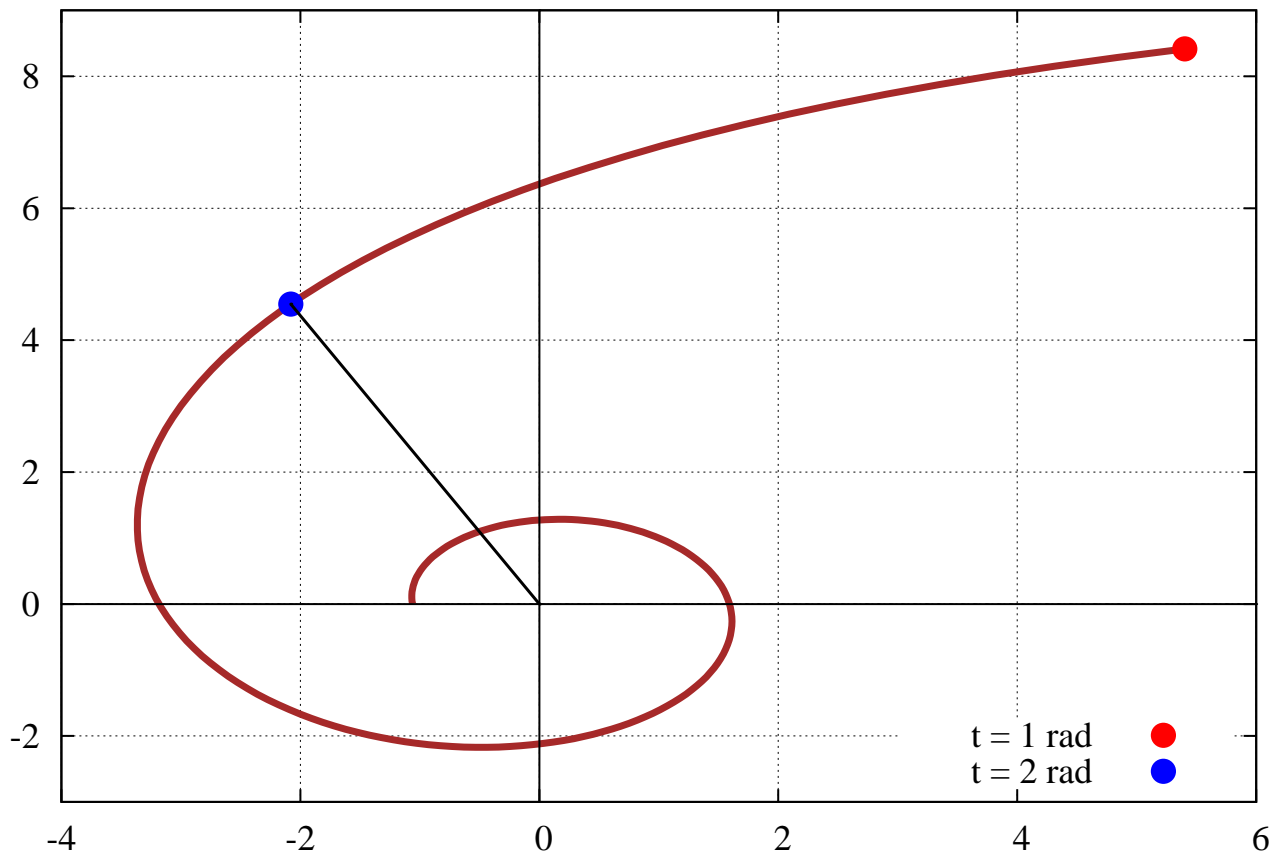


Figure 42: Polar Plot with  $r = 10/\theta$

## 5.11 Geometric Figures: line(...)

The **qdraw** function **line** has the syntax:

```
line( x1,y1,x2,y2, lc(c), lw(n), lk(string) )
```

which draws a line from  $(x1, y1)$  to  $(x2, y2)$ . The last three arguments are optional and can be in any order after the first four arguments.

For example, `line(0,0,1,1,lc(blue),lw(6),lk("radius"))` will draw a line from  $(0,0)$  to  $(1,1)$  in blue with line width 6 and with a key entry with the text 'radius'. The defaults are color black, line width 3, and no key entry.

```
(%i9) qdraw( line(0,0,1,1) )$
```

produces the default line with **draw2d**'s default range:

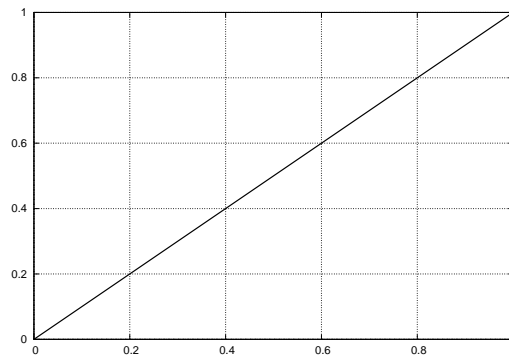


Figure 43: Default line(..)

Adding some options and extending the canvas range in both directions

```
(%i10) qdraw( line(0,0,1,1,lc(blue),lw(6),lk("radius")) ,  
             xr(0,2),yr(0,2),key(bottom),  
             pts([ [1,1] ] ,pc(red),pk("point")) )$
```

produces a line to a point marked in red:

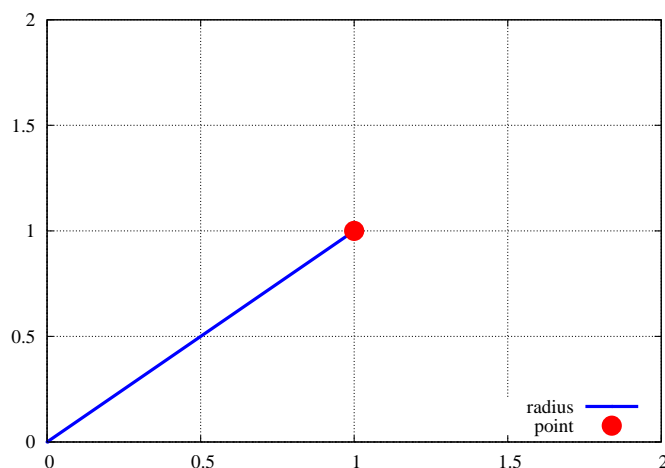


Figure 44: Adding options to line(..)



Here we define a Maxima function "doplot1(n)" in a file "doplot1.mac" which has the following contents:

```
/* file doplot1.mac */

disp("doplot1(nlw) ")$

doplot1(nlw) := block([cc,qlist,x,val,i ],
/* list of 20 single name colors */
cc : [aquamarine,beige,blue,brown,cyan,gold,goldenrod,green,khaki,
      magenta,orange,pink,plum,purple,red,salmon,skyblue,turquoise,
      violet,yellow ],
qlist : [ xr(-3.3,3) ],
for i thru length(cc) do (
  x : -3.3 + 0.3*i,
  val : line( x,-1,x,1, lc( cc[i] ),lw(nlw) ),
  qlist : append(qlist, [val] )
),
qlist : append( qlist,[ cut(all) ] ),
apply('qdraw, qlist)
)$
```

Here is a record of loading and using the function defined to produce a series of vertical colored lines.

```
(%i11) load(doplot1);
                                doplot1(nlw)
(%o11)                          c:/work2/doplot1.mac
(%i12) doplot1(20);
```

which produces the graphic (note use of **cut**(all) to get a blank canvas):

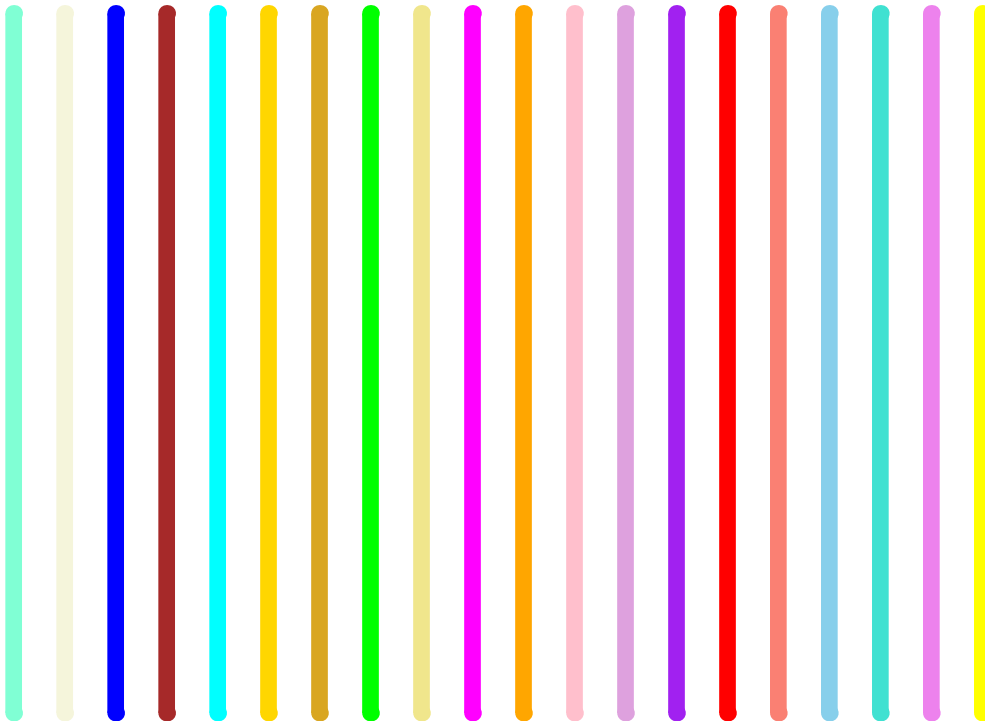


Figure 45: Using line(...) to Display Colors

## 5.12 Geometric Figures: rect(...)

The **qdraw** function **rect** has the syntax:

```
rect( x1,y1,x2,y2, lc(c), lw(n), fill(c) )
```

which will draw a rectangle with opposite corners  $(x1,y1)$  and  $(x2,y2)$ . The last three arguments are optional; without them the rectangle is drawn in black with line thickness 3 and with no fill color. An example is `rect(0,0,1,1,lc(brown),lw(2),fill(khaki) )`. We start with the basic rectangle call:

```
(%i13) qdraw( xr(-1,2),yr(-1,2),rect(0,0,1,1) )$
```

with the result

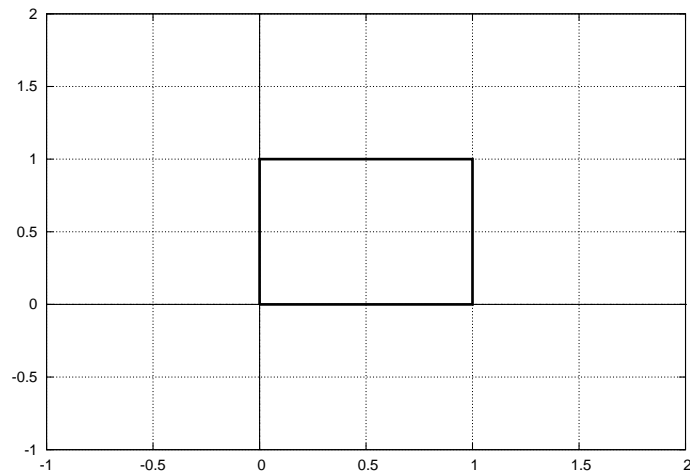


Figure 46: Default rect(0,0,1,1)

We now add some color, thickness and fill:

```
(%i14) qdraw( xr(-1,2),yr(-1,2),  
             rect(0,0,1,1,lw(5),lc(brown),fill(khaki) ) )$
```

with the output:

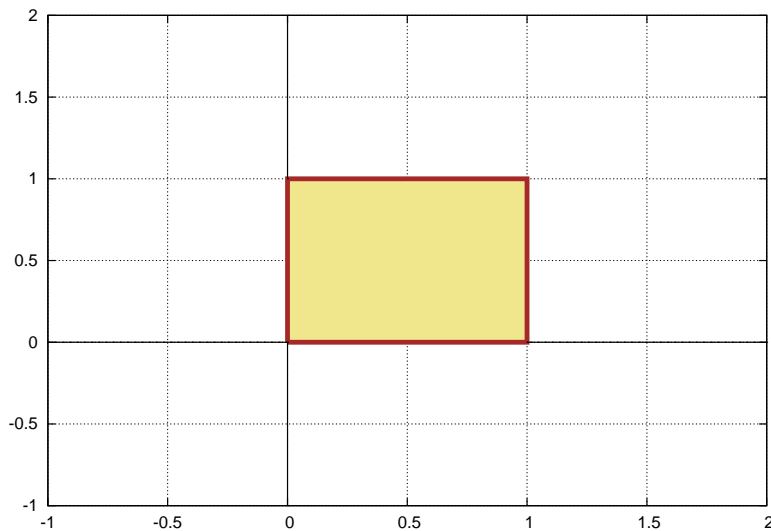


Figure 47: rect(0,0,1,1,lc(brown),lw(5),fill(khaki) )

Finally, we use **rect** for a set of nested rectangles.

```
(%i15) qdraw( xr(-3,3),yr(-3,3), rect( -2.5,-2.5,2.5,2.5,lw(4),lc(blue) ),
             rect( -2,-2,2,2,lw(4),lc(red) ),
             rect( -1.5,-1.5,1.5,1.5,lw(4),lc(green) ),
             rect( -1,-1,1,1,lw(4),lc(brown) ),
             rect( -.5,-.5,.5,.5,lw(4),lc(magenta) ),
             cut(all) )$
```

which produces:

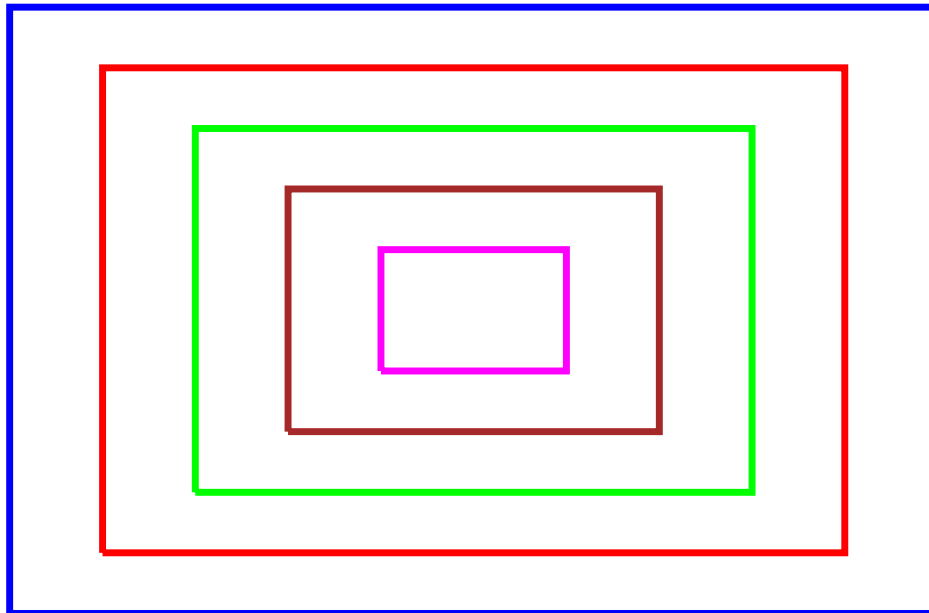


Figure 48: Nested Rectangles using rect(..)

### 5.13 Geometric Figures: poly(...)

The **qdraw** function **poly** has the syntax:

```
poly( pointlist, lc(c), lw(n), fill(c) )
```

in which "pointlist" has the same form as when used with **pts**:

```
[ [x1,y1], [x2,y2], ... [xn,yn] ] ,
```

and the arguments **lc**, **lw**, and **fill** are optional and can be in any order after pointlist. The last point in the list will be automatically connected to the first.

The default call to **poly** has color black, line width 3 and no fill color.

```
(%i16) qdraw( xr(-2,2),yr(-1,2),cut(all),
             poly([ [-1,-1],[1,-1], [2,2] ] ) )$
```

This default use of **poly** produces a "plain jane" triangle:

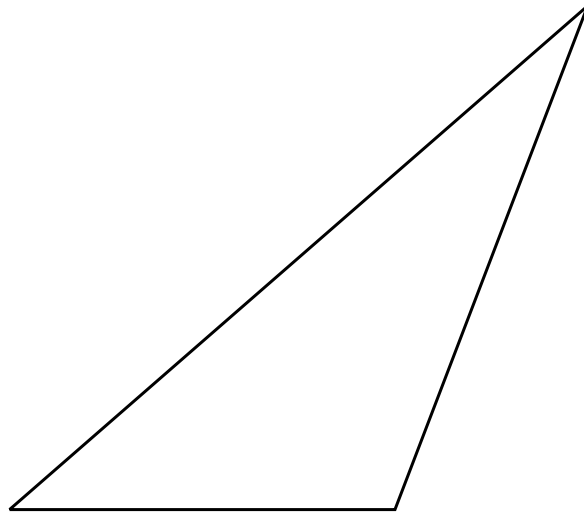


Figure 49: Default use of poly(...)

Next we create the work file "doplot2.mac" which contains the following Maxima function which will draw eighteen right triangles in various colors:

```
/* eighteen triangles */
disp("doplot2()")$
print("eighteen colored triangles")$
doplot2() :=
  block([cc, qlist, x1, x2, y1, y2, i, val ],
    cc : [aquamarine, beige, blue, brown, cyan, gold, goldenrod, green, khaki,
          magenta, orange, pink, plum, purple, red, salmon, skyblue, turquoise,
          violet, yellow ],
    qlist : [ xr(-3.3, 3.3), yr(-3.3, 3.3) ],
    /* top row of triangles */
    y1 : 1,
    y2 : 3,
    for i:0 thru 5 do (
      x1 : -3 + i,
      x2 : x1 + 1,
      val : poly( [ [x1, y1], [x2, y1], [x1, y2] ], fill( cc[i+1] ) ),
      qlist : append(qlist, [val] )
    ),
    /* middle row of triangles */
    y1 : -1,
    y2 : 1,
    for i:0 thru 5 do (
      x1 : -3 + i,
      x2 : x1 + 1,
      val : poly( [ [x1, y1], [x1, y2], [x2, y2] ], fill( cc[i+7] ) ),
      qlist : append(qlist, [val] )
    ),
  ),
```

```

/* bottom row of triangles */
y1 : -3,
y2 : -1,
for i:0 thru 5 do (
  x1 : -3 + i,
  x2 : x1 + 1,
  val : poly( [ [x1,y1],[x2,y1],[x1,y2]], fill( cc[i+13] ) ),
  qlist : append(qlist, [val ] )
),
qlist : append(qlist,[ cut(all) ] ),
apply( 'qdraw, qlist )
)$

```

Here is a record of use of this work file:

```

(%i17) load(doplot2);
                                doplot2()
eighteen colored triangles
(%o17)                                c:/work2/doplot2.mac
(%i18) doplot2();

```

and the resulting figure:

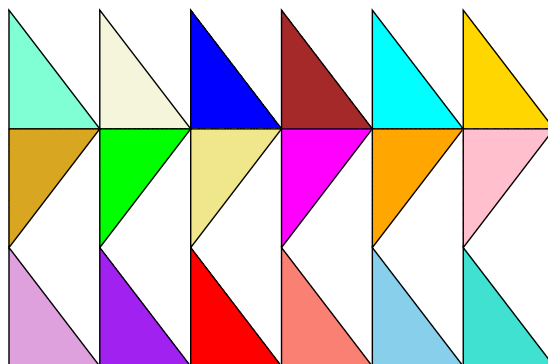


Figure 50: Using poly(...) with Color

For "homework", use **poly** and **pts** to draw the following figure. (Hint: you should also use **xr(...)** ).

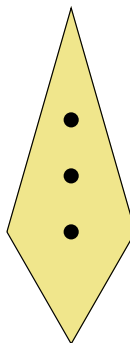


Figure 51: Homework Problem

## 5.14 Geometric Figures: circle(...) and ellipse(...)

The **qdraw** function **circle** has the syntax:

```
circle( xc,yc, r, lc(c), lw(n), fill(c) )
```

to draw a circle centered at  $(xc, yc)$  and having radius  $r$ . The last three arguments are optional and may be entered in any order after the required first three arguments. This object will not "look" like a circle unless you take care to make the horizontal extent of the "canvas" about 1.4 times the vertical extent (by using **xr(...)** and **yr(...)** ).

Here is the default circle in black, with line width 3, and no fill color.

```
(%i19) qdraw( xr(-2, 2), yr(-2, 2), circle( 0, 0, 1 ) )$
```

which looks like:

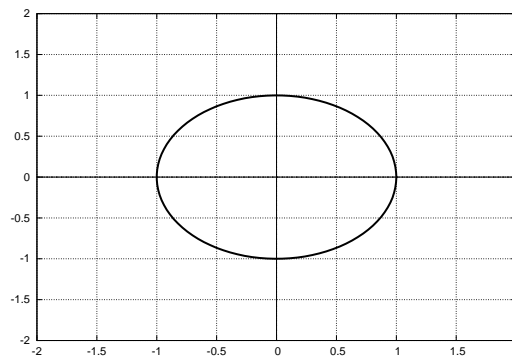


Figure 52: Default "circle"

By using **xr(...)** and **yr(...)** we try for a "round" circle and also add what should be a 45 degree line.

```
(%i20) qdraw(xr(-2.1,2.1),yr(-1.5,1.5),cut(all),  
             circle(0,0,1,lw(5),lc(brown),fill(khaki) ),  
             line(-1.5,-1.5,1.5,1.5,lw(8), lc(red) ),  
             key(bottom) )$
```

with the result:

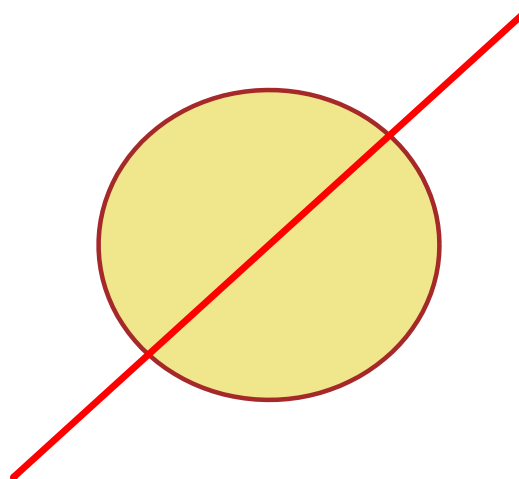


Figure 53: line over "round" circle

The line painted over the circle because of the order of the arguments to **qdraw**. If we reverse the order, drawing the line before the circle:

```
(%i21) qdraw(xr(-2.1,2.1),yr(-1.5,1.5),cut(all),
            line(-1.5,-1.5,1.5,1.5,lw(8),lc(red) ),
            circle(0,0,1,lw(8),lc(brown),fill(khaki) ),
            key(bottom) )$
```

then the circle color will hide the line:

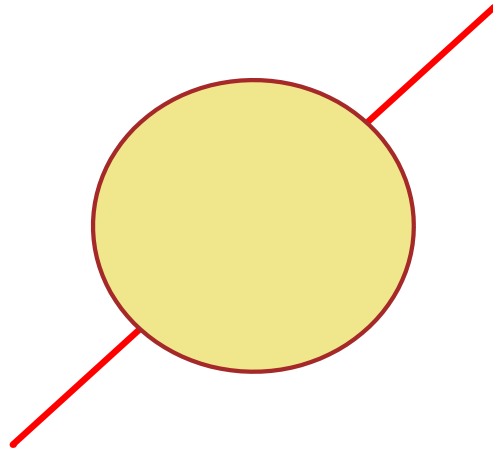


Figure 54: circle over line

The **qdraw** function **ellipse** has the syntax:

```
ellipse( xc,yc,xsma,ysma,th0deg,dthdeg, lw(n), lc(c), fill(c) )
```

which will plot a partial or whole ellipse centered at  $(xc, yc)$ , oriented with ellipse axes aligned along the  $x$  and  $y$  axes, having horizontal semi-axis  $xsma$ , vertical semi-axis  $ysma$ , beginning at  $th0deg$  degrees measured counter clockwise from the positive  $x$  axis, and drawn for  $dthdeg$  degrees in the counter clockwise direction.

The last three arguments are optional. The default is the outline of an ellipse for the specified angular range in color black, line width 3, and no fill color.

Here is the default behavior:

```
(%i22) qdraw( xr(-4.2,4.2),yr(-3,3),
            ellipse(0,0,3,2,90,270) )$
```

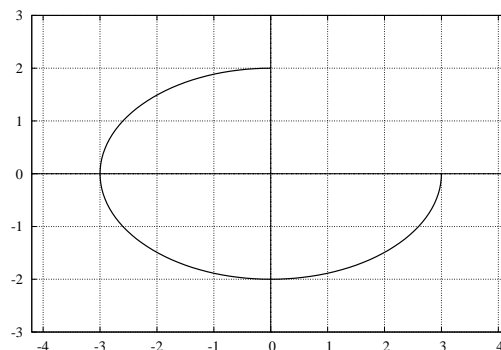


Figure 55: ellipse(0,0,3,2,90,270)

If we add color, fill, and some curvy background, as in

```
(%i23) qdraw( xr(-5.6,5.6),yr(-4,4),exl(x,x,-4,4,lc(blue),lw(5)),
              exl(4*cos(x),x,-4,4,lc(red),lw(5)),
              ellipse(0,0,3,1,90,270,lc(brown),lw(5),fill(khaki)),
              cut(all))$
```

we get

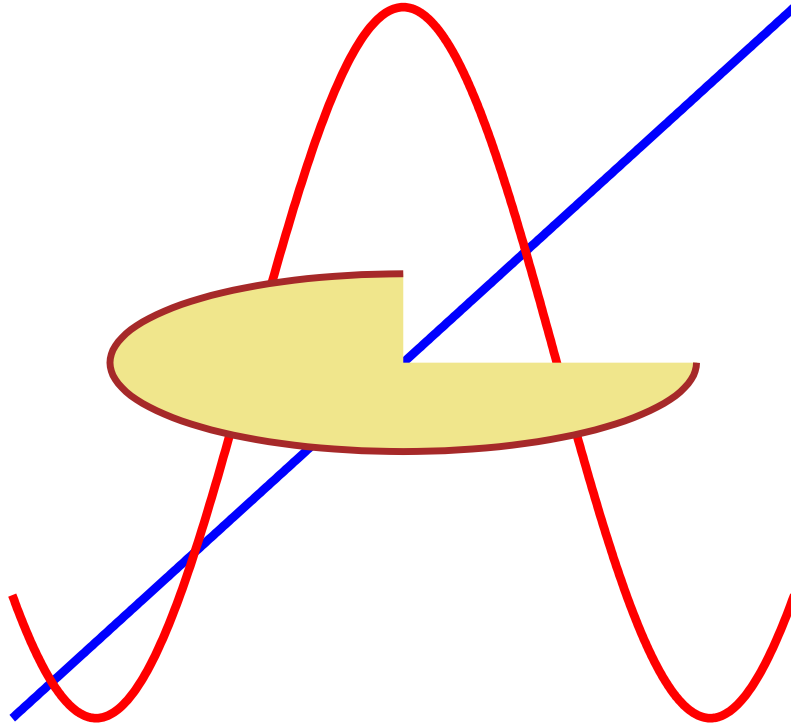


Figure 56: Filled Ellipse

## 5.15 Geometric Figures: **vector(..)**

The **qdraw** function **vector** has the syntax:

```
vector([x,y],[dx,dy],ha(thdeg),hb(v),hl(v),ht(t),lw(n),lc(c),lk(string))
```

which draws a vector with components [dx,dy] starting at [x,y].

The first two list arguments are required, all others are optional and can be entered in any order after the first two required arguments.

The default "head angle" is 30 deg; change to 45 deg using ha(45) for example.

The default "head both" value is f for false; use hb(t) to set it to true, and hb(f) to return to false.

The default "head length" is 0.5; use hl(0.7) to change to 0.7.

The default "head type" is "nofilled"; use ht(e) for "empty", ht(f) for "filled", and ht(n) to change back to "nofilled".

Once one of the "head properties" has been changed in a call to **vector**, the next calls to **vector** assume the change is still in force.

The default line width is 3; use lw(5) to change to 5.

The default line color is black; use lc(brown) to change to brown.

The default is no key string; use lk("A1"), for example, to create a text string for the key.



Here is a use of **vector** which accepts all defaults:

```
(%i24) qdraw( xr(-2,2), yr(-2,2), vector( [-1,-1], [2,2] ) )$
```

with the result:

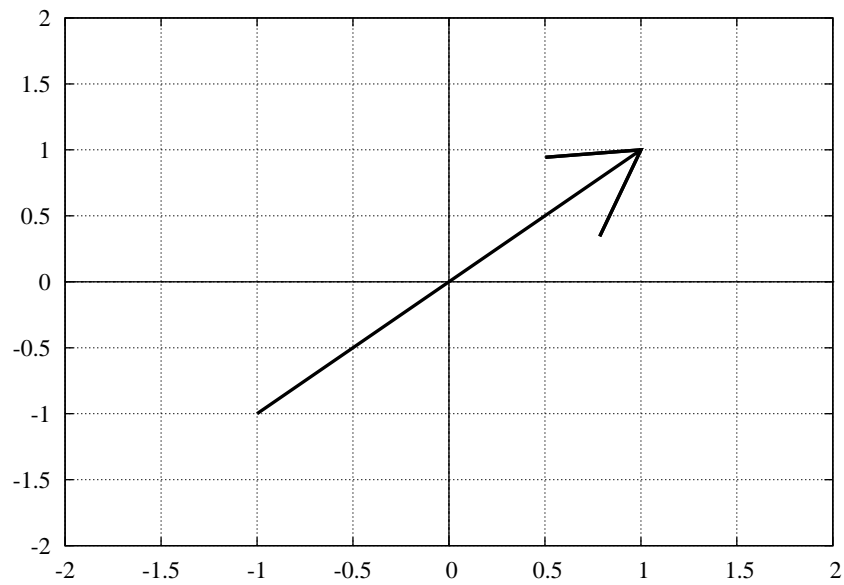


Figure 57: Default Vector

We can thicken and apply color to this basic vector with

```
(%i25) qdraw(xr(-2,2), yr(-2,2),  
             vector([-1,-1], [2,2], lw(5), lc(brown), lk("vec 1")),  
             key(bottom) )$
```

which looks like:

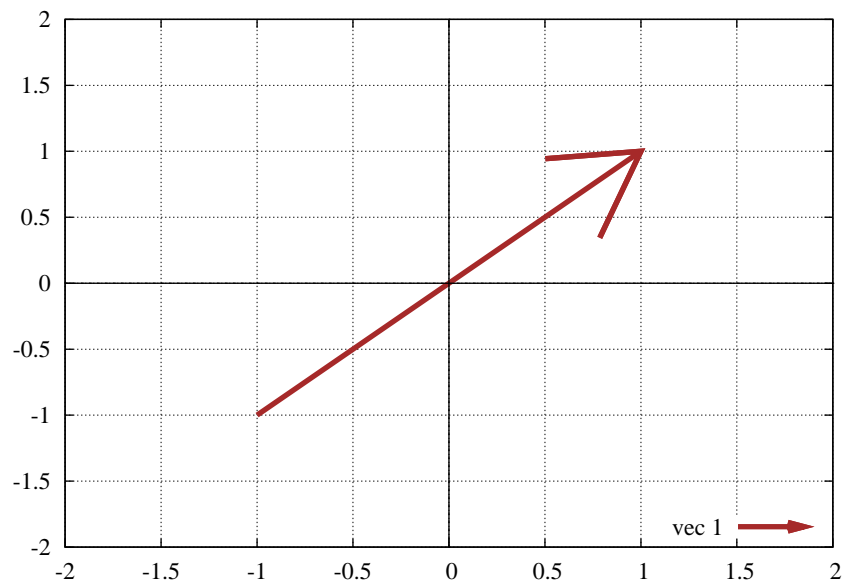


Figure 58: Adding Color, etc.

Next we can alter the "head" properties, but let's also make this vector shorter. We use `ht(e)` to set `head_type` to "empty", `hb(t)` to set `head_both` to "true", and `ha(45)` to set `head_angle` to 45 degrees.

```
(%i26) qdraw(xr(-2,2),yr(-2,2),
            vector([0,0],[1,1],lw(5),lc(blue),lk("vec 1"),
                ht(e),hb(t),ha(45) ), key(bottom) )$
```

which produces:

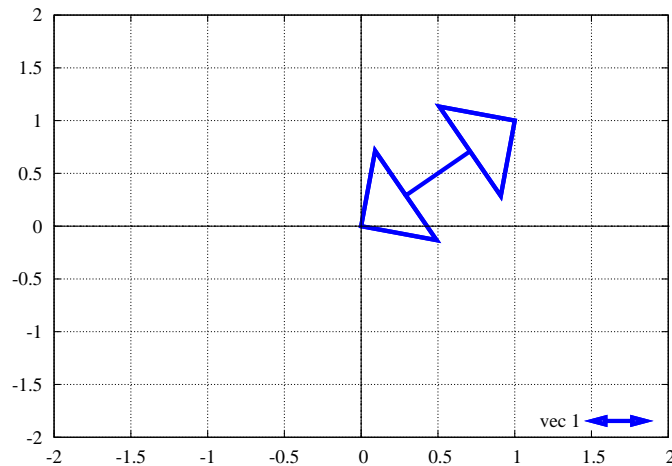


Figure 59: Changing Head Properties

Once you invoke the head properties options, the new settings are used on your next calls to **vector** (unless you deliberately change them). Here is an example of that memory feature at work:

```
(%i27) qdraw(xr(-2.8,2.8),yr(-2,2),
            vector([0,0],[1,1],lw(5),lc(blue),lk("vec 1"),
                ht(e),hb(t),ha(45) ),
            vector([0,0],[-1,-1],lw(5),lc(red),lk("vec 2"),
                key(bottom) ) )$
```

and we also used the x-range setting to get the geometry closer to reality, with the result:

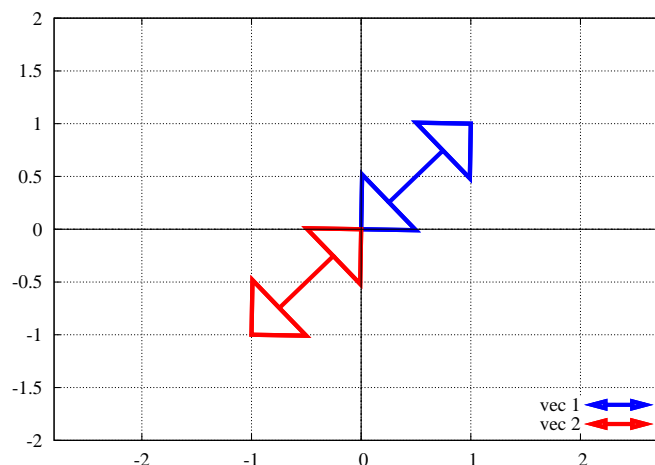


Figure 60: Head Properties Memory at Work

## 5.16 Geometric Figures: arrowhead(..)

The syntax of the **qdraw** function **arrowhead** is:

```
arrowhead( x, y, theta-degrees, s, lc(c), lw(n) )
```

which will draw an arrow head with the vertex at (x,y).

The first four arguments are required and must be numbers.

The third argument theta is an angle in degrees and is the direction the arrowhead is to point relative to the positive x axis, ccw from x axis taken as a positive angle.

The fourth argument s is the length of the sides of the arrowhead.

The arguments lc(c) and lw(n) are optional, and are used to alter the default color (black) and line width (3).

The opening half angle is hardwired to be  $\phi = 25 \text{ deg} = 0.44 \text{ radians}$ .

The geometry will look better if the x-range is about 1.4 times the y-range.

Here are four arrow heads drawn with the default line widths and color and "size" 0.3, which show the use of the direction argument in degrees.

```
(%i28) qdraw(xr(-2.8,2.8),yr(-2,2),  
            arrowhead(1.5,0,180,.3),arrowhead(0,1,270,.3),  
            arrowhead(-1.5,0,0,.3),arrowhead(0,-1,90,.3) )$
```

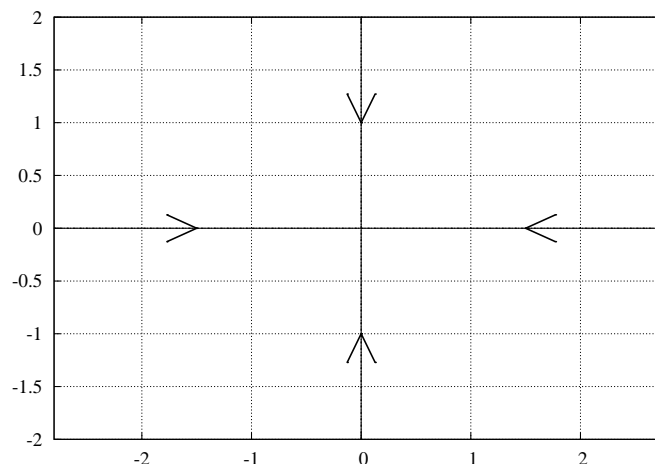


Figure 61: Using arrowhead(..)

## 5.17 Labels with Greek Letters

### 5.17.1 Enhanced Postscript Methods

Here we combine **line**(..), **ellipse**, **arrowhead** and **label**, using the enhanced postscript abilities of **draw2d**'s `terminal = eps` option, which became effective with the June 11, 2008 draw package update. This update includes an extension of the `terminal = eps` abilities to include local conversion of font properties and the use of Greek and some math characters. As of the writing of this section, it was necessary to download this update from the webpage

<http://maxima.cvs.sourceforge.net/maxima/maxima/share/draw/draw.lisp>. On that webpage you will see Log of /maxima/share/draw/draw.lisp, and the top entry is Revision 1.31 - (view) (download) (annotate) - [select for diffs] Wed Jun 11 18:19:55 2008 UTC. Click on the "download" link, and the text of draw.lisp will appear in your browser with the top looking like:

```

;;;                                COPYRIGHT NOTICE
;;;
;;;  Copyright (C) 2007 Mario Rodriguez Riotorto
;;;
;;;  This program is free software; you can redistribute
;;;  it and/or modify it under the terms of the
;;;  ..... etc

```

This is a program written in the Lisp language, and down near the bottom is a series of lines which provide for the enhanced postscript behavior:

```

($eps (format cmdstorage "set terminal postscript eps enhanced ~a
                        size ~acm, ~acm~%set out '~a.eps'"
                        (write-font-type) ; other alternatives are Arial, Courier
                        (get-option '$eps_width)
                        (get-option '$eps_height)
                        (get-option '$file_name)))

```

If you save this text file with the name "draw.lisp" and place it in the draw package folder (on my Windows computer, the rather complicated path:

drive c, program files, maxima-5.15.0, share, maxima, 5.15.0, share, draw ) to replace the old "draw.lisp" (which you could rename prior to the save as ), then you can use the label syntax inside strings as shown in the following.

```

(%i29) qdraw(xr(0,2.8),yr(0,2),
            line(0,0,2.8,0,lw(2)),
            line(0,0,2,2,lc(blue),lw(8) ),
            ellipse(0,0,1,1,0,45 ),
            arrowhead(0.707,0.707,135,0.15),
            label(["{/=36 {/Symbol q  \254 }  The Incline Angle}",1,0.4]),
            cut(all),
            pic(eps,"ch5p52" ) );

```

The result looks like:

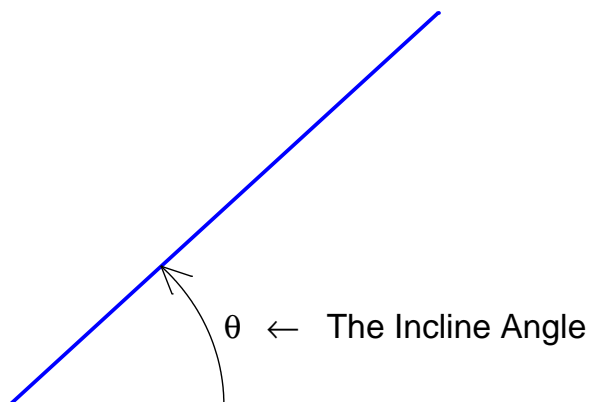


Figure 62: line(..), ellipse(..), arrowhead(..), label(..)

A summary of the enhanced postscript syntax can be downloaded:

[http://www.telefonica.net/web2/biomates/maxima/gpdraw/ps/ps\\_guide.ps](http://www.telefonica.net/web2/biomates/maxima/gpdraw/ps/ps_guide.ps) although the examples of using the "characters" works for me only if I use two back slashes, as in the example just shown,

where the entry `\254` inside the `{/Symbol }` structure produces the leftward pointing arrow (thanks to the draw package developer, Mario Rodriguez Riotorto, for the enhanced postscript abilities, and for aiding my understanding of how to use these features). An example of creating text for a label which includes the integral sign and Greek letters is given on the webpage:

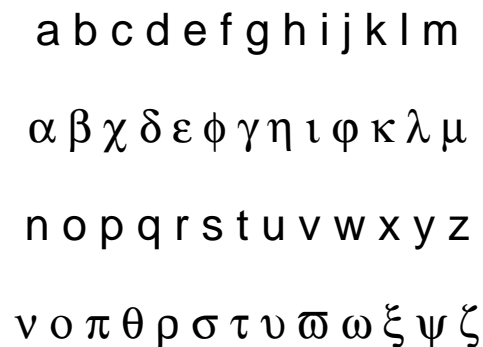
<http://www.telefonica.net/web2/biomates/maxima/gpdraw/ps/>.

The entry `{/Symbol q }` by itself would produce just the Greek letter  $\theta$ . Wrapping the text entry in the structure `{/=36 }` accepts the default font type and sets the font size to 36 for the text inside the pair of braces.

Here we make a lower case Latin to Greek conversion reminder using four instances of **label**, although we could alternatively have used the syntax `label( [s1,x1,x2], [s2,x2,y2], ... )`.

```
(%i30) qdraw(xr(-3,3),yr(-2,2),label_align(c),
  label( ["{/=48 a b c d e f g h i j k l m}",0,1.5] ),
  label( ["{/Symbol=48 a b c d e f g h i j k l m}",0,0.5] ),
  label( ["{/=48 n o p q r s t u v w x y z",0,-.5] ),
  label( ["{/Symbol=48 n o p q r s t u v w x y z",0,-1.5] ),
  cut(all), pic(eps,"ch5p53") )$
```

Note how we increase the font size of the latin alphabet `a, b, c, ...`. Here is the resulting eps figure:



a b c d e f g h i j k l m

$\alpha \beta \chi \delta \epsilon \phi \gamma \eta \iota \varphi \kappa \lambda \mu$

n o p q r s t u v w x y z

$\nu \omicron \pi \theta \rho \sigma \tau \upsilon \wp \omega \xi \psi \zeta$

Figure 63: Lower Case Latin to Greek

We can repeat that label figure using upper case Latin letters:

```
(%i31) qdraw(xr(-3,3),yr(-2,2),label_align(c),
  label( ["{/=48 A B C D E F G H I J K L M}",0,1.5] ),
  label( ["{/Symbol=48 A B C D E F G H I J K L M}",0,0.5] ),
  label( ["{/=48 N O P Q R S T U V W X Y Z",0,-.5] ),
  label( ["{/Symbol=48 N O P Q R S T U V W X Y Z",0,-1.5] ),
  cut(all), pic(eps,"ch5p54") )$
```

You can see the resulting figure on the next page.

Useful character codes, used as `{/Symbol \abc \rst etc }` or as `{/Symbol=36 \abc \rst etc }` are:

- `\243` (less than or equal)
- `\245` (infinity symbol)
- `\253` (double ended arrow)
- `\254` (left arrow)
- `\256` (right arrow)

$\pm$  ( $\backslash\backslash 261$ ) (plus or minus)  
 $\geq$  ( $\backslash\backslash 263$ ) (greater than or equal)  
 $\times$  ( $\backslash\backslash 264$ ) (times)  
 $\neq$  ( $\backslash\backslash 271$ ) (not equal)  
 $\approx$  ( $\backslash\backslash 273$ ) (approx equal)  
 $\sum$  ( $\backslash\backslash 345$ ) (summation sign)  
 $\int$  ( $\backslash\backslash 362$ ) (integral sign)

A B C D E F G H I J K L M  
 A B X  $\Delta$  E  $\Phi$   $\Gamma$  H I  $\vartheta$  K  $\Lambda$  M  
 N O P Q R S T U V W X Y Z  
 N O  $\Pi$   $\Theta$  P  $\Sigma$  T Y  $\varsigma$   $\Omega$   $\Xi$   $\Psi$  Z

Figure 64: Upper Case Latin to Greek

Here we use **label** to illustrate these possible symbols you can use:

```

(%i32) s1 : "{/Symbol=48 \243 \245 \253 \254 \256}"$
(%i33) s2 : "{/Symbol=48 \261 \263 \264 \271 \273 \345 \362}"$
(%i34) qdraw(xr(-3,3),yr(-2,2),label_align(c),
           label( [s1,0,1] ), label( [s2,0,-1] ), cut(all), pic(eps,"ch5p55") )$
  
```

which produces the figure:

$\leq$   $\infty$   $\leftrightarrow$   $\leftarrow$   $\rightarrow$   
 $\pm$   $\geq$   $\times$   $\neq$   $\approx$   $\sum$   $\int$

Figure 65: Useful Character Code Symbols

### 5.17.2 Windows Fonts Methods with jpeg Files

We can produce the Greek letter  $\theta$  in a jpeg file or a png file by using the Greek font files in the `c:\windows\fonts` folder as follows:

```
(%i35) qdraw(xr(0,2.8),yr(0,2),
            line(0,0,2.8,0),
            line(0,0,2,2,lc(blue),lw(5) ),
            ellipse(0,0,1,1,0,45 ),
            arrowhead(0.707,0.707,135,0.15),
            label(["q",1,0.4]), cut(all),
            pic(jpg,"ch5p52p",font("c:/windows/fonts/grii.ttf",36)) );
```

which produces the file `ch5p52.jpg`, which we converted to an eps file using cygwin's convert function:  
`convert name.jpg name.eps` which will also work to convert a png graphics file.

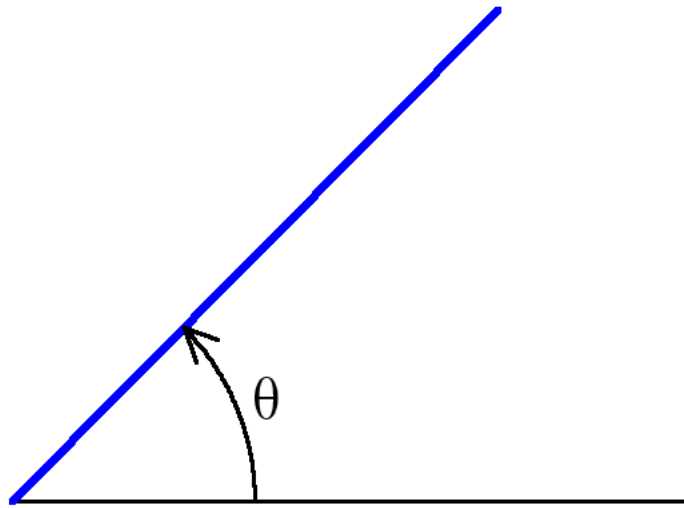


Figure 66: Greek in jpeg converted to eps

Here we use the label function to show all the greek letters available via the windows, fonts folder:

```
(%i36) qdraw(xr(-3,3),yr(-2,2),
            label(["a b c d e f g h i j k l m",-2.5,1.5],
                ["n o p q r s t u v w x y z",-2.5,0.5],
                ["A B C D E F G H I J K L M",-2.5,-0.5],
                ["N O P Q R S T U V W X Y Z",-2.5,-1.5] ),
            cut(all),
            pic(png,"ch5p60",font("c:/windows/fonts/grii.ttf",24)) );
```

After conversion of the png to an eps graphics file using Cygwin's convert function, we get:

$\alpha \beta \chi \delta \epsilon \phi \gamma \eta \iota' \kappa \lambda \mu$   
 $\nu \omicron \pi \theta \rho \sigma \tau \upsilon' \omega \xi \psi \zeta$   
 $A B X \Delta E \Phi \Gamma H \Gamma' K \Lambda M$   
 $N O \Pi \Theta P \Sigma T \Upsilon' \Omega \Xi \Psi Z$

Figure 67: windows fonts conv. of a - Z to Greek

### 5.17.3 Using Windows Fonts with the Gnuplot Console Window

In the default Windows Gnuplot console mode, you can convert some Latin letters to Greek as follows:

```
(%i37) qdraw(xr(0,2.8),yr(0,2),
            line(0,0,2.8,0),
            line(0,0,2,2,lc(blue),lw(5)),
            ellipse(0,0,1,1,0,45),
            arrowhead(0.707,0.707,135,0.15),
            label(["q",1,0.4]), cut(all));
```

When the console graphics window appears, right click on the upper left corner icon and select Options, Choose Font. In the Font panels, choose Graecall font, "regular" from the middle panel, and size 36 from the right panel and click "ok". The English letter "q" (lower case) is then converted to the Greek lower case theta. Use the Gnuplot window menu again to save the resulting image to the clipboard, and open an image viewer. I use the freely available Infanview. If you use View, Paste, the clipboard image appears inside Infanview, and you can save the image as a jpeg file in your choice of folder. Since I am using eps graphics files for this latex file, I converted the jpg to eps using Cygwin's convert function:

```
convert name.jpg name.eps.
```

Here is the result:

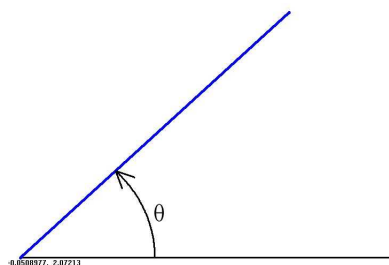


Figure 68: Greek via Windows Clipboard



Unfortunately, saving the Gnuplot window image to the Window's clipboard also saves the current cursor position, which is not desirable.

## 5.18 Even More with more(...)

You can use the **qdraw** function **more(...)**, containing any legal **draw2d** elements, as we illustrate by adding a label to the x-axis and a title. We focus here on producing an eps graphics file to display the enhanced ability to show subscripts and superscripts.

```
(%i38) qdraw( lw(8), ex([x,x^2,x^3],x,-2,2),
            more(xlabel = "X AXIS", title="intersections of x, x^2, x^3" ),
            cut(key), line(-2,0,2,0,lw(2)), line(0,-8,0,8,lw(2)),
            vector([-1,5],[-0.4,-2.7],lc(red),hl(0.1) ),
            label(["x^2",-0.9,6]),
            vector([-1.2,-6],[-0.5,0],lc(turquoise),lw(8)),
            label(["x^3",-1,-5.5] ),
            pts( [[-1,-1],[0,0],[1,1]],ps(2),pc(magenta) ),
            pic(eps,"ch5p56",font("Times-Roman",28)) )$
```

The lines for the x and y axes need special emphasis to show up clearly with an eps file, so we have used **qdraw**'s **line** function for that task. We also need to increase the line width setting for the eps file case, which we have done with the **qdraw** top level function **lw**, which only affects the "quick plotting" functions **ex** and **imp**. We have used **qdraw**'s **more** function to provide an x-axis label and a title. The font setting in the **pic** function supplies an overall drawing font type and size which affects all elements unless locally over-ridden with the special enhanced postscript features. In the title and labels,  $x^n$  is converted automatically to  $x^n$ .

Here is the resulting plot:

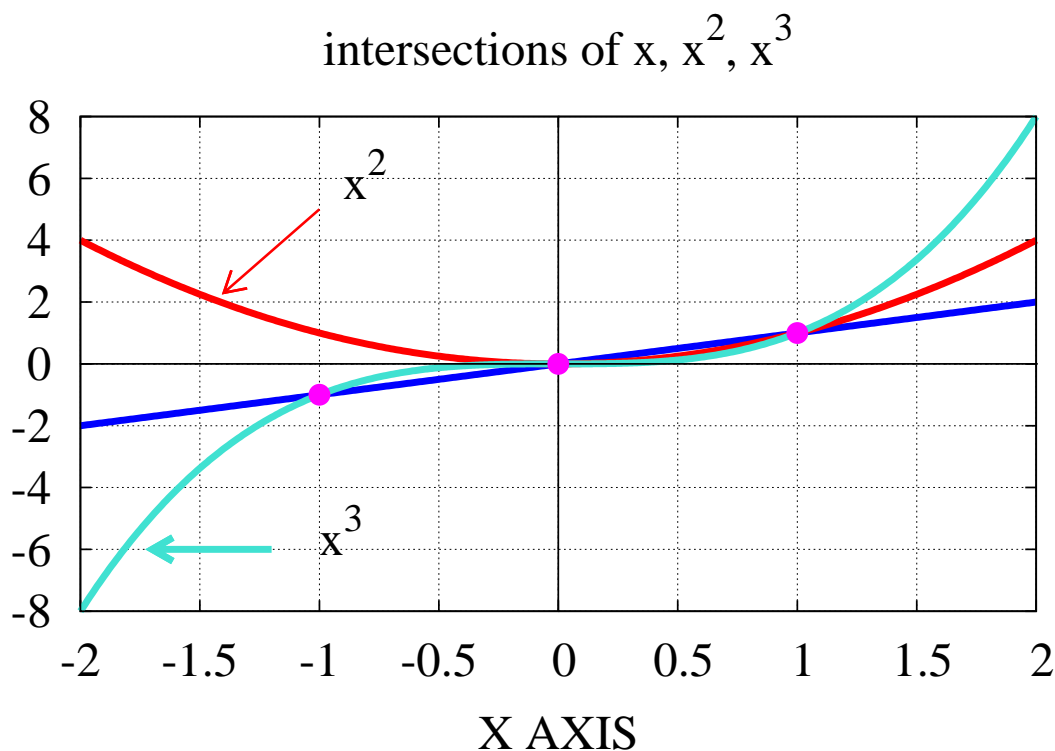


Figure 69: Using more(...) for Title and X Axis Label

## 5.19 Programming Homework Exercises

### 5.19.1 General Comments

The file `qdraw.mac` is a text file which you can modify with a good text editor such as `notepad2`. This Maxima code is heavily commented as an aid to passing on some Maxima language programming examples. You can get some experience with the Maxima programming language elements by copying the file `qdraw.mac` to another name, say `myqdraw.mac`, and use that copy to make modifications to the code which might interest you. By frequently loading in the modified file with `load(myqdraw)`, you can let Maxima check for syntax errors, which it does immediately.

The most common syntax errors involve parentheses and commas, with strange error messages such as "BLANK IS NOT AN INFIX OPERATOR", or "TOO MANY PARENTHESES", etc. Placing a comma just before a closing parenthesis is a fatal error which can nevertheless creep in. This sounds obvious, but you may find it useful to insert some special debug printouts, such as `print("in blank, a = ", a)` or `display(a)`, perhaps at the end of a `do` loop, so you are working with the structure:

```
for i thru n do (  
    job1,  
    job2,  
    job3,  
print(" i = ", i, " blank = ", blank)  
/* end do loop */  
) ,  
...program continues...
```

When you are finished debugging a section, you either will comment out the debug printout or simply delete it to clean up the code. If you are not fully awake, you might then load into Maxima

```
for i thru n do (  
    job1,  
    job2,  
    job3,  
/* end do loop */  
) ,  
...program continues...
```

and, of course, Maxima will object, since that extra comma no longer makes sense.

It is crucial to use a good text editor which will "balance" parentheses, brackets, and braces to minimize parentheses etc errors.

If you look at the general structure of **qdraw**, you will see that most of the real work is done by **qdraw1**. If you call **qdraw1** instead of **qdraw**, you will be presented with a rather long list of elements which are understood by **draw2d**. Even if you use **qdraw**, you will see the same long list wrapped by "draw2d" if you have not loaded the **draw** package.

One feature you should look at is how a function which takes an arbitrary number of arguments, depending on the user (as does the function **draw2d**), is defined. If this seems strange to you, experiment with a toy function having a variable number of arguments, and use printouts inside the function to see what Maxima is doing.

### 5.19.2 XMaxima Tips

It is useful to first try out a small code idea directly in XMaxima, even if the code is ten or fifteen lines long, since the XMaxima interface has been greatly improved. When you want to edit your previous "try", use `Alt-p` to enter your previous code, and immediately backspace over the final `);` or `);`. You can then cursor up to an area where you want to add a new line of code, and with the cursor placed just after a comma, press `ENTER` to create a new (blank) line. Since the block of code has not been properly concluded with either a `);` or `);`, Maxima will not try to "run" with the version you are working on when you press `ENTER`. Once you have made the changes you want, cursor your way to the end and put back the correct ending and then pressing `ENTER` will send your code to the Maxima engine.

The use of `HOME`, `END`, `PAGEUP`, `PAGEDOWN`, `CNTRL-HOME`, and `CNTRL-END` greatly speeds up working with XMaxima. For example to copy a code entry up near the top of your current workspace, first enter `HOME` to put the cursor at the beginning of the current line, then `PAGEUP` or `CNTRL-HOME` to get up to the top fast, then drag over the code (don't include the `(%i5)` part) to the end but not to the concluding `);` or `);`. You can hold down the `SHIFT` key and use the right (and left) cursor key to help you select a region to copy. Then press `CNTRL-C` to copy the selected code to Window's clipboard. Then press `CTRL-END` to have the cursor move to the bottom of your workspace where XMaxima is waiting for your next input. Press `CNTRL-V` to paste your selection. If the selection extends over multiple lines, use the down cursor key to find the end of the selection which should be without the proper code ending `);` or `);`. You are then in the driver's seat and can cursor your way around the code and make any changes without danger of XMaxima pre-emptively sending your work to the computing engine until you go to that end and provide the proper ending.

### 5.19.3 Suggested Projects

You will have noticed that we used the **qdraw** function **more** in order to insert axis labels and a title into our plot. Design **qdraw** functions `xlabel(string)`, `ylabel(string)`, and `title(string)`. Place them in the "scan3" section of **qdraw** and try them out. You will need to pay attention to how new elements get passed to **draw2d**. In particular, look at the list `drlist`, using your text editor search function (in `notepad2`, `Ctrl-f`) to see how that list is constructed based on the user input.

A second small project would be to add a "line type" option for the **qdraw** function **line**. My experience is that setting `line_type = dots` in **draw2d** produces no immediate change in the Windows Gnuplot console window, produces a finely dotted line for jpeg and png image files, and produces a dashed line with eps image files. Your addition to **qdraw** should follow the present style, so the user would use the syntax `line(x1,y1,x2,y2,lc(c),lw(n),lk(string),lt(type))`, where `type` is either `s` or `d` (for solid or dots).

A third small project would be to design a function **triangle** for **qdraw**, including the options which are presently in **poly**.

A fourth small project would be to include the option `cbox(..)` in the **qdensity** function. The present default is to include the colorbox key next to the density plot, but if the user entered `qdensity(....,cbox(f))`, the colorbox would be removed.

A more challenging project would be to write a **qdraw** function which would directly access the creation of bar charts. These notes are written with the needs of the typical physical science or engineering user in mind, so no attention has been paid to bar charts here. Naturally, if you frequently construct bar charts, this project would be interesting for you. Start this project by first working with **draw2d** directly, to get familiar with what is already available, and to avoid "re-creating the wheel".

One general principle to keep in mind is that the Maxima language is an "interpretive language"; Maxima does not make multiple passes over your code to reconcile function references, such as a compiler does. This means that if a part of your code "calls" some user defined function, Maxima needs to have already read about that function definition in your code.

## 5.20 Acknowledgements

The author would like to thank Mario Rodriguez Riotorto, the creator of Maxima's **draw** graphics interface to Gnuplot, for his encouragement and advice at crucial stages in the development of the **qdraw** interface to **draw2d**. The serious graphics user should spend time with the many powerful features of the **draw** package, and the examples provided on the **draw** webpages,

<http://www.telefonica.net/web2/biomates/maxima/gpdraw/>.

These examples go far beyond the simple graphics in this chapter. The recent updating of the **draw** package to allow use of Gnuplot's enhanced postscript features makes Maxima a more attractive tool for the creation of educational diagrams.