Maxima by Example:
Ch.13: 2D Plots and Graphics using qdraw and wxqdraw *

Edwin L. (Ted) Woollett

April 16, 2018

Contents

*This version uses Maxima 5.36.1 for Windows. 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`

This document is Ch. 13 of the series "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.

Supplementary files available in the Ch. 13 section are
qdraw.mac, wxqdraw.wxm, qdrawcode.txt, faithful.dat, fit1.dat, and fit2.dat.

1 2D Plots and Graphics using the qdraw package

Chapter 13 provides an introduction to a graphics interface to the **draw** package `...share\draw\draw.lisp`. Using the **xMaxima** interface, just use `load(qdraw);` or `load("c:/work5/qdraw.mac");`, etc to load the qdraw file. If you are using the **wxMaxima** interface, you can either use the same command, or else use the menus at the top, **File, Open...** and select the file **qdraw.mac**.

If you plan to exclusively use the **qdraw** syntax, (as you must using **xMaxima**, or as you can in **wxMaxima**, then you must separately load the draw package, for example useing: `load(draw);`. If you are using **wxMaxima**, and at least your first plot command will use the syntax **wxqdraw**, then the draw package loads automatically.

```
(%i1) load("C:/work5/qdraw.mac")$
```

qdraw.mac: see Maxima by Example, Ch. 13

qdraw(...), wxqdraw(...), qdensity(...), wxqdensity(...)

for syntax info, type: qdraw();

Using the **xMaxima** interface, you always need to load both the **draw** and **qdraw** packages separately (and in any order).

```
Maxima 5.36.1 http://maxima.sourceforge.net
using Lisp SBCL 1.2.7
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
(%i1) load(draw);
;; loading #P"C:/Documents and Settings/Edwin Woollett/maxima/binary/5_36_1/sbcl/1_2_7/share/draw/grcommon.fasl"
;; loading #P"C:/Documents and Settings/Edwin Woollett/maxima/binary/5_36_1/sbcl/1_2_7/share/draw/gnuplot.fasl"
;; loading #P"C:/Documents and Settings/Edwin Woollett/maxima/binary/5_36_1/sbcl/1_2_7/share/draw/vtk.fasl"
;; loading #P"C:/Documents and Settings/Edwin Woollett/maxima/binary/5_36_1/sbcl/1_2_7/share/draw/picture.fasl"
(%o1)
   C:/Program Files/Maxima-sbcl-5.36.1/share/maxima/5.36.1/share/draw/draw.lisp
(%i2) load(qdraw);
 qdraw.mac: see Maxima by Example, Ch. 13
 qdraw(...), wxqdraw(...), qdensity(...), wxqdensity(...)
  for  syntax info,  type:  qdraw();
(%o2)                         c:/work5/qdraw.mac
```

The examples we show here using `qdraw(...)` or `qdensity(...)` using the **xMaxima** interface can also be used with the **wxMaxima** interface. If you replace **qdraw** with **wxqdraw**, and replace **qdensity** with **wxdensity** (inside the **wxMaxima** interface), and if you have **display2d** set to **true**, then you will generate inline plots in your `*.wxm` worksheet. You can open the file **wxqdraw.wxm** and execute selected cells to see a variety of graphics examples from those we discuss here. (Remember that you must have **display2d** set to **true** to see inline plots, and you must execute the cell which loads (opens) our **qdraw.mac** package of routines.

Our function **qdraw** calls **draw2d** and our function **wxqdraw** calls **wxdraw2d**. To save space and editing effort, we normally show here the **xMaxima** interface input and output (with **display2d** set equal to **false**), using the **qdraw** syntax.

2 qdraw or wxqdraw Syntax Summary

All arguments to **`qdraw`** (or **`wxqdraw`**) are optional and can be entered in any order.

You can have no more than one **`xr(..)`** argument. Likewise, no more than one **`yr(..)`**, one **`cut(..)`**, one **`lw(n)`** (as an arg of **`qdraw`**), one **`nticks(n)`** and one **`ipgrid(n)`**.

You can have an arbitrary number of the other args in any order.

The complete set of possible arguments (in alphabetic order) with the maximum number and type of arguments follow. In general, arguments with names **`lc, lw, lk, fill, pc, ps, pt, pk, pj, ha, hb, hl`**,and **`ht`** are optional.

```
qdraw(
     arrowhead(x,y,theta-degrees,s,lc(c),lw(n) ),
     circle(x,y,radius,lc(c),lw(n),fill(cc) ),
     contour(expr,x,x1,x2,y,y1,y2,crange(n,min,max),options )
              or contour(expr,x,x1,x2,y,y1,y2,cvals(v1,v2,..),options),
              contour options are lc(c),lw(n), add( add-options );
                      add-options are grid,xaxis,yaxis,and xyaxes,
     cut(cut-options);
              cut-options are key,grid,xaxis,yaxis,xyaxes,edge,all,
     ellipse(xc,yc,xsma,ysma,th0-deg,dth-deg,lw(n),lc(c),fill(cc) ),
     errorbars(ptlist,dylist,lc(c),lw(n) ),
     ex(exprlist,x,x1,x2),
     ex1(expr,x,x1,x2,lc(c),lw(n),lk(string) ),
     imp(eqnlist,x,xx1,xx2,y,yy1,yy2),
     imp1(eqn,x,x1,x2,y,y1,y2,lc(c),lw(n),lk(string) ),
     ipgrid(n),
     key(bottom)
              or key(top),
     label( [string1,x1,y1],[string2,x2,y2],...),
     label_align(p-options); p-options are l, r, or c,
     line(x1,y1,x2,y2,lc(c),lw(n),lk(string) ),
     log(log-options);
               log-options are x, y, or xy,
     lw(n),
     more( any legal draw2d arguments),
     nticks(n),
     para( xofu,yofu,u,u1,u2,lc(c),lw(n),lk(string) ),
     polar( expr,theta,th1,th2,lc(c),lw(n),lk(string) );
          expr depends on variable theta, and limits th1 and th2 must be in radians,
     poly([ [x1,y1],[x2,y2],.,[xN,yN] ], lc(c),lw(n),fill(cc) ),
     pts( [ [x1,y1],[x2,y2],.,[xN,yN] ],pc(c),ps(s),pt(t),pk(string) ),
     pic( type, fname(string)); type is either eps or eps_color
     rect( x1,y1,x2,y2, lc(c),lw(n),fill(cc) ),
     vector([x,y],[dx,dy],lw(n),lc(c),lk(string),ha(deg),hb(v),hl(v),ht(t)),
                    type vector_use(); to see vector option details,
     xr(xa,xb),
     yr(ya,yb)  )
```

3 Quick Plots for Explicit Functions: ex(...) and ex1(...)

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** for plotting explicit functions: **ex(...)**, **ex1(...)**. Both of these functions call the `draw2d` function `explicit`.

The simplest plot of one or more expressions uses the `qdraw` arg `ex(expr,x,x1,x2)`, for example for one expression, `ex(x^3/5,x,0,2)`, or `ex(exp(u),u,-2,5)`. For the simultaneous plot of two expressions, use a **list** for the first arg of `ex`, such as `ex([x,x^2],x,-2,3)` or `ex([v,v^2],v,-2,3)` (which will both produce the same plot).

```
(%i3) qdraw ( ex (cos(x), x, 0, 6))$
```

which produces the "plane jane" plot

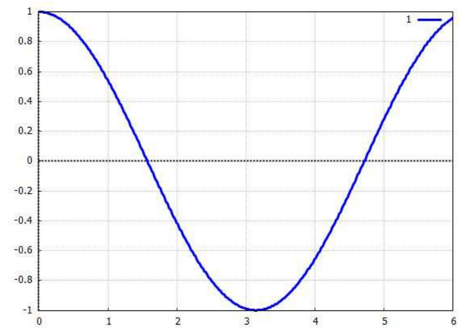

Figure 1: $\cos(x)$ using ex(...)

The `ex(...)` method does not allow you to control the color, as seen here:

```
(%i4) qdraw ( ex (cos(x), x, 0, 6, lc(red)))$
...syntax error
ex() should have exactly four arguments
```

You get more control options if you use `ex1(...)`, but this second method can only be used for one expression; if you want to simultaneously plot several expressions using the `ex1` method, you must include several separate `ex1(...)` invocations inside your `qdraw` wrapper. Sticking to our single expression for now, `lc(red)` stands for "line color red",

```
(%i5) qdraw ( ex1 (cos(x), x, 0, 6, lc(red)))$
```

which produces

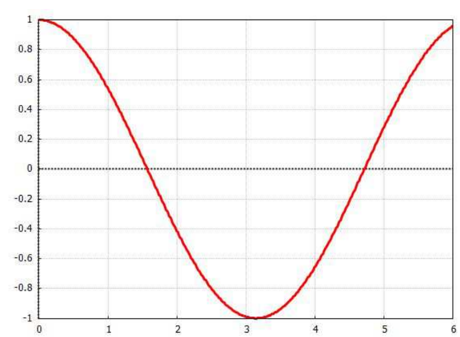

Figure 2: $\cos(x)$ using ex1(...)

We can add labels to the x and y axes, and add a title using the optional arg **more** which can contain any extra legal draw2d assignments:

```
(%i6) qdraw ( ex1 (cos(x), x, 0, 6, lc(red)),
           more (xlabel = "X", ylabel = "COS(X)", title = "single function"))$
```

which produces

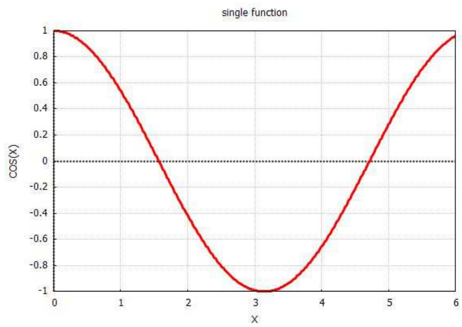

Figure 3: Adding labels, title using more(...)

We can use the **line** command to add a brown x-axis. The arg **lw(1)** forces the line width to be small; **lw(5)** would be a thick line.

```
(%i7) qdraw ( ex1 (cos(x), x, 0, 6, lc(red)),
          line ( 0,0,6,0, lc(brown), lw(1)),
          more (xlabel = "X", ylabel = "COS(X)", title = "single function"))$
```

which produces

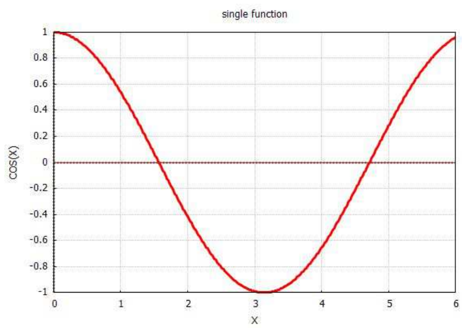

Figure 4: Adding a brown x-axis using line(...)

Thus far the vertical canvas range (y-axis range) has been the default. We can control the vertical range using the extra arg **yr(y1,y2)** as shown here:

```
(%i8) qdraw ( ex1 (cos(x), x, 0, 6, lc(red)), yr (-1.2, 1.2),
         line ( 0,0,6,0, lc(brown), lw(1)),
         more (xlabel = "X", ylabel = "COS(X)", title = "single function"))$
```

which produces

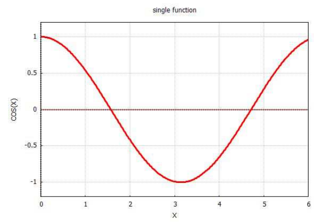

Figure 5: Controlling the vertical range with yr(y1,y2)

All of the above additions to the basic plot can also be done when using the arg **ex(...)**, which controls its own colors, but can be used for fast simultaneous plots, as shown here

```
(%i9) qdraw ( ex ( [x,x^2,x^3],x,-3,3),
            line ( -3,0,3,0, lc(brown), lw(1)),
            more (xlabel = "X",  title = "Using ex(..) for three functions"))$
```

which produces

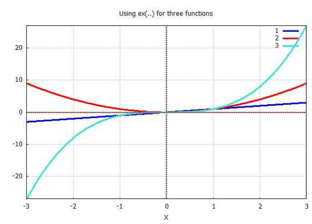

Figure 6: Using ex(...) for three expressions

We can add a vertical range control and move the “key” to the bottom:

```
(%i10) qdraw ( ex ( [x,x^2,x^3],x,-3,3),  yr (-2, 2),
        line ( -3,0,3,0, lc(brown), lw(1)), key (bottom),
         more (xlabel = "X",  title = "Using ex(..) for three functions"))$
```

which produces

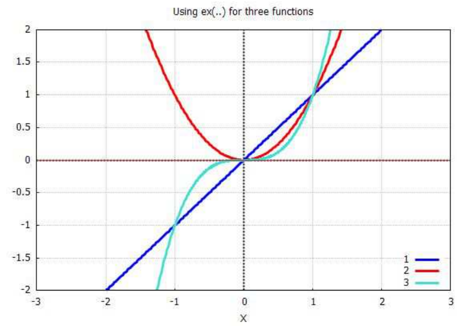

Figure 7: Controlling the vertical range and key position

We can use **pts(...)** to add three points to this plot.

```
(%i11) qdraw ( ex ( [x,x^2,x^3],x,-3,3),  yr (-2, 2),
        line ( -3,0,3,0, lc(brown), lw(1)), key (bottom),
        pts ( [ [-1,-1], [0,0],[1,1] ] ),
        more (xlabel = "X",  title = "Using ex(..) for three functions"))$
```

which produces

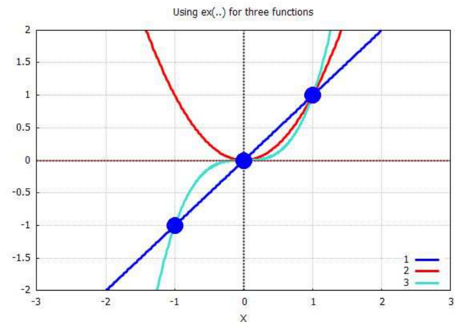

Figure 8: Adding three points using pts (...)

We can use **pc(..)** to control the color and **ps(..)** to control the size of these points

```
(%i12) qdraw ( ex ( [x,x^2,x^3],x,-3,3),  yr (-2, 2),
        line ( -3,0,3,0, lc(brown), lw(1)), key (bottom),
        pts ( [ [-1,-1], [0,0],[1,1] ], ps(2), pc(magenta)),
        more (xlabel = "X",  title = "Using ex(..) for three functions"))$
```

which produces

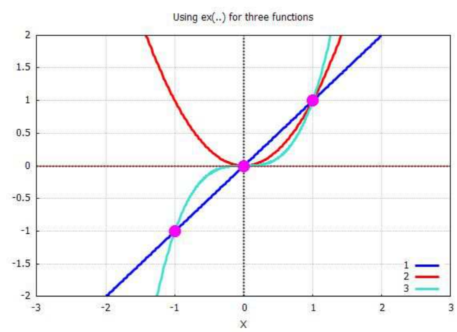

Figure 9: Adjusting point size with ps(..), point color with pc(..)

We can add a key entry for the points using **pk(..)**.

```
(%i13) qdraw ( ex ( [x,x^2,x^3],x,-3,3),  yr (-2, 2),
         line ( -3,0,3,0, lc(brown), lw(1)), key (bottom),
         pts ( [ [-1,-1], [0,0],[1,1] ], ps(2), pc(magenta), pk("intersections")),
         more (xlabel = "X",  title = "Using ex(..) for three functions"))$
```

which produces

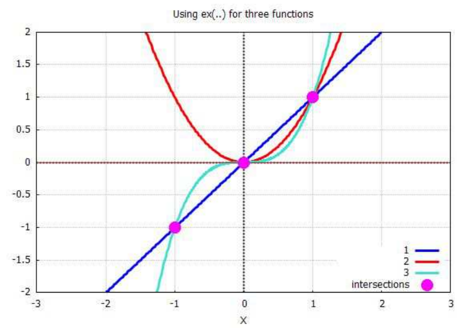

Figure 10: Adding points legend (key) entry with pk(..)

3.1 Default colors and available colors

Using **`ex(...)`** to plot a set (list) of expressions means that the **color** choices are controlled by the program, namely a local list called **`cc`** inside the **`qdraw1`** code. You can see the default color names and what they look like by using the function **`default_colors(nwidth)`**:

```
(%i14) default_colors(15)$
 default color list =  [blue, red, turquoise, brown, magenta, green, black]
```

which prints out the default color list and draws the graphic:

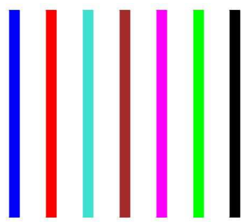

Figure 11: Default colors used by ex(...)

Repeated use of **`pts(..)`** does **not** cycle through colors (note use of **`xr(x1,x2)`** to control horizontal range):

```
(%i15) (L1:[[-1,-1],[-1,0],[-1,1]], L2:[[1,-1],[1,0],[1,1]],
       qdraw ( pts(L1), pts(L2), xr(-2,2),yr(-2,2)))$
```

which produces

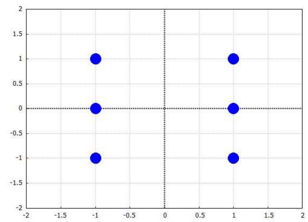

Figure 12: Default color used by pts(...) is blue

We can gain control of colors used for plotting expressions, and also include a meaningful legend (key entry), if we use **ex1(..)** instead of **ex(..)**. The downside is that we have to use a separate **ex1(..)** entry for each expression to be included in the plot. We can then choose any color available in the **draw** package. In the **draw** package section of the Maxima help manual, in the description of Graphic option: color, one finds a list which includes:

white	black	gray0	grey0
gray	grey	light_gray	light_grey
dark_gray	dark_grey	red	light_red
dark_red	yellow	light_yellow	dark_yellow
green	light_green	dark_green	spring_green
forest_green	sea_green	blue	light_blue
dark_blue	midnight_blue	navy	medium_blue
royalblue	skyblue	cyan	light_cyan
dark_cyan	magenta	light_magenta	dark_magenta
turquoise	light_turquoise	dark_turquoise	pink
light_pink	dark_pink	coral	light_coral
orange_red	salmon	light_salmon	dark_salmon
aquamarine	khaki	dark_khaki	goldenrod
light_goldenrod	dark_goldenrod	gold	beige
brown	orange	dark_orange	violet
dark_violet	plum	purple	

You can use the function **show_colors(color_list, nlw)** to display the colors corresponding to any of these names. (Note that you can use hyphenated names without quotes.)

```
(%i16) mycL : [aquamarine,beige,blue,brown,cyan,gold,goldenrod,green,khaki,
            magenta,orange,pink,plum,purple,red,salmon,skyblue,turquoise,
            violet,yellow ]$
(%i17) show_colors(mycL,10)$
 show color list =  [aquamarine, beige, blue, brown, cyan, gold, goldenrod, green,
khaki, magenta, orange, pink, plum, purple, red, salmon, skyblue, turquoise, violet,
yellow]
```

which produces

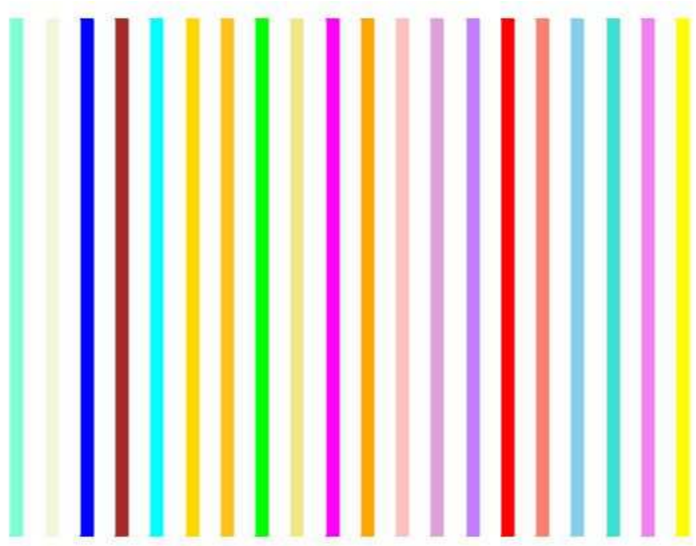

Figure 13: Some of the available colors in the draw package

As an example of multiple uses of **ex1** to gain control over the colors of individual expression plots, we make a simultaneous plot of the first few Bessel functions of the first kind $J_n(x)$ for integral n and real x,

```
(%i18) 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:

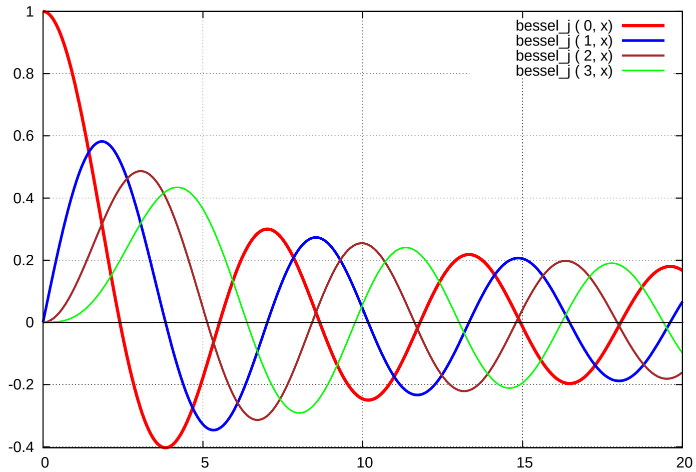

Figure 14: $J_n(x)$

Here is a plot of $J_0(\sqrt{x})$ using **ex1(..)**:

```
(%i19) 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) )") ) )$
```

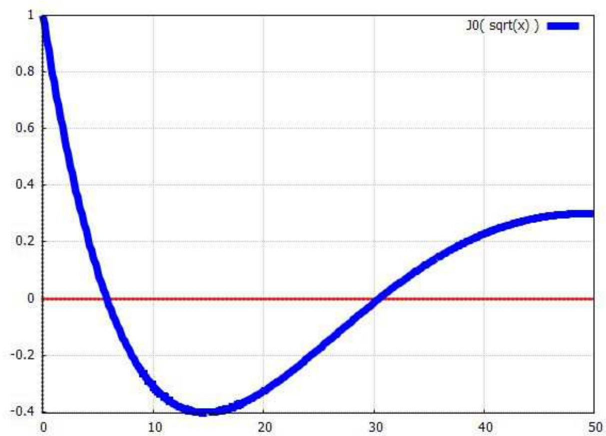

Figure 15: $J_0(\sqrt{x})$

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.

3.2 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 behavior using the expression $x\,e^{-x}$ bound to the symbol **`g`**.

```
(%i20) g : x*exp(-x)$
(%i21) qdraw( ex1( log(g),x,0.001,10, lc(red) ),yr(-8,0)  )$
```

which displays the plot

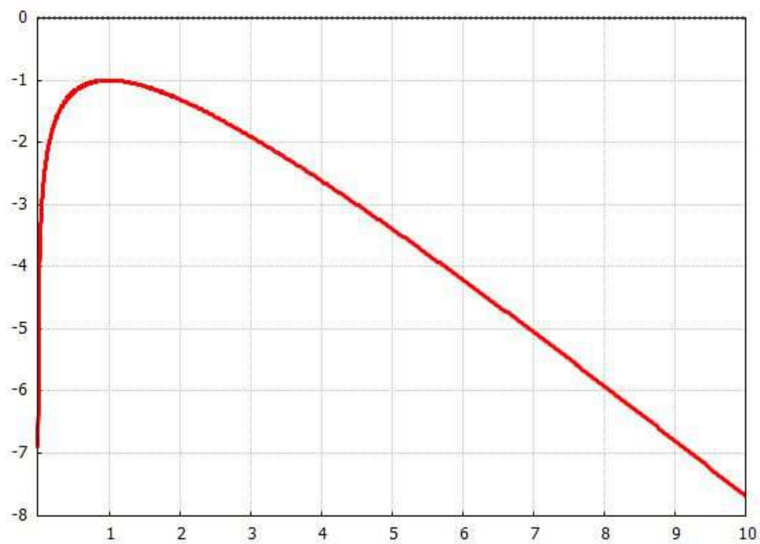

Figure 16: Linear Graph Paper Plot of $\ln(g)$

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:

```
(%i22) 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:

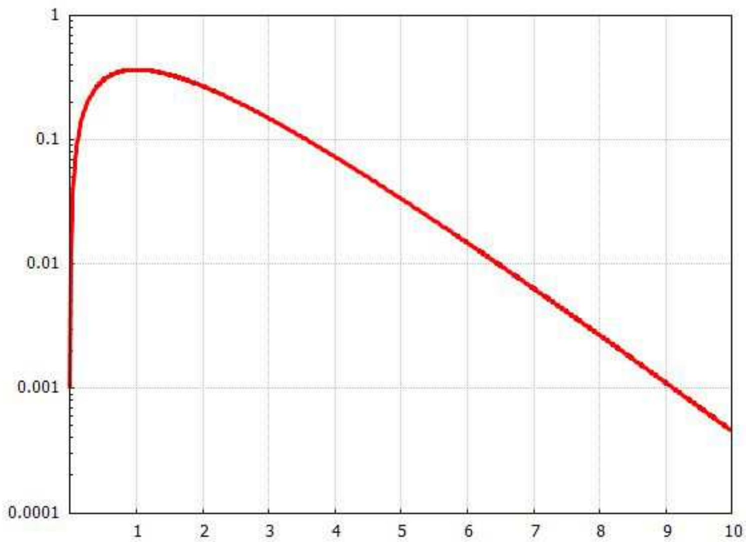

Figure 17: Log Paper Plot of g

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 expression **g**, we generate this plot by using the **log(x)** option to **qdraw**:

```
(%i23) qdraw( ex1(g, x, 0.001,10,lc(red),lw(7) ),
                                    yr(0,0.4), log(x) )$
```

This generates the plot

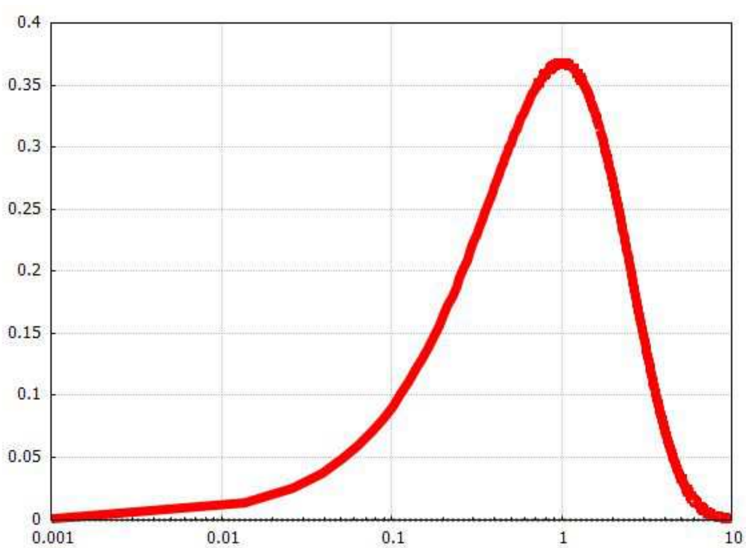

Figure 18: Log-Linear Plot of g

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**.

```
(%i24) qdraw( ex1(g, x, 0.001,10,lc(red) ),
                  yr(0.0001,1), log(xy) )$
```

which produces

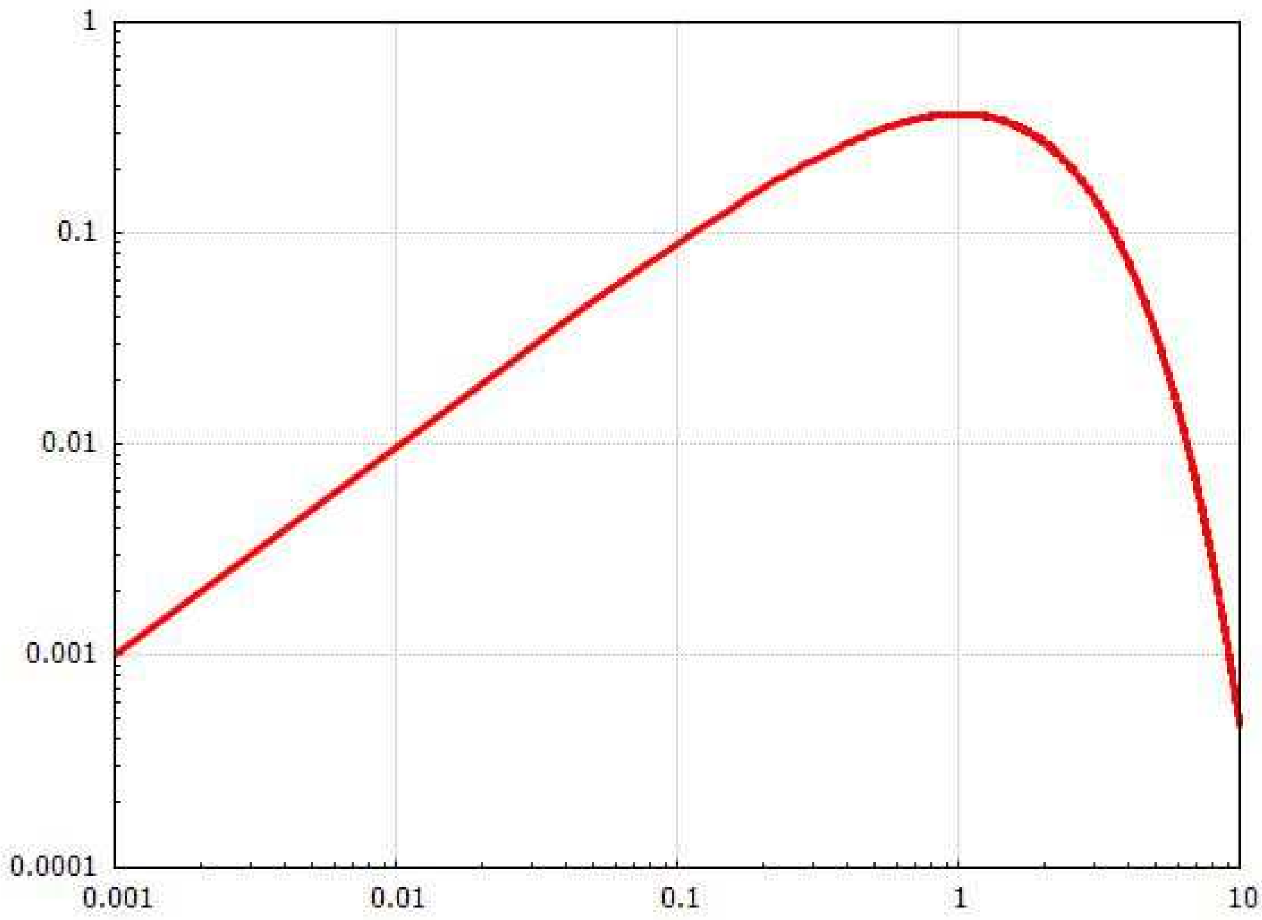

Figure 19: Log-Log Plot of g

3.3 Placing discrete points: the syntax of pts(...)

The syntax of **pts(...)** is

```
pts ( pointlist, pc(color), ps(nsize), pt(ntype), pj(nwidth), pk(string) )
```

The only required argument is the first argument **`pointlist`** which has the form:
`[ [x1,y1], [x2,y2], [x3,y3],...]` .
The remaining arguments, such as **`ps`**, are all optional and may be entered in any order following the first required argument.

The optional argument **`pc(color)`** (point color) overrides the default color (blue); an example is **`pc(red)`**. The point color should be a **name**, not a number.
The optional argument **`ps(nsize)`** (point size) overrides the default size (3), and an example is **`ps(2)`**. The point size should be a positive integer.
The optional argument **`pt(ntype)`** (point type), in which **`ntype`** is a positive integer in the range (1 - 15) overrides the default type (7), which is the integer used for a filled circle type. For example, **`pt(6)`** would request an dot surrounded by an open circle rather than the default filled circle.

The function **`point_types()`**, defined in **`qdraw.mac`**, makes a graphic which shows the correspondence between the integer **`t`** used and the point image produced.

```
(%i25) point_types()$
```

This produces

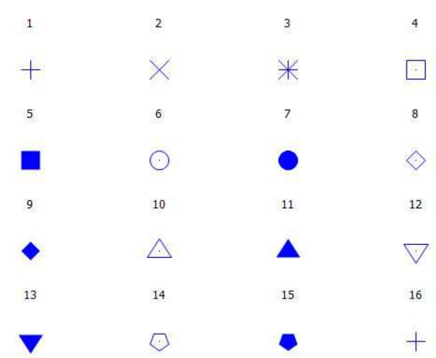

Figure 20: Point Type Integer Table

The optional argument **`pj(nwidth)`**, (points joined) if present, will cause the points provided by the nested list **`pointlist`** to be joined using a line whose width is given by the integer **`nwidth`**; an example is **`pj(2)`** which would use the line width 2.
The optional argument **`pk(string)`** (points key) provides text for a key entry for the set of points represented by **`pointlist`**; an example is **`pk("case x^2")`**.

Here is a simple example of two sets of points, using different types, colors, sizes, legends, and using **`pj(n)`** for the second set of points.

```
(%i26) (L1:[[-1,-1],[-1,0],[-1,1]], L2:[[1,-1],[1,0],[1,1]],
       qdraw ( pts(L1,pt(3),ps(1),pc(red),pk("type 3")),
               pts(L2, pt(10),ps(2),pc(black),pj(3),pk("type 10")),
              xr(-2,2),yr(-2,2)))$
```

which produces

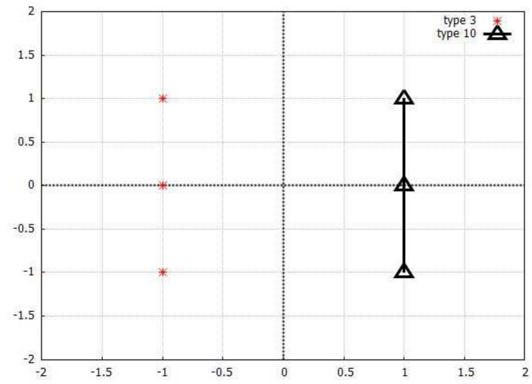

Figure 21: Two examples of pts(...) with different options

You can construct your own "line types" by combining different types and colors of points, either joining or not joining them.

3.4 Using the line_type option with draw2d

The **qdraw** program does **not** allow access to the **draw2d line_type** option. If you want to construct an eps file which incorporates the **line_type** specification allowed by **draw2d**, you should use the standard **draw2d** syntax we show here.

```
(%i27) draw2d( title = "draw2d line type examples",
      file_name = "c:/work5/linetype1", terminal = 'eps,
      line_width = 4,  yrange = [-0.2,2.2],
      line_type = dots, explicit (x^2,x,-1,1),
      color = red, line_type = solid, explicit (0.2 + x^2,x,-1,1),
      color = turquoise, line_type = dashes, explicit (0.4 + x^2,x,-1,1),
      color = brown,line_type = dot_dash, explicit (0.6 + x^2,x,-1,1),
      color = magenta,line_type = short_long_dashes, explicit (0.8 + x^2,x,-1,1),
      color = green, line_type = short_short_long_dashes, explicit (1 + x^2,x,-1,1))$
```

which yields the figure

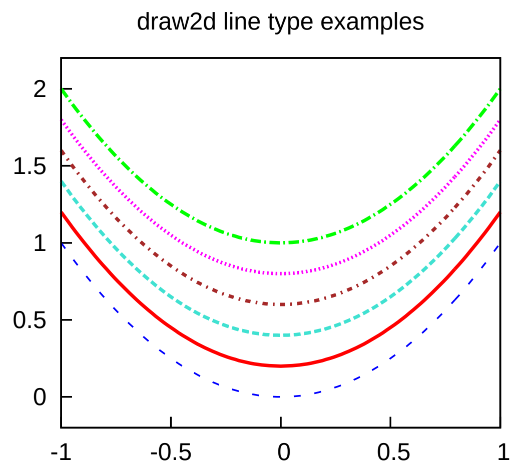

Figure 22: line type options in draw2d with eps terminal

The input **`file_name = "c:/work5/linetype1", terminal = 'eps,`** in the above **`draw2d`** command causes the produced file **`linetype.eps`** to be written to the **`c:\work5`** folder when using a Microsoft Windows operating system. You can use the **`GSview`** graphical interface for **`Ghostscript`** (**`www.gsview.com`**) for an independent **`*.eps`** file viewer. If you replace **`'eps`** with **`'svg`**, the resulting graphics file can be viewed with Inkscape.

4 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 entries are optional and can be in any order. The parameter **`u`** should match the parameter used in the first two args.

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

```
(%i3) qdraw(xr(-1.5,2),yr(-2,2),
        para(sin(t),sin(2*t),t,0,2*%pi ),
        pts( [ [0,0] ],ps(1),pc(brown),pk("t = 0")),
        pts( [ [sin(%pi/8),sin(%pi/4)] ],ps(1),pc(red),pk("t = pi/8")),
        pts( [ [1,0] ],ps(1),pc(green),pk("t = pi/2")),
        more (title = "parametric plot", xlabel = "sin(t)", ylabel = "sin(2*t)"))$
```

which produces the plot:

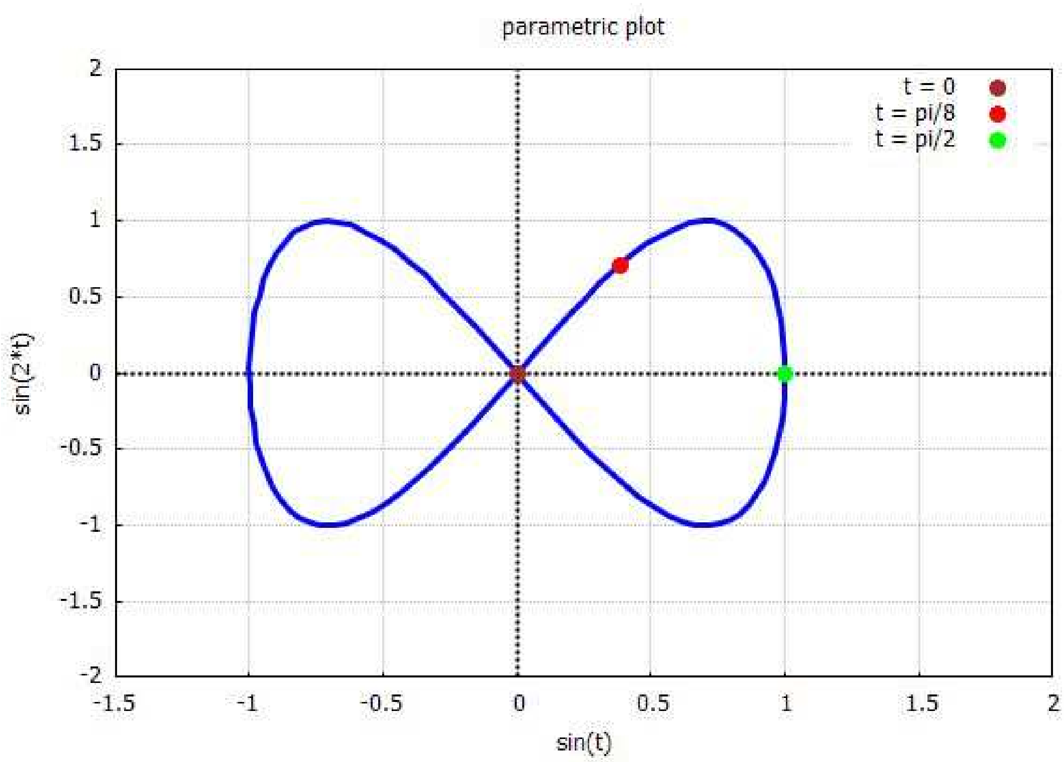

Figure 23: Parametric plot with $x = \sin(t), y = \sin(2\,t)$

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

```
(%i4) qdraw(xr(-3,4),yr(-1,40), para(2*cos(u),u^2,u,0,2*%pi) ,
         pts([ [2,0] ],ps(1),pc(blue),pk("u = 0")),
         pts( [ [0,(%pi/2)^2] ],ps(1), pc(red), pk("u = pi/2")),
         pts([ [-2,%pi^2]],ps(1),pc(green),pk("u = pi")),
         pts( [[0,(3*%pi/2)^2]],ps(1),pc(magenta),pk("u = 3*pi/2")),
         more (title = "parametric plot",xlabel = "2*cos(u)",ylabel = "u^2"))$
```

which yields the plot:

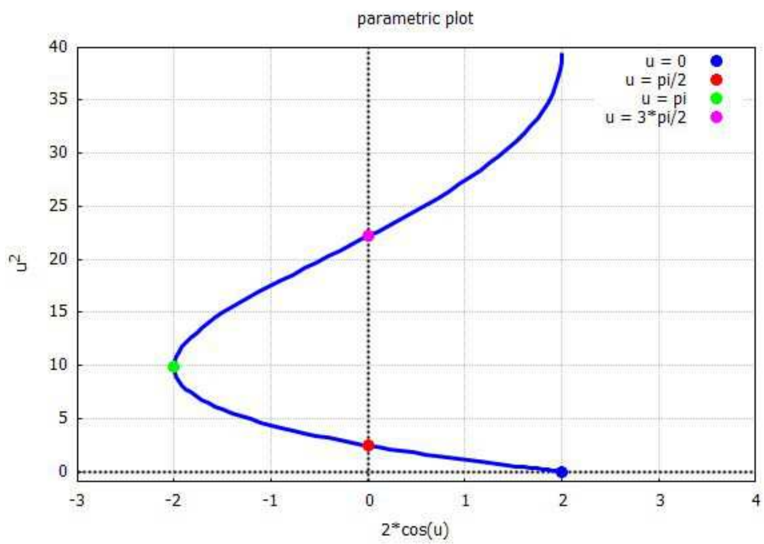

Figure 24: Parametric plot with $x = 2\, cos(u), y = u^2$

5 Polar Plots with polar(...)

A "polar plot" plots the points $(x = r(\theta)\ \cos(\theta), y = r(\theta)\ \sin(\theta))$, where the expression $r(\theta)$ 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 $r(\theta) = 10/\theta$. The parameter **`t`** represents θ and we make a plot for $1 \le \theta \le 3\,\pi$.

```
(%i5) 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)]],ps(2),pc(red),pk("t = 1 rad")),
            pts([[5*cos(2),5*sin(2)]],ps(2),pc(blue),pk("t = 2 rad") ),
            line(0,0,5*cos(2),5*sin(2)),
            more(title = "polar plot",xlabel = "10*cos(t)/t",ylabel = "10*sin(t)/t"))$
```

which looks like:

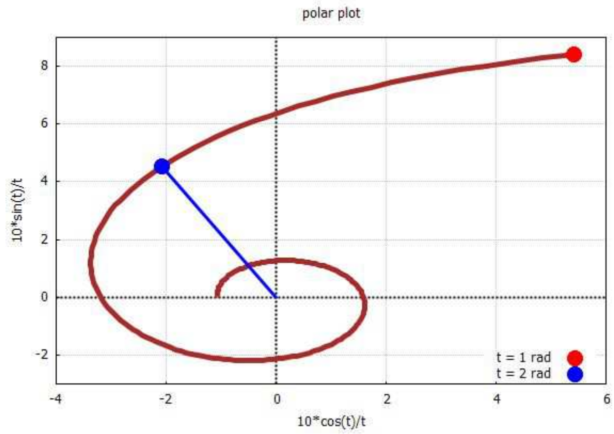

Figure 25: Polar Plot with $r = 10/\theta$

6 Implicit plots with imp(...) and imp1(...)

An implicit plot is here a two dimensional plot of an implicitly defined curve.

6.1 Quick implicit plots with imp(...)

The quick plotting function **imp(...)** syntax has two forms:
either `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 $x \to u$ and $y \to v$. 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:

```
(%i3) e : sin(2*x)*cos(y)$
(%i4) qdraw( imp( e=0.4,x,-3,3,y,-3,3 ),cut(key),
          more(title=" sin(2 x) cos(y) = 0.4 ",xlabel = "x", ylabel = "y"))$
```

which produces the “implicit plot”:

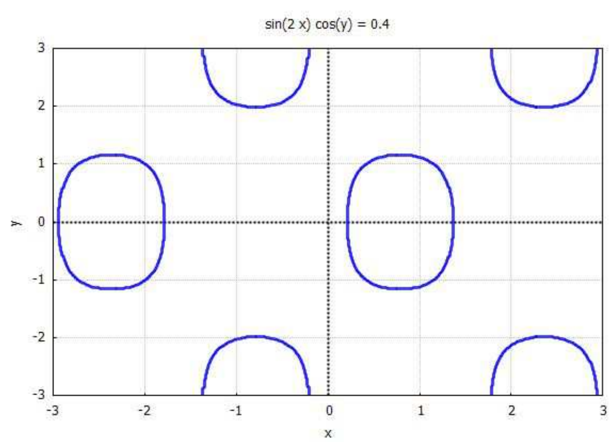

Figure 26: Implicit plot of $\sin(2\,x)\ \cos(y) = 4/10$

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:

```
(%i5) qdraw( imp( [e=0.4,e=0.7,e=0.9],x,-3,3,y,-3,3 ),cut(key),
         more(title=" sin(2 x) cos(y) = 0.4,0.7,0.9 ",xlabel = "x", ylabel = "y"))$
```

The resulting plot with the default rotating color set is

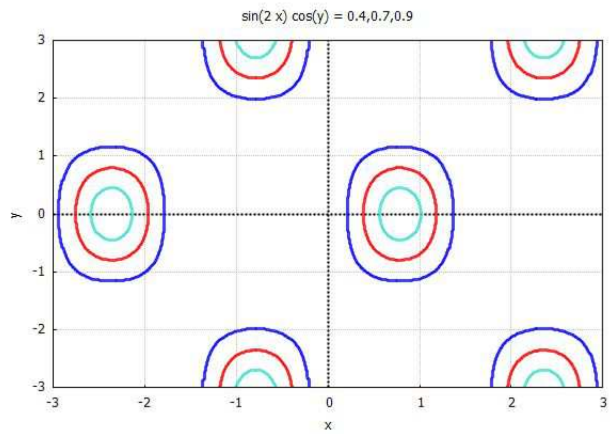

Figure 27: plot of $\sin(2\,x)\;\cos(y) = 0.4, 0.7, 0.9$

We need to arrange that the horizontal canvas width is about 1.4 times 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:

```
(%i6) qdraw( imp( [e = 0.4,e = 0.7,e = 0.9] ,x,-4.2,4.2,y,-3,3 ), cut(key) )$
```

which produces a slightly different looking plot:

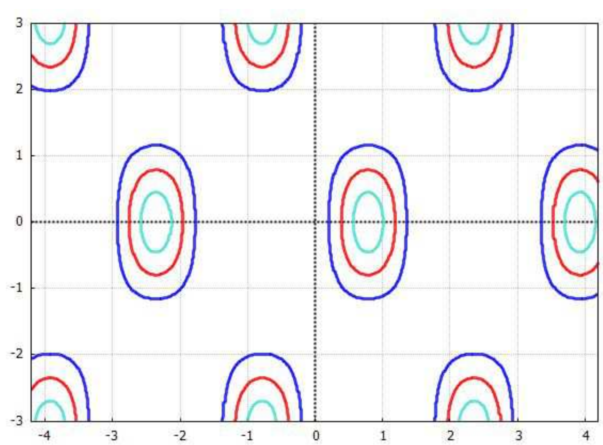

Figure 28: using canvas limits $\Delta x = 1.4\,\Delta y$

6.2 implicit plot ($r = 1 - \cos(\theta),\ r, 0, 2, \theta, 0, 2\,\pi$)

There is no **draw2d** implicit plot version specifically adapted to a description in terms of polar coordinates (r, θ), so we present a numerical method to make a plot which starts with a finite grid of θ values.

In our file **qdraw.mac** we have a function called **make_xygrid**:

```
make_xygrid(Xfunc,Yfunc,Th0,Thf,Num) :=
block([dTh,Xgrid,Ygrid], numer:true,
      dTh : float((Thf - Th0)/Num),
          Xgrid : makelist(Xfunc(Th0 + n*dTh),n,0,Num),
          Ygrid : makelist(Yfunc(Th0 + n*dTh),n,0,Num),
          makelist([Xgrid[n],Ygrid[n]],n,1,Num+1))$
```

Our approach is then to express the x and y coordinates in terms of the angle (in radians) alone, by replacing r by its expression in terms of the angle. We then construct a set of (x, y) points (an "x-y-grid") corresponding to various discrete values of the angle (in radians), using **make_xygrid**. We then use **qdraw (pts(...),...)** to make a plot, using the option **pj(m)** to join the discrete points. If we choose a fine enough mesh of angle values (ie., a large enough value of **Num**), then we approach an implicit plot of the type sought. We will divide the angle interval $[0, 2\,\pi]$ into **Num = 20** subintervals as a first experiment.

```
(%i7) x(th):= cos(th)*(1-cos(th))$
(%i8) y(th):= sin(th)*(1-cos(th))$
(%i9) fpprintprec : 6$
(%i10) xygrid : make_xygrid(x,y,0,2*%pi,20);
(%o10) [[0.0, 0.0], [0.046548, 0.0151244], [0.154508, 0.112257],
[0.242294, 0.333489], [0.213525, 0.657164], [6.12303e-17, 1.0],
[- 0.404508, 1.24495], [- 0.933277, 1.28455], [- 1.46353, 1.06331],
[- 1.85557, 0.60291], [- 2.0, 2.44921e-16], [- 1.85557, - 0.60291],
[- 1.46353, - 1.06331], [- 0.933277, - 1.28455], [- 0.404508, - 1.24495],
[- 1.83691e-16, - 1.0], [0.213525, - 0.657164], [0.242294, - 0.333489],
[0.154508, - 0.112257], [0.046548, - 0.0151244], [0.0, 0.0]]
(%i11) qdraw(pts(xygrid,ps(0.1),pj(1)))$
```

which produces

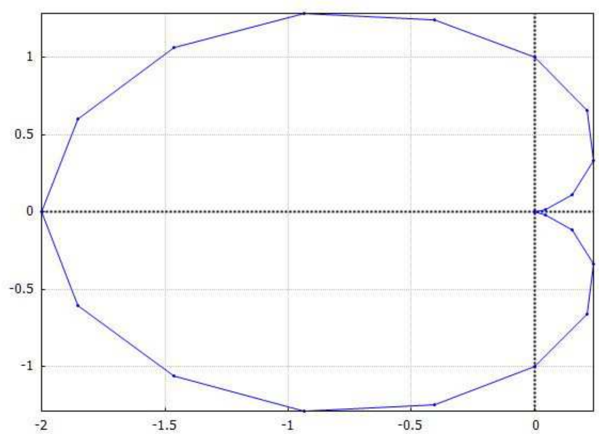

Figure 29: implicit plot $(r = (1 - \cos(\theta)))$ for Num = 20

We can then add a title, x and y labels, and adjust the x and y ranges to get a better looking plot (keeping the 21 point description intact). We have approximately enforced our rule of thumb $\Delta x \approx 1.4\,\Delta y$ for visual realism.

```
(%i12) qdraw(pts(xygrid,ps(0.1),pj(1)),xr(-3,1),yr(-1.4,1.4),
        more(title = "r = (1- cos(th))",xlabel = "x = r cos(th)",
                ylabel = "y = r sin(th)"))$
```

which produces

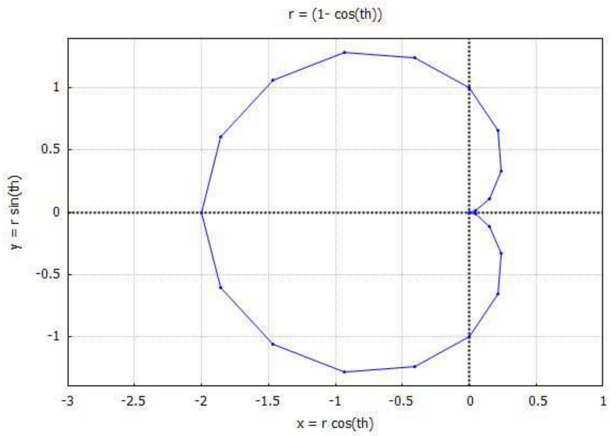

Figure 30: implicit plot ($r = (1 - \cos(\theta))$) for Num = 20

We can then increase the angle grid to **`Num = 60`** subintervals:

```
(%i13) xygrid : make_xygrid(x,y,0,2*%pi,60)$
(%i14) qdraw(pts(xygrid,ps(0.1),pj(1)),xr(-3,1),yr(-1.4,1.4),
        more(title = "r = (1- cos(th))",xlabel = "x = r cos(th)",
                ylabel = "y = r sin(th)"))$
```

which produces a fairly smooth plot

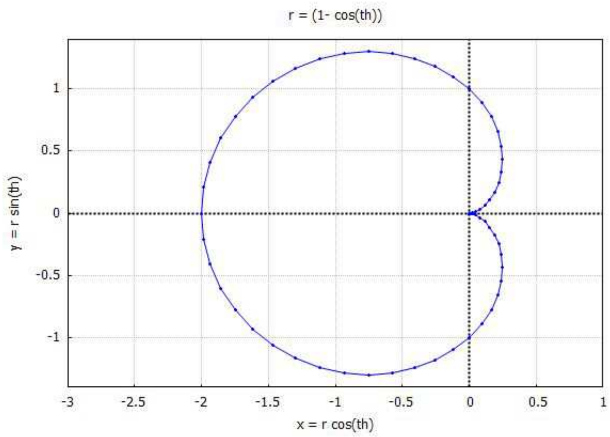

Figure 31: implicit plot ($r = (1 - \cos(\theta))$) for Num = 60

We can compare our numerical grid approach above with using **`para`** for a parametric plot.

```
(%i15) qdraw(para(x(th),y(th),th,0,2*%pi,lw(1)),xr(-3,1),yr(-1.4,1.4),
        more(title = "r = (1- cos(th))",xlabel = "x = r cos(th)",
                ylabel = "y = r sin(th)"))$
```

which produces

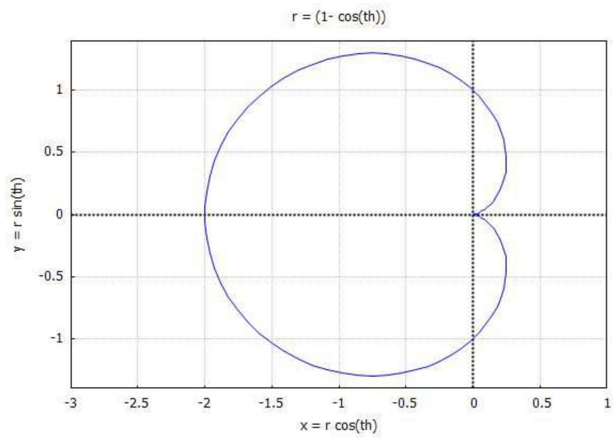

Figure 32: Using **para** for plot of ($r = (1 - \cos(\theta))$)

We have good agreement between the two methods.

6.3 Implicit plot with two equations

```
(%i16) qdraw ( imp([x^2 - y^2 = 1, y = exp(x)],x,-1.4*%pi,1.4*%pi,y,-%pi,%pi),
         more( xlabel = "x", ylabel = "y", title = "x^2 - y^2 = 1, y = exp(x)"))$
```

which produces

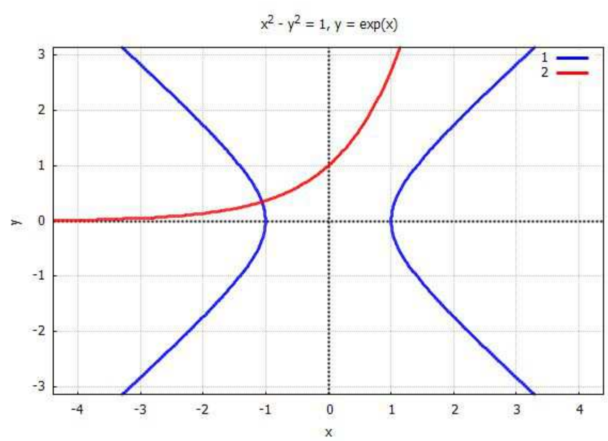

Figure 33: Using **imp** for plot of ($x^2 - y^2 = 1, y = e^x$)

6.4 Implicit plot of a circle

Since **1.4*1.2 = 1.68**,

```
(%i17) qdraw ( imp (x^2 + y^2 = 1,x,-1.68,1.68,y,-1.2,1.2),cut(key),
         more(title = "x^2 + y^2 = 1",xlabel = "x",ylabel = "y"))$
```

which produces

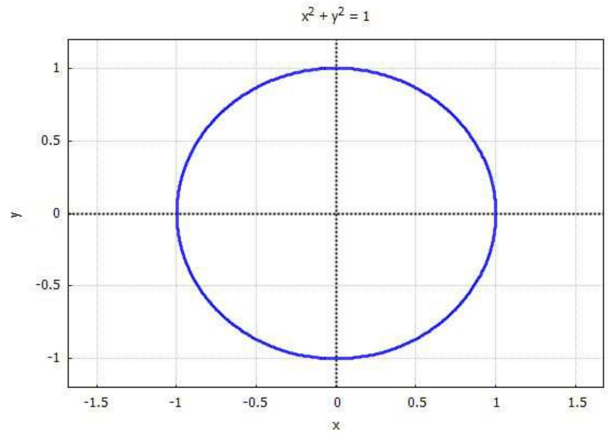

Figure 34: Using **imp** for plot of $x^2 + y^2 = 1$

6.5 Implicit plot of concentric circles

```
(%i18) e : (x^2+y^2-1)*(x^2+y^2-0.73)*(x^2+y^2-0.5)*(x^2+y^2-0.3)$
(%i19) qdraw (imp (e=0,x,-1.68,1.68,y,-1.2,1.2),cut(key),
         more (xlabel = "x",ylabel = "y",title = "circles"))$
```

produces

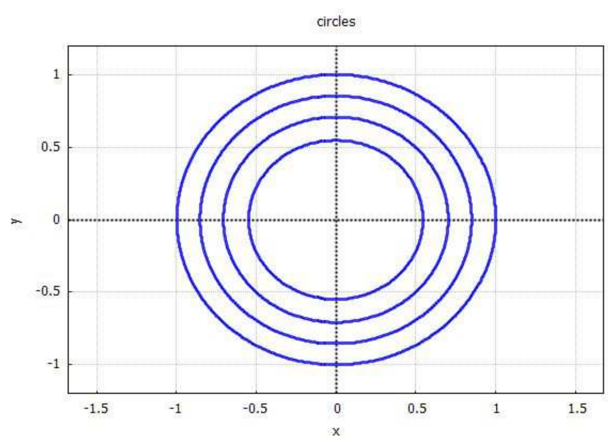

Figure 35: Using **imp** for plot of concentric circles

6.6 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 $x \to u$, and $y \to v$. 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 $63.4\,\mathrm{deg}$ (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 (although this is highly dependent on the graphics window width and height).

```
(%i20) eqn1 : 5*x^2 + 4*x*y + 8*y^2 - 16*x + 8*y - 16 = 0$
(%i21) eqn2 : y+1 = 2*(x-2)$
(%i22) eqn3 : y+1 = -(x-2)/2$
(%i23) 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") ) )$
```

which produces the plot

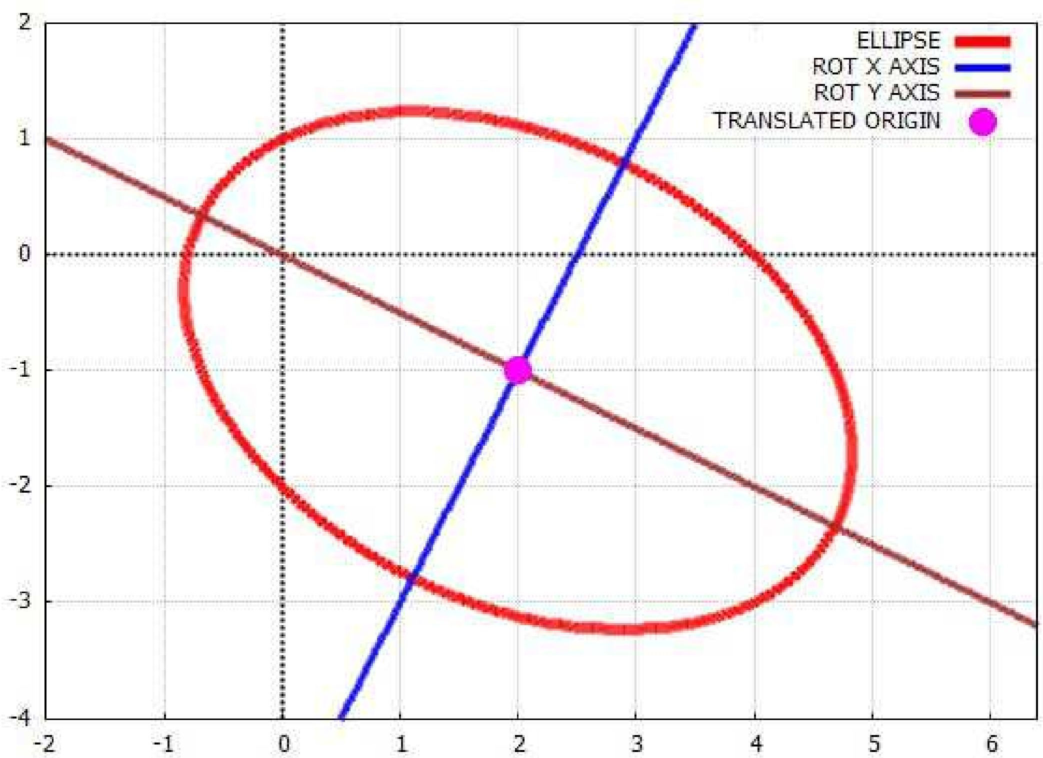

Figure 36: Rotated and Translated Ellipse

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

```
(%i24) qdraw( imp1(y^3=x^2,x,-3,3,y,-1,3, lw(10), lc(dark-blue), lk("Y^3 = X^2")))$
```

which produces the plot:

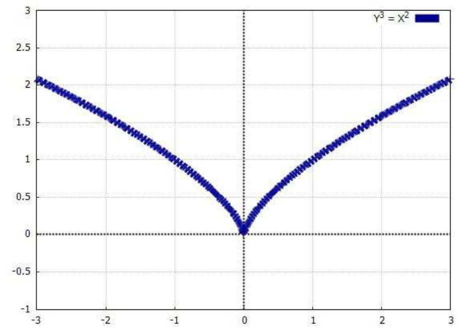

Figure 37: Implicit Plot of $y^3 = x^2$

7 Contour Plots with contour(...)

The function **`contour(...)`**, as an argument to either **`qdraw`** or **`wxqdraw`**, has the 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 uses the supplied numerical values for contour curves 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):

```
(%i3) e : sin(2*x)*cos(y)$
(%i4) qdraw( contour(e, 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, with the key, grid, and (x,y)-axes removed, drawn in the color "blue":

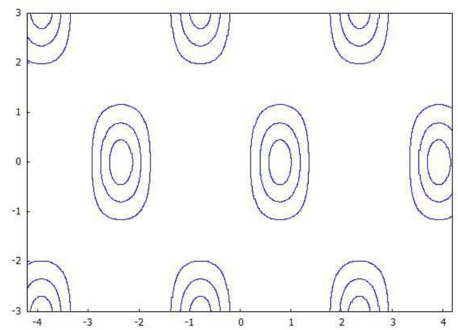

Figure 38: simplest default contour example using cvals(..) form

Since 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 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**(add-options), where the "add-options" are any or all of the set **`{grid,xaxis,yaxis,xyaxes}`**.

For example one could use

```
(%i5) qdraw( contour(e, x,-4.2,4.2, y,-3,3, cvals(0.4,0.7,0.9), lw(2), add(grid)), ipgrid(15))$
   contour working...
```

which also adds the **qdraw** function **ipgrid(n)** to get smoother curves than the default. This produces the plot

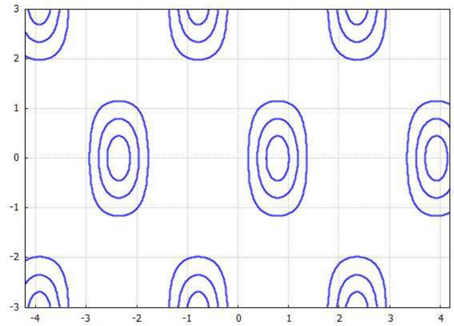

Figure 39: adding lw(2), add(grid), ipgrid(15)

Thus the following invocation of **contour**:

```
(%i6) qdraw( contour(e,x,-4.2,4.2,y,-3,3,cvals(0.4,0.7,0.9),
              lw(2), lc(brown) ), ipgrid(15) )$
```

produces:

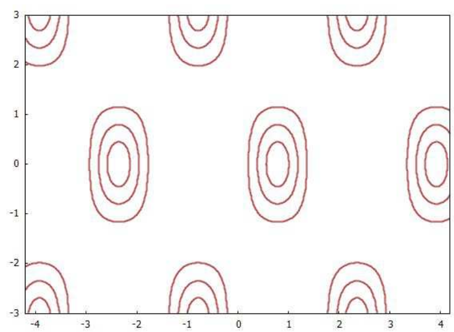

Figure 40: adding lw(2), lc(brown), ipgrid(15)

The added **qdraw** function **ipgrid** with argument 15 over-rides the **qdraw** default value of the **draw2d** parameter `ip_grid_in` . The **draw2d** default for this parameter is 5, which results in some "jaggies" 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(n,min,max)", form of **contour**:

```
(%i7) qdraw( contour(e, x, -4.2,4.2, y,-3,3, crange(4,0.2,0.9),
                                lc(brown) ), ipgrid(15)  )$
   contour working...
```

which produces the plot:

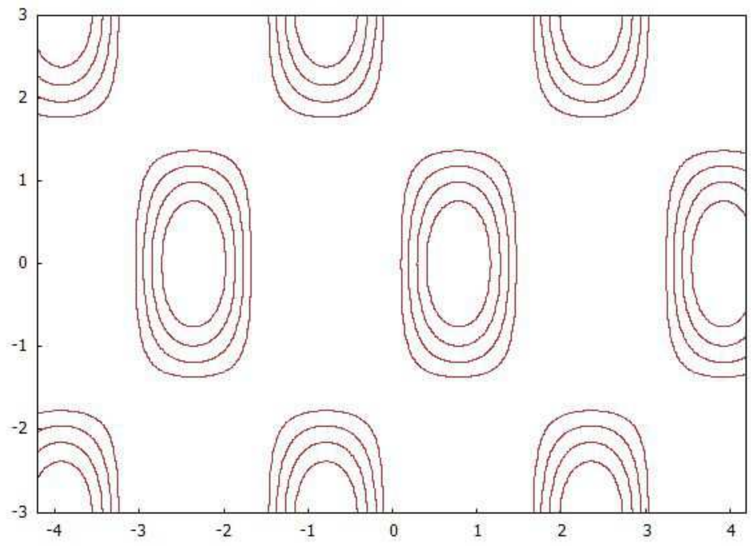

Figure 41: using $crange(4, .2, .9)$

A final example illustrates the **contour** option **add(xyaxes)** to make a contour plot of the expression **`sin(x)*sin(y)`**, using the **`crange`** form.

```
(%i8) 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),
             more(title = "sin(x) sin(y) contours",xlabel = "x",
                   ylabel = "y"))$
   contour working...
```

which produces

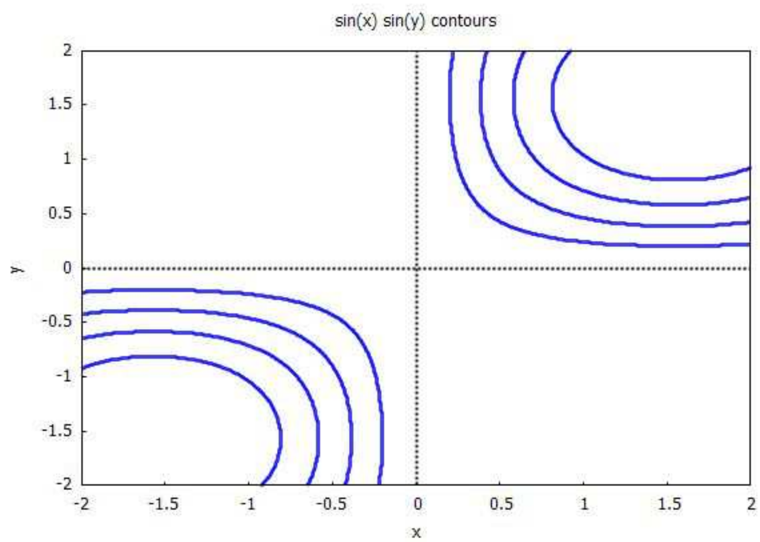

Figure 42: using add(xyaxes) option

8 Density Plots

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 a function of two variables takes on large values. Four completely separate density plotting functions, **qdensity**, **wxqdensity**, **qdensity_mat**, and **wxqdensity_mat** are defined in **qdraw.mac**. These four density plotting functions are completely independent of the default conventions and syntax associated with the function **qdraw**.

8.1 qdensity (expr, [x, x1, x2, dx], ...) or wxqdensity (expr, [x, x1, x2, dx], ...)

The syntax of **qdensity** or **wxqdensity** is

```
qdensity(expr,[x,x1,x2,dx],[y,y1,y2,dy], options )
```

(which assumes the expression **expr** depends on the symbols **x** and **y**), 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 of the three words: **blue**, **gray**, or **color**, or else a three positive integer "red, green, blue" specification, such as **palette(8,3,1)** (which produces a density plot in "shades of red").

To use the **pic(type, filename)** option, **type** can be either **eps** or **eps_color**, and the filename is a **string** – for example: **"c:/work2/case5a"** (the double quotes are required).

In the second and third argument (lists), 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 example of an expression is $x\,y$, which increases from zero at the origin to 1 at $(x = 1, y = 1)$.

```
(%i3) qdensity(x*y,[x,0,1,0.2],[y,0,1,0.2] )$
```

This produces the density plot:

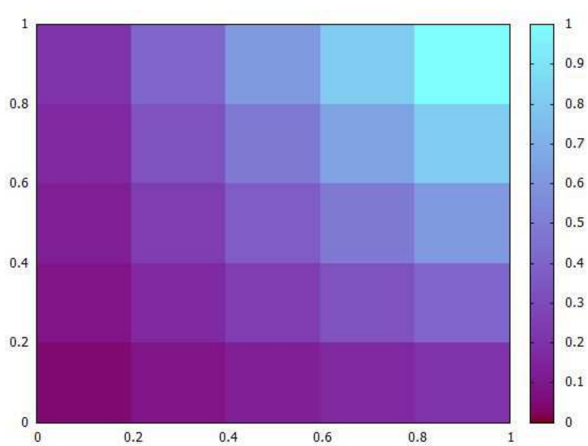

Figure 43: default palette density plot

If we use the gray palette option

```
(%i4) qdensity(x*y,[x,0,1,0.2],[y,0,1,0.2],palette(gray) )$
```

we get

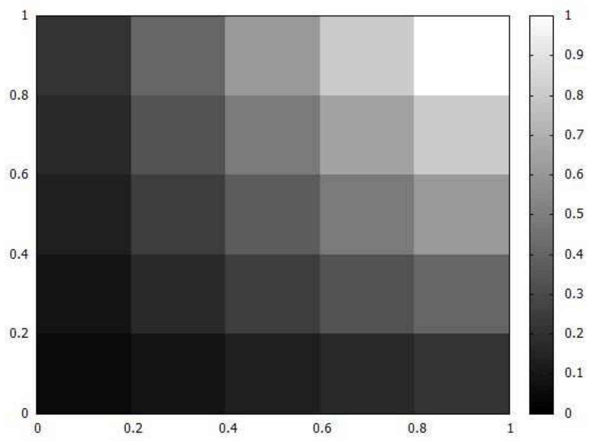

Figure 44: palette(gray) option

while if we use palette(color), we get

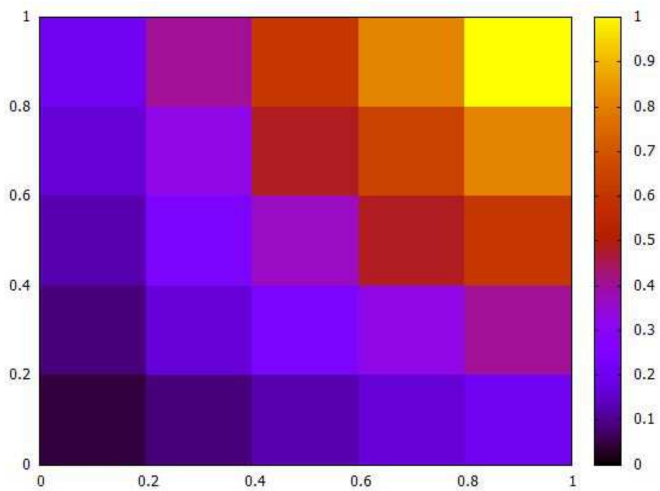

Figure 45: palette(color) option

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`**,

```
(%i5) qdensity(x*y,[x,0,1,0.05],[y,0,1,0.05] )$
```

yields a refined density plot with **`20 x 20 = 400`** painted rectangular panels.

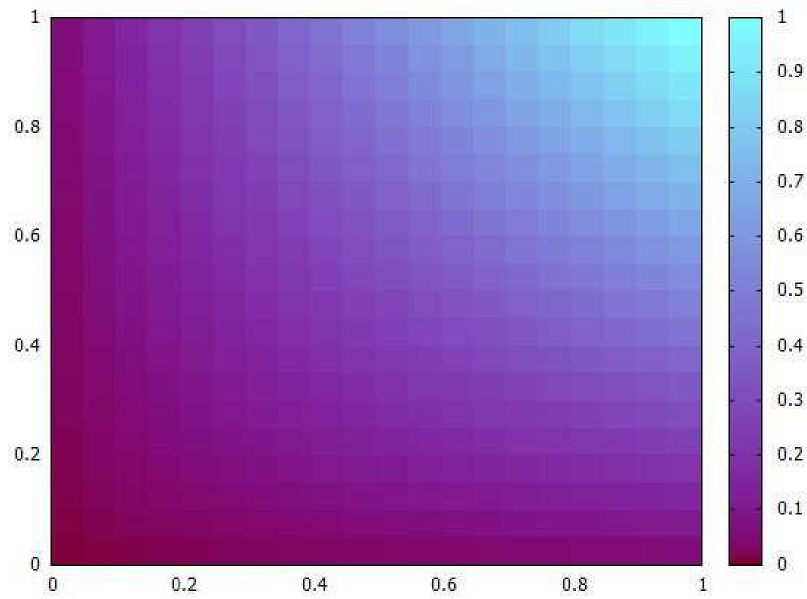

Figure 46: interval set to 0.05

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

```
(%i6) qdensity(sin(x)*sin(y),[x,-2,2,0.05],[y,-2,2,0.05] )$
```

which yields

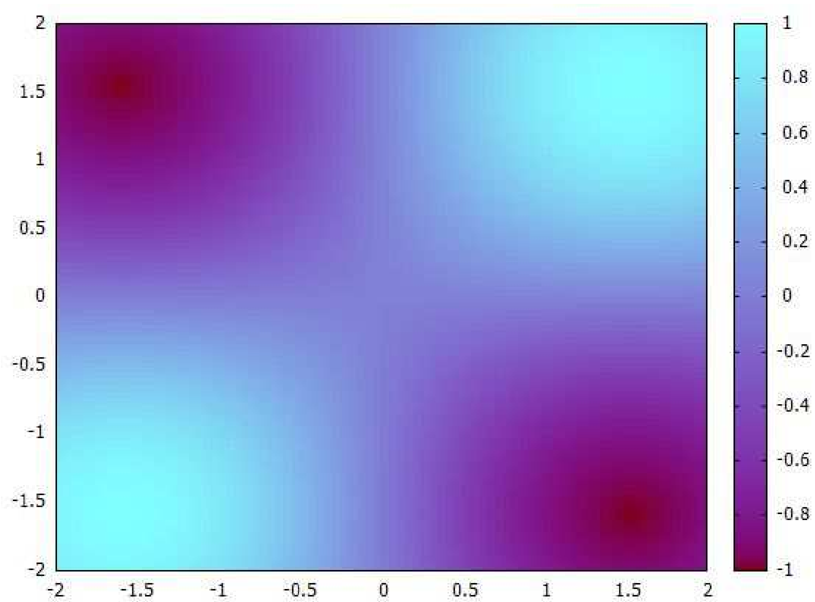

Figure 47: $\sin(x)\ \sin(y)$

8.2 qdensity_mat (Amatrix, [x1,x2],[y1,y2] , options) or wxqdensity_mat

The syntax of **qdensity_mat** or wxqdensity_mat is

```
        qdensity_mat (Amatrix, [x1,x2], [y1,y2], options )
```

where the two optional arguments (which can be in any order after the first three args), are **`palette(p)`** and **`pic(type,filename)`**.

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 of the three words: **`blue`**, **`gray`**, or **`color`**, or else a three positive integer "red, green, blue" specification, such as **`palette(8,3,1)`** (which produces a density plot in "shades of red").

To use the **`pic(type, filename)`** option, **`type`** can be either **`eps`** or **`eps_color`**, and the filename is a **string** – for example: **`"c:/work2/case5a"`** (the double quotes are required).

An interactive example in which we create a matrix containing values of **`mod (x,y)`**, (x modulo y), using integral values $1 \leq x \leq 30$ and integral values $1 \leq y \leq 20$, and then invoke **`qdensity_mat`** follows. Note that the matrix thus constructed has 20 rows and 30 columns, and 600 matrix elements. Also, we routinely set **`display2d:false`** in our **`maxima-init.mac`** file.

```
(%i7) makelist( makelist(mod(x,y),x,1,5),y,1,4);
(%o7) [[0,0,0,0,0],[1,0,1,0,1],[1,2,0,1,2],[1,2,3,0,1]]
(%i8) mod_table : makelist( makelist(mod(x,y),x,1,30),y,1,20)$
(%i9) length(mod_table);
(%o9) 20
(%i10) first(mod_table);
(%o10) [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
(%i11) last(mod_table);
(%o11) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,1,2,3,4,5,6,7,8,9,10]
(%i12) M : apply ('matrix, mod_table)$
(%i13) length(M);
(%o13) 20
(%i14) length(transpose(M));
(%o14) 30
(%i15) row (M,20);
(%o15) matrix([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0,1,2,3,4,5,6,
               7,8,9,10])
(%i16) qdensity_mat(M,[1,30],[1,20])$
```

which produces the plot

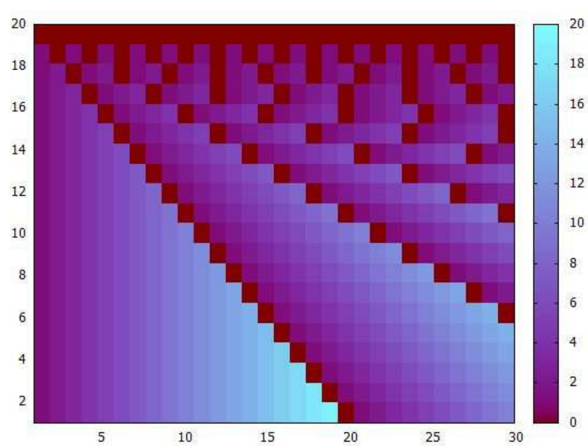

Figure 48: x modulo y

9 Scatterplot Example: Old Faithful Wait Times vs. Eruption Durations

The Old Faithful geyser (Yellowstone National Park) data file **faithful.dat** (available with the Ch. 13 files) contains 272 data points describing geyser eruption events, with the first number being the duration (in min.) of the eruption event, and the second number being the time (min) from the end of the eruption event to the start of the next eruption event (the "wait time").

We will make a plot of the "wait times" (vertical) vs the "eruption times" (horizontal). We first use **read_nested_list** to create a nested list of event "points."

```
(%i3) fL : read_nested_list("c:/work5/faithful.dat")$
(%i4) fll(fL);
(%o4) [[3.6,79],[4.467,74],272]
```

The **qdraw.mac** function **fll** returns **[first, last, length]** of the given list. We can then use the **pts** arg to **qdraw** to make a simple scatterplot of these points.

```
(%i5) qdraw (pts (fL, ps(1), pc(black), pt(6)))$
```

which produces

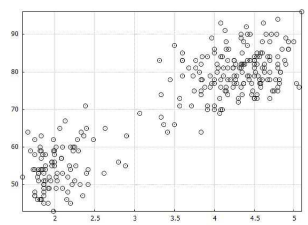

Figure 49: Old Faithful: Wait Times vs. Duration Times

We see from this scatterplot that the wait times increase after an eruption event which has a long duration. If the eruption duration is of the order of 4.5 min, then the wait time for the next eruption is of the order of 80 min. This makes physical sense, since a long duration eruption relieves more stress, and it should take longer for the stress to reach the next eruption stage.

We let **fLs** be the subset of event "points" which satisfy the condition (eruption duration) $<$ 3 min.

```
(%i6) fLs : []$
(%i7) for j thru length(fL) do
          if fL[j][1] < 3 then fLs : cons(fL[j],fLs)$
(%i8) fLs : reverse (fLs)$
(%i9) fll(fLs);
(%o9) [[1.8,54],[1.817,46],97]
```

We can redraw the scatterplot with the points having eruption durations less than 3 min being drawn in solid red color (same point size though) to have an easy visual look:

```
(%i10) qdraw ( pts (fL, ps(1),pc(black),pt(6)),  pts (fLs, ps(1),pc(red)) )$
```

which produces the plot

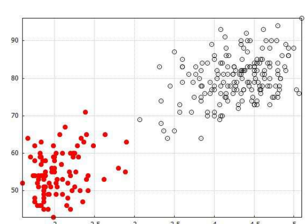

Figure 50: Old Faithful: Short Duration Times in Red

We can force a better range of values in both directions, and add a vertical line at the duration time of 3 min:

```
(%i11) qdraw ( xr(1,6),yr(40,100), line(3,40,3,100),
           pts (fL, ps(1),pc(black),pt(6)), pts (fLs, ps(1),pc(red)))$
```

which produces the plot

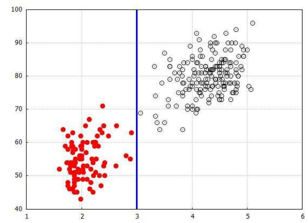

Figure 51: Old Faithful: Wait Times vs. Duration Times

We now produce a least squares fit of this data (assuming a linear fit) similar to the linear fit we carried out in Maxima by Example, Chapter 2. See our discussion there for an explanation of what we are doing here. The list of points for all duration times **fL** will be used to get the best fit straight line.

```
(%i12) load(lsquares);
(%o12) "C:/Program Files/Maxima-sbcl-5.36.1/share/maxima/5.36.1/share/lsquares/lsquares.mac"
(%i13) Mf : apply('matrix,fL)$
(%i14) row(Mf,1);
(%o14) matrix([3.6,79])
(%i15) length(Mf);
(%o15) 272
(%i16) soln : (lsquares_estimates(Mf, [x,y], y = a*x+b,
                   [a,b])), numer;
(%o16) [[a = 10.72964139513352,b = 33.47439702275336]]
(%i17) [a,b] : (fpprintprec:5, map('rhs, soln[1]));
(%o17) [10.73,33.474]
(%i18) qdraw ( xr(1,6),yr(40,100), line(3,40,3,100), key(bottom),
           pts (fL, ps(1),pc(black),pt(6)), pts (fLs, ps(1),pc(red)),
                   ex1(a*x + b,x,1,6,lc(magenta),lk("linear fit")))$
```

which produces the plot

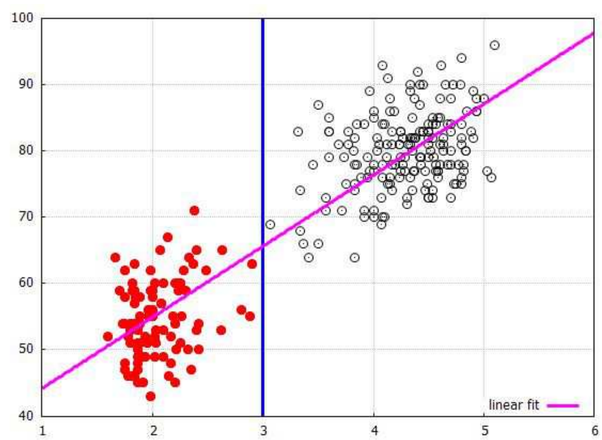

Figure 52: Old Faithful: Wait Times vs. Duration Times plus Linear Fit

10 Data Plots, Error Bars, Least Squares Fit

Two space-separated data files, **fit1.dat** and **fit2.dat**, are available for download or viewing with the Ch. 13 files on the author's webpage. We will use those two files to illustrate making simple data plots using the **qdraw** functions **pts(...)** and **errorbars(...)**.

You can print out the file content, using **printfile("c:/work2/fit1.dat")$**. We use Maxima's **read_nested_list** function to create a nested list of data points from the data file.

```
(%i3) plist : read_nested_list("c:/work2/fit1.dat");
(%o3) [[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]]
(%i4) length (plist);
(%o4) 10
```

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

```
(%i5) qdraw( pts(plist) )$
```

which produces size 3 blue filled circle point markers:

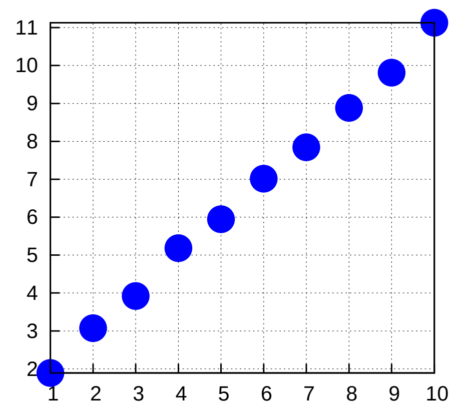

Figure 53: Using pts(...) Defaults

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

```
(%i6) qdraw( pts(plist, ps(2)), xr(0,12), yr(0,15) )$
```

with the result:

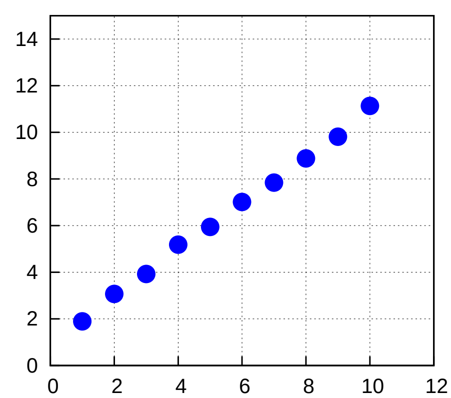

Figure 54: Adding ps(2), xr(..), yr(..)

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

```
(%i7) qdraw( pts(plist,pc(red),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars(plist, 1) )$
```

which shows thin error bars in the default black color:

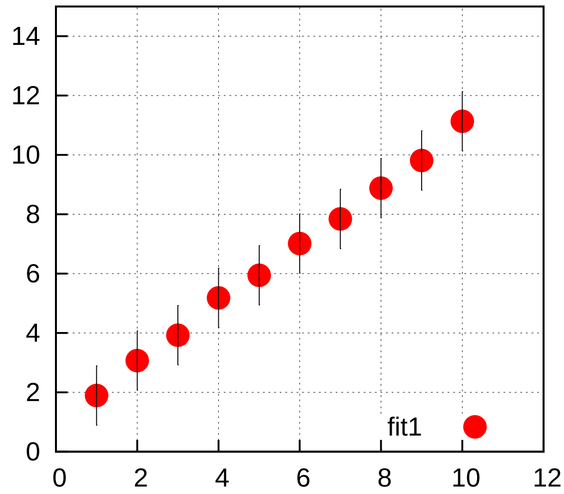

Figure 55: Adding pc(red) and Simple Error Bars

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

```
(%i8) qdraw( pts(plist, pc(red), pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, 1, lw(3), lc(blue) )  )$
```

with the result:

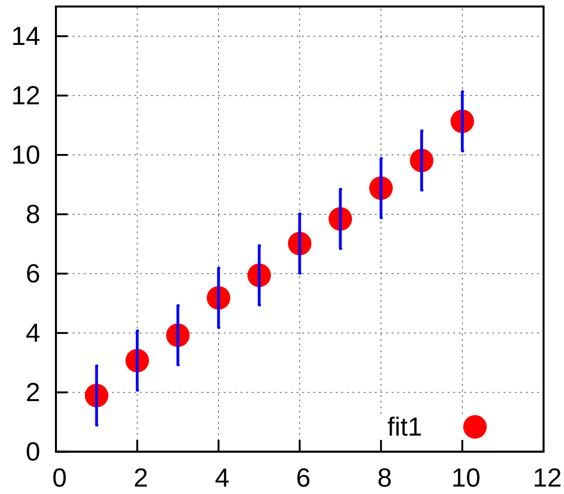

Figure 56: Adding lw(3), lc(blue) to errorbars(...)

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

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

Here is an example:

```
(%i9) dyL : [0.2,0.3,0.5,1.5,0.8,1,1.4,1.8,2,2];
(%o9) [0.2,0.3,0.5,1.5,0.8,1,1.4,1.8,2,2]
(%i10) map ('length,[plist,dyL] );
(%o10) [10,10]
(%i11) qdraw( pts (plist, pc(red), pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars (plist, dyL, lw(3), lc(blue)))$
```

with the result

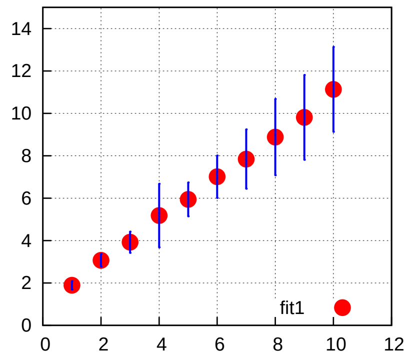

Figure 57: Using a list of dy values with errorbars(..)

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. Recall, from above, **`plist`** is a list of **`[x,y]`** pairs obtained from the data file **`fit1.dat`**.

```
(%i12) pmatrix : apply( 'matrix, plist );
(%o12) matrix([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])
(%i13) soln : (lsquares_estimates(pmatrix, [x,y], y = a*x+b,
                   [a,b])), numer;
(%o13) [[a = 0.9951478787878788,b = 0.9957666666666667]]
(%i14) [a,b] : (fpprintprec:5, map('rhs, soln[1]));
(%o14) [0.99515,0.99577]
(%i15) qdraw( pts(plist, pc(red),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, dyL, lw(3),lc(blue) ),
         ex1(a*x + b,x,0,12,lc(brown),lk("linear fit")))$
```

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

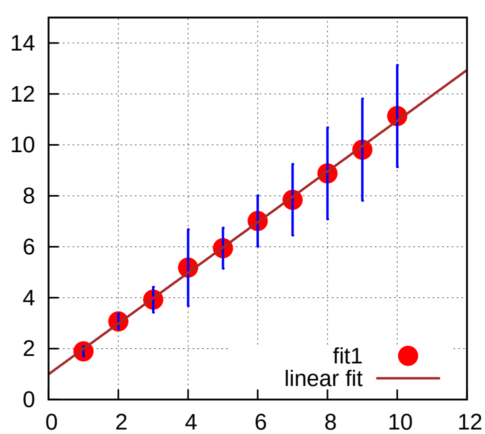

Figure 58: Adding the Linear Fit Line

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

```
(%i16) printfile("c:/work5/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
(%o16) "c:/work5/fit2.dat"
(%i17) p2list: read_nested_list("c:/work5/fit2.dat");
(%o17) [[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]]
(%i18) length(p2list);
(%o18) 10
(%i19) qdraw( pts(plist,pc(red),pk("fit1"), ps(2)), xr(0,12),yr(0,15),
          key(bottom), errorbars( plist, dyL, lw(3),lc(blue) ),
         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)))$
```

which produces the plot

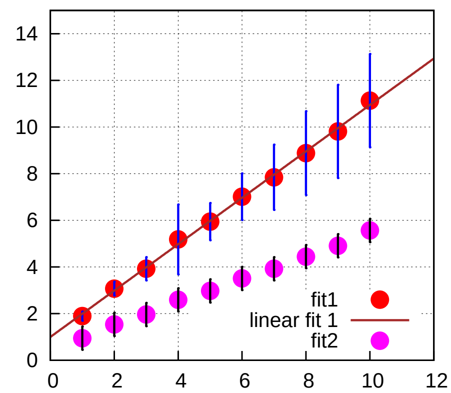

Figure 59: Adding the Second Set of Data

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.

11 Geometric Figures

11.1 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(red), lw(6), lk("radius"))** will draw a line from $(0, 0)$ to $(1, 1)$ in red with line width 6 and with a key entry with the text “radius”. The defaults are color blue, line width 3, and no key entry.

```
(%i3) qdraw( line(0,0,1,1) )$
```

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

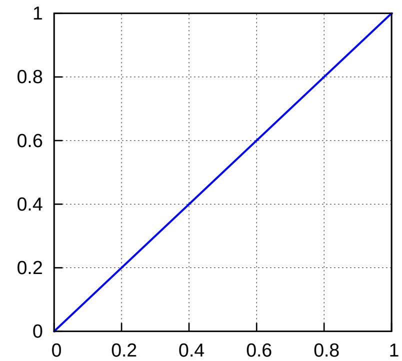

Figure 60: Default line(..)

Adding some options and extending the canvas range in both directions

```
(%i4) qdraw( line(0,0,1,1,lc(red),lw(6),lk("radius") ),
              xr(0,2),yr(0,2),key(bottom),
              pts([ [1,1] ] , ps(2), pc(blue), pk("point")) )$
```

produces a red line to a blue point:

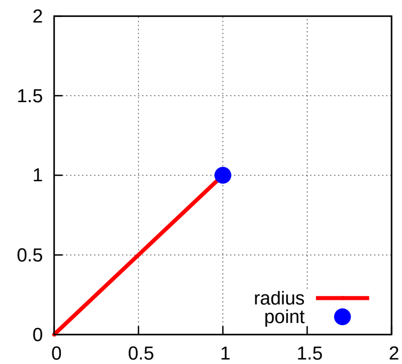

Figure 61: Adding options to line(..)

`qdraw.mac` contains the definition of the function **`doplot1(nlw)`** (nlw is the requested line width). The definition is:

```
doplot1(nlw) := block([cc,qlist,x,val ],
     /* 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))$
```

(Using **`append`** instead of **`cons`** is slower, but doesn't matter here.) Here use **`doplot1`** to produce a series of vertical colored lines.

```
(%i5) doplot1(10)$
```

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

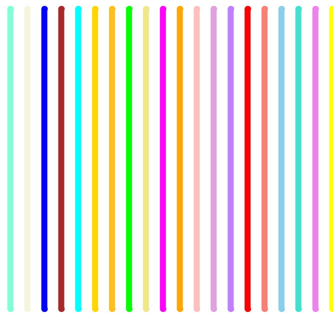

Figure 62: Using line(...) to Display Some Colors

11.2 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 default blue with line thickness 3 and with no fill color. An example with all three optional args is: **`rect(0,0,1,1,lc(brown),lw(2),fill(khaki) )`** .

We start with the basic default rectangle call:

```
(%i6) qdraw ( xr (-1,2), yr (-1,2), rect (0,0,1,1) )$
```

with the result

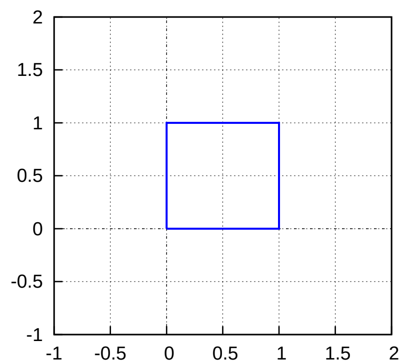

Figure 63: Default rect (0,0,1,1)

We now add some color, thickness and fill:

```
(%i7) qdraw( xr(-1,2),yr(-1,2),
            rect(0,0,1,1, lw(5), lc(brown), fill(khaki) ) )$
```

with the output:

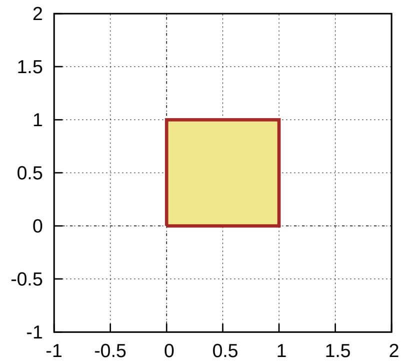

Figure 64: rect(0,0,1,1, lc(brown), lw(5), fill(khaki))

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

```
(%i8) 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:

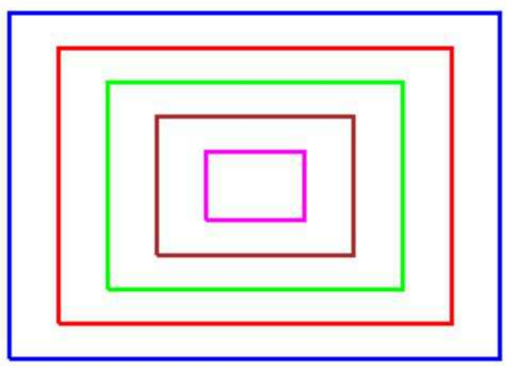

Figure 65: Nested Rectangles using rect(..)

11.3 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 connectd to the first.

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

```
(%i9) 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:

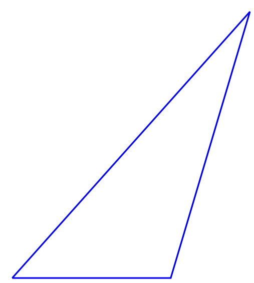

Figure 66: Default use of poly(...)

qdraw.mac contains the Maxima function **doplot2()** which will draw eighteen stacked right triangles in various colors:

```
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 we use **doplot2()**:

```
(%i10) doplot2()$
this is doplot2
```

with the resulting graphics:

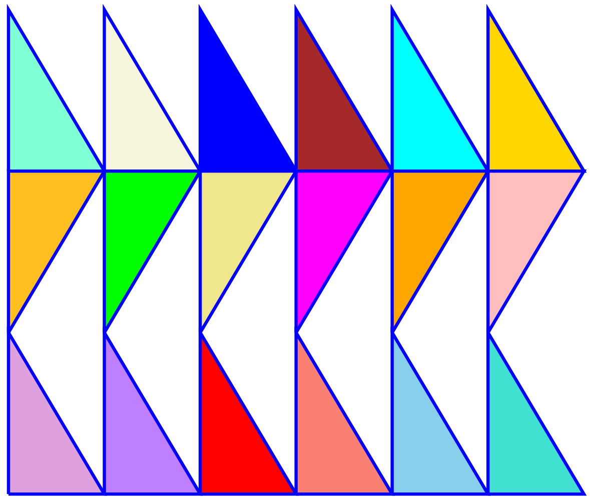

Figure 67: Using poly(...) with Color

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

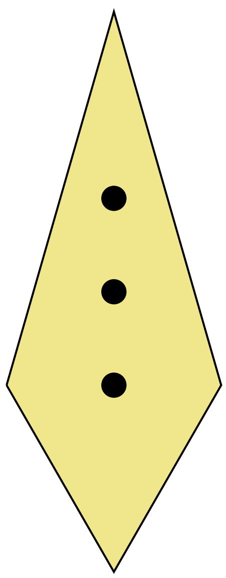

Figure 68: Homework Problem

11.4 circle(...) and ellipse(...)

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

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

which draws 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(...)** (although this is complicated by the size and configuration of the window used.

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

```
(%i11) qdraw(xr(-2,2),yr(-2,2),circle(0,0,1))$
```

which looks like

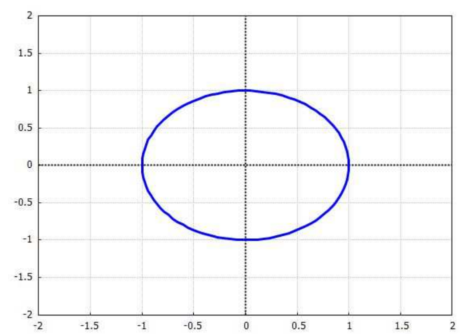

Figure 69: Default “circle”

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

```
(%i12) 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) ))$
```

with the result:

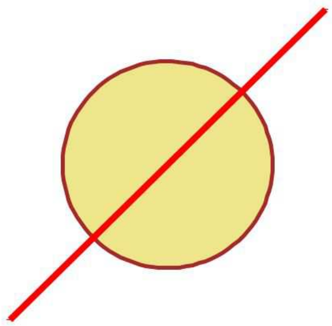

Figure 70: line over “round” circle

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:

```
(%i13) 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)))$
```

then the circle fill color will hide the line:

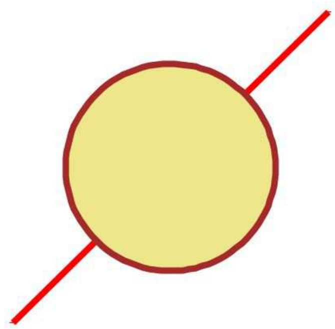

Figure 71: circle over line

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 blue, line width 3, and no fill color.

Here is the default **ellipse** behavior:

```
(%i14) qdraw( xr(-4.2,4.2),yr(-3,3),
        ellipse(0,0,3,2,90,270) )$
```

which produces

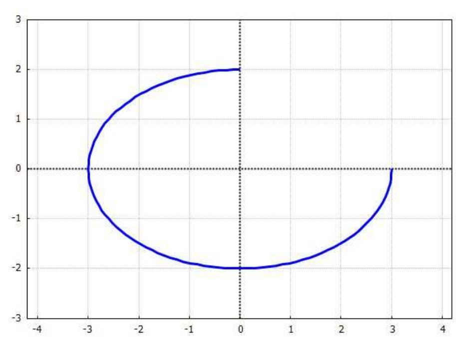

Figure 72: ellipse(0,0,3,2,90,270)

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

```
(%i15) qdraw( xr(-5.6,5.6),yr(-4,4),ex1(x,x,-4,4,lc(blue),lw(5)),
                  ex1(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 the plot

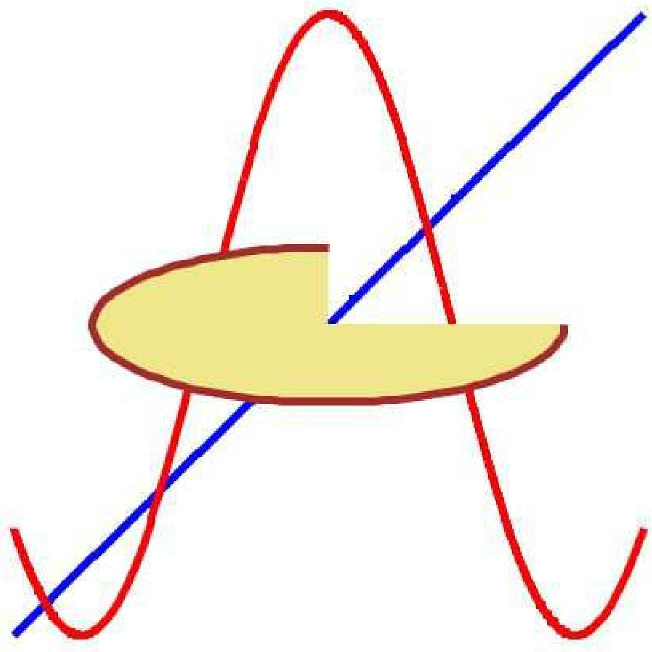

Figure 73: Filled Ellipse plus . . .

11.5 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 "not-filled"; use `ht(e)` for "empty", `ht(f)` for "filled," and `ht(n)` to change back to "not-filled."
Once one of the "head properties" has been changed in a call to **vector**, the next call to `vector` assumes the change is still in force.
The default line width is 3; use `lw(5)` to change to 5.
The default line color is blue; use, for example, `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:

```
(%i16) qdraw( xr(-2,2), yr(-2,2), vector( [-1,-1], [2,2] ) )$
```

with the result:

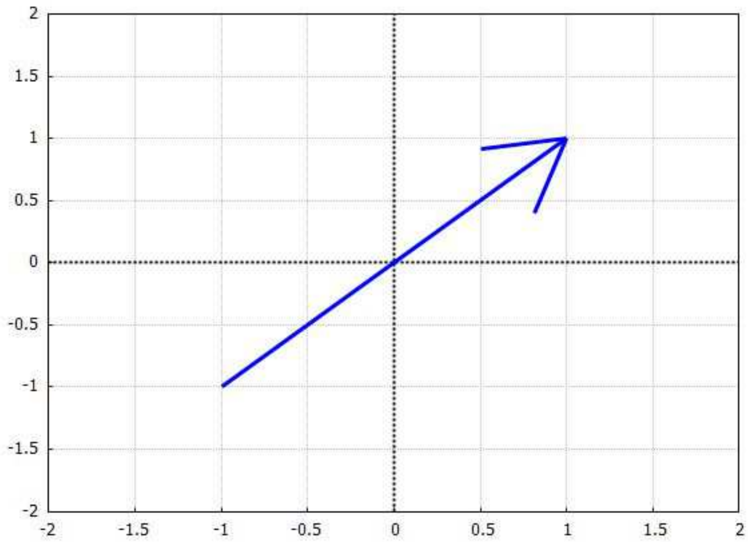

Figure 74: Default Vector

We can thicken and apply color to this basic vector with

```
(%i17) qdraw(xr(-2,2),yr(-2,2),
           vector([-1,-1],[2,2],lw(5),lc(brown),lk("vec 1")),
            key(bottom) )$
```

which produces

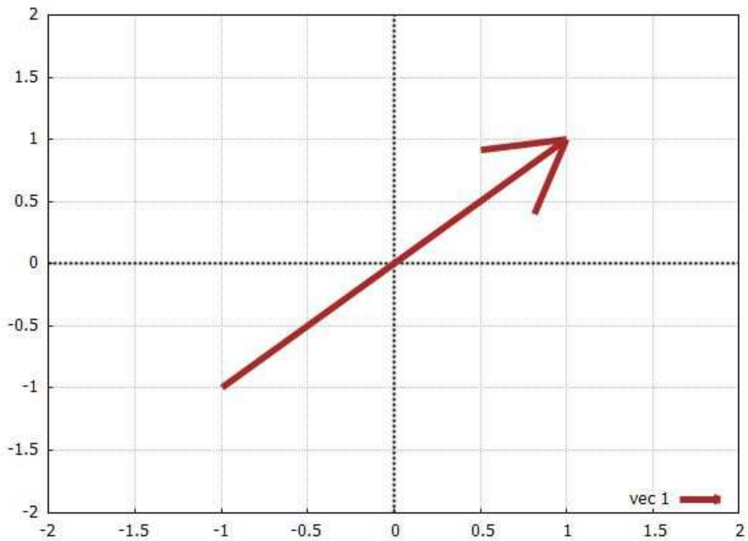

Figure 75: Adding Color, etc.

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.

```
(%i18) qdraw(xr(-2,2),yr(-2,2),
           vector([0,0],[1,1],lw(5),lc(brown),lk("vec 1"),
                  ht(e), hb(t), ha(45)), key(bottom))$
```

which produces:

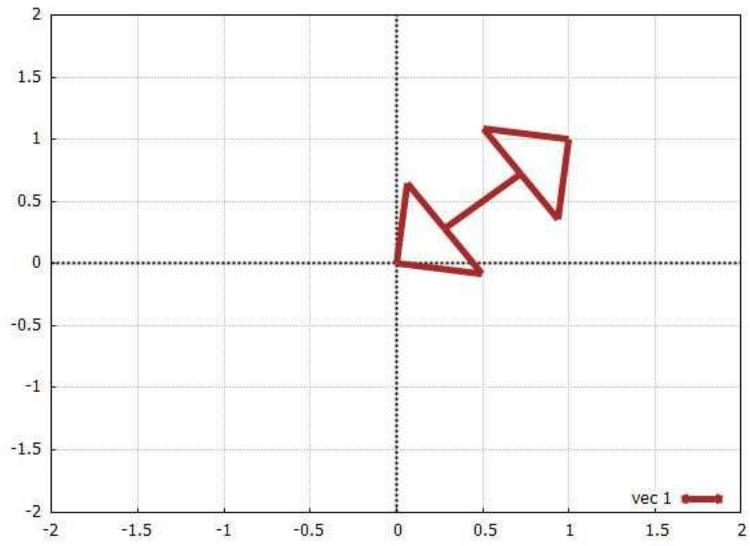

Figure 76: Changing Head Properties

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:

```
(%i19) qdraw(xr(-2.8,2.8),yr(-2,2),vector([0,0],[1,1],lw(5),lc(brown),lk("vec 1"),ht(e),
  hb(t),ha(45) ), vector([0,0],[-1,-1],lw(5),lc(magenta),lk("vec 2")),key(bottom) )$
```

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

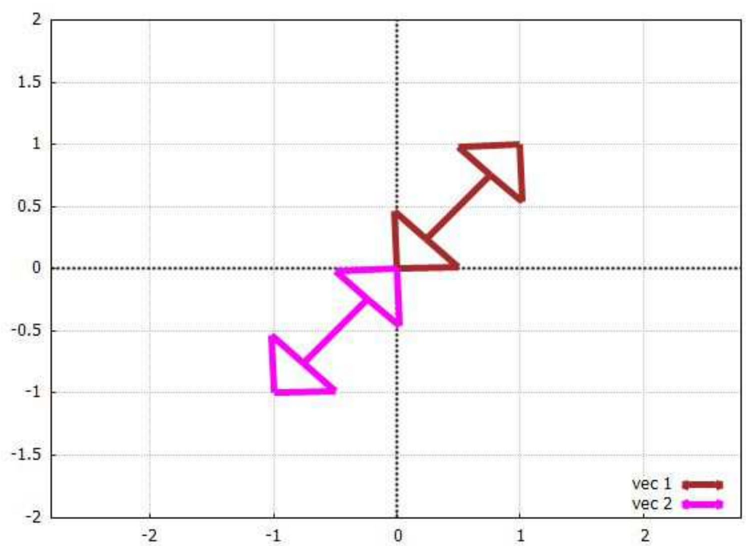

Figure 77: Head Properties Memory at Work

11.6 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 the 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 (blue) and line width (3).
The opening half angle is hardwired to be **phi = 25 deg = 0.44 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.

```
(%i20) 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) )$
```

which produces the plot:

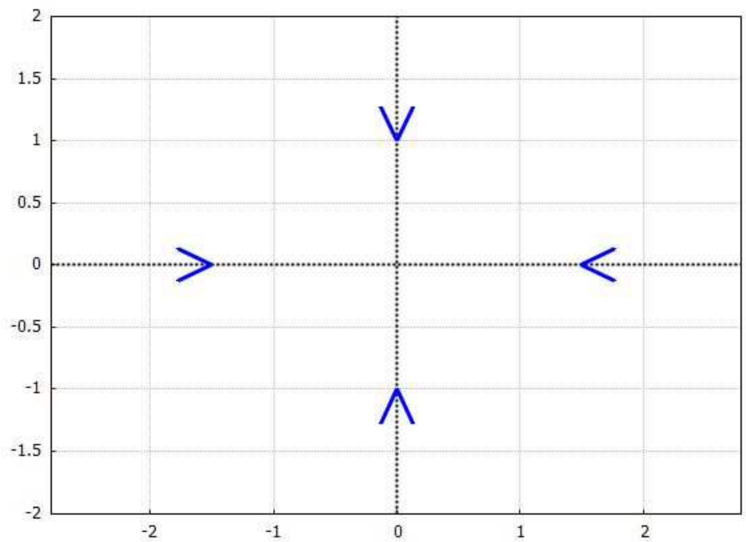

Figure 78: Default arrowhead(...) Examples

12 Greek Letters, Math Symbols, and Adjustable Font Size with Labels

The **qdraw.mac** function **label** can be used to place Greek letters, some mathematical symbols (and normal text) with adjustable font size, on your plots. This ability requires the use of a more elaborate "string" than we have used above. The default color used with **label(s,x,y)** is black. You can produce a label in your choice of color by using the syntax **label(s,x,y,lc(A-Color))**, as in **label("mytext",1,1,lc(blue))**, for example.

As a first example, we combine **line**, **ellipse**, **arrowhead**, and **label** to show an angle labeled with the Greek letter θ, as well as a small left pointing arrow and a normal text description as part of one label.

```
(%i3) qdraw(xr(0,4),yr(0,2),line(0,0,4,0,lc(black),lw(2)), line(0,0,2,2,lc(blue),lw(3) ),
             ellipse(0,0,1,1,0,45 ), arrowhead(0.707,0.707,135,0.15),
             label(["{/=36 {/Symbol q   \\254 }  The Angle}",1,0.4]),
             cut(all))$
```

which produces the plot (you may need to expand the window of the graphics to see all of the text description part of the label):

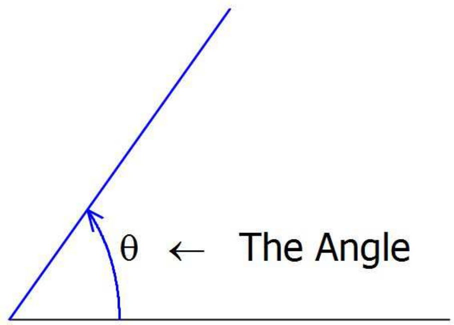

Figure 79: line(..), ellipse(..), arrowhead(..), label(..)

The syntax used was

```
    label ( [ String, x, y ])
```

in which

```
  String = "{ /=36  symbols-bracket  latin-text }"
```

and the beginning **{/=36** set the font size for both of the following items until a matching brace (**}**) is found. The **symbols-bracket** began with **/Symbol** and forced **draw** to use the symbol dictionaries which convert **q** to θ and convert **\\254** to $\leftarrow$.

The entry **{/Symbol q }** by itself, inside the string, would produce just the Greek letter θ. 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 matching pair of braces.

We can instead use **label** twice to get more control over the font size and position of the text **The Incline Angle**, as shown here

```
(%i4) 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(["{ {/Symbol=36 q   \\254 } }",1,0.4]),
                         label ( ["{ /=15 The Incline Angle}", 1.7, 0.42]))$
```

which produces the plot

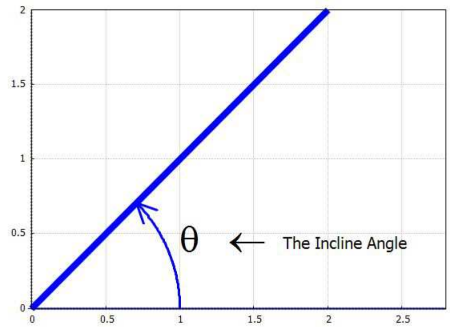

Figure 80: line(..), ellipse(..), arrowhead(..), label(..)

We can get exactly the same plot using only one **label** by using the two bracket syntax

```
label ( [String1,x1,y1], [String2, x2,y2] )
```

as in the following:

```
(%i5) 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(["{ {/Symbol=36 q   \\254 } }",1,0.4],
                                  ["{ /=15 The Incline Angle}", 1.7, 0.42] ))$
```

A summary of advanced use of the draw package functions is found at
http://riotorto.users.sourceforge.net/Maxima/gnuplot/index.html.
If you use the link to
http://riotorto.users.sourceforge.net/gnuplot,
at the top of the introduction to the draw package in the Contents section of the Maxima html help manual, you are taken to a revised link and finally to the same contents as above, although inside the Maxima html Help manual.

You can then find several examples of the use of Greek letters if you click on the successive links
Graphics objects, label, enhanced text.

As a second example, we write the equation $P = \rho\, k\, T$ using labels as shown in this example, using three different font sizes and also switching from the default font to the Helvetica font.

```
(%i6) qdraw (xr(-3,3),yr(-3,3), label (["P = {/Symbol r}kT",-1,1]),
                 label (["{/Helvetica=18 P = {/Symbol r}kT}",1,1]),
                  label (["{/Helvetica=24 P = {/Symbol r}kT}",1,-1,lc(blue)]))$
```

which produces the figure

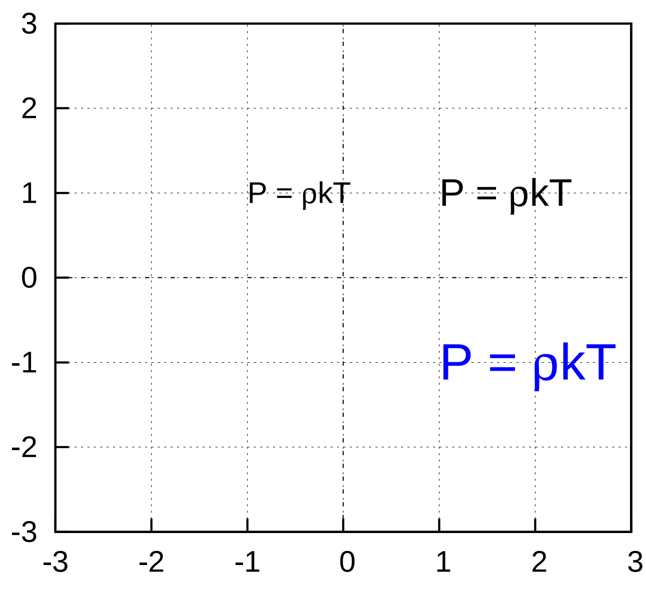

Figure 81: Font and Greek Examples

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

```
(%i7) qdraw(xr(-5,5),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))$
```

which produces (you may need to expand the gnuplot window to see the complete graphic):

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

α β χ δ ε ϕ γ η ι φ κ λ μ

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

ν ο π θ ρ σ τ υ ϖ ω ξ ψ ζ

Figure 82: Lower Case Latin to Greek

To see the complete graphic, you will need to increase the inline graphics window temporarily if you are using **`wxqdraw`** with the **`wxMaxima`** interface, as in

```
(%i3)  wxplot_size;
(%o3)  [600,400]
(%i4)  wxqdraw(xr(-5,5),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)), wxplot_size = [1024,768]$
```

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

```
(%i8) 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))$
```

which produces (again, you may have to widen-drag the Gnuplot window to see the whole graphic)

A B C D E F G H I J K L M

Α Β Χ Δ Ε Φ Γ Η Ι ϑ Κ Λ Μ

N O P Q R S T U V W X Y Z

Ν Ο Π Θ Ρ Σ Τ Υ ς Ω Ξ Ψ Ζ

Figure 83: Upper Case Latin to Greek

Useful mathematical character codes, consisting of three numbers preceded by a double backslash, are
`\\243` $\leq$ (less than or equal)
`\\245` ∞ (infinity symbol)
`\\253` $\leftrightarrow$ (double ended arrow)
`\\254` $\leftarrow$ (left arrow)
`\\256` $\rightarrow$ (right arrow)
`\\261` $\pm$ (plus or minus)
`\\263` $\geq$ (greater than or equal)

`\\264` $\times$ (times)
`\\271` $\neq$ (not equal)
`\\273` $\approx$ (approx equal)
`\\345` $\sum$ (summation sign)
`\\362` $\int$ (integral sign)

We can use `label` to make a graphics table of available mathematical symbols. The use of `&{junk}` inside the string creates empty space corresponding to the number of characters inside the braces.

```
(%i9) s1 : "{/=36 243   {/Symbol \\243} &{abcd} 254   {/Symbol \\254} &{abcd} 263
{/Symbol \\263} &{abcd} 273   {/Symbol \\273}}"$
(%i10) s2 : "{/=36 245   {/Symbol \\245} &{abcd} 256   {/Symbol \\256} &{abcd} 264
{/Symbol \\264} &{abcd} 345   {/Symbol \\345}}"$
(%i11) s3 : "{/=36 253   {/Symbol \\253} &{abcd} 261   {/Symbol \\261} &{abcd} 271
{/Symbol \\271} &{abcd} 362   {/Symbol \\362}}"$
(%i12) qdraw(xr(-3,3),yr(-2,2),label([s1,-2,1]), label([s2,-2,0]), label([s3,-2,-1]),cut(all))$
```

which produces the figure:

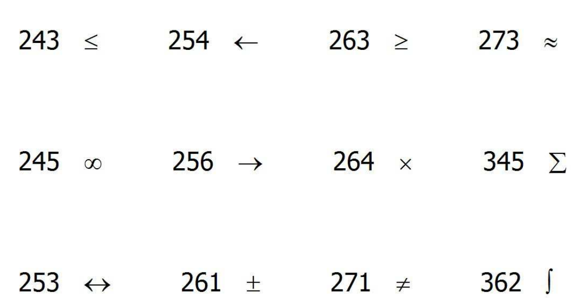

Figure 84: Useful Character Code Symbols

You Can Convert Latin to Greek inside the Gnuplot Window

You can convert Latin letters to Greek as follows. First produce a graphic in the gnuplot window, in which the angle is labeled with the letter **q**:

```
(%i13) 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) )$
```

which produces the plot

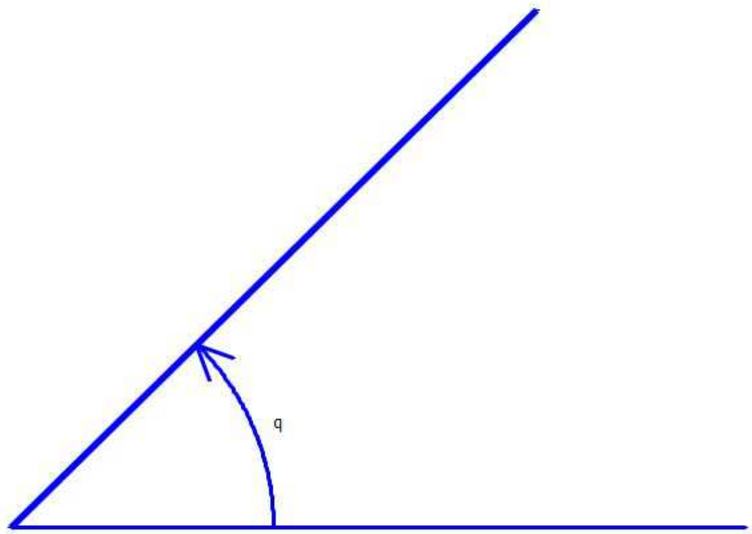

Figure 85: Start with **q** as angle label

When the gnuplot window appears, click on the Options icon and then click on Font. In the left Font panel, choose **`Graecall`** font, from the middle panel choose **`regular`**, and click on size 36 in the right panel and click **`ok`**. The English letter **`q`** (lower case) is then converted to θ. You then have the graphic Gnuplot window with

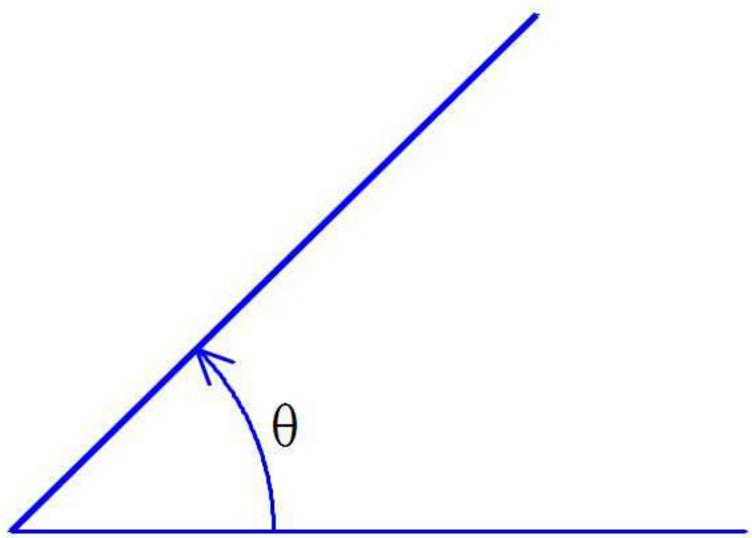

Figure 86: options,font, Graecall converts **q** to θ

In the Gnuplot window, copy the new graphic with the Greek letter θ labeling the angle to the Clipboard and then open an image viewer. In the freely available Irfanview program, if you use Edit, Paste, the clipboard image appears inside Irfanview, and you can then save the image as a jpeg (.jpg) file in your choice of folder.

You can convert the jpg graphics file to an eps graphics file using the freely available Cygwin program, with the command

```
convert myfile.jpg  myfile.eps
```

13 Even More with more(...)

You can use the **qdraw** function **more(...)**, containing some legal **draw2d** elements, (which we used above when presenting examples of **ex(...)** and **ex1(...)**, etc.) as we illustrate here by adding an x-axis label and a title. This is done by using **`more(...)`** with two legal **`draw`** arguments.

```
(%i3) qdraw( ex([x,x^2,x^3],x,-2,2),
          more(xlabel = "X AXIS", title="intersections of x, x^2, x^3" ),
                   cut(key),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)))$
```

which produces

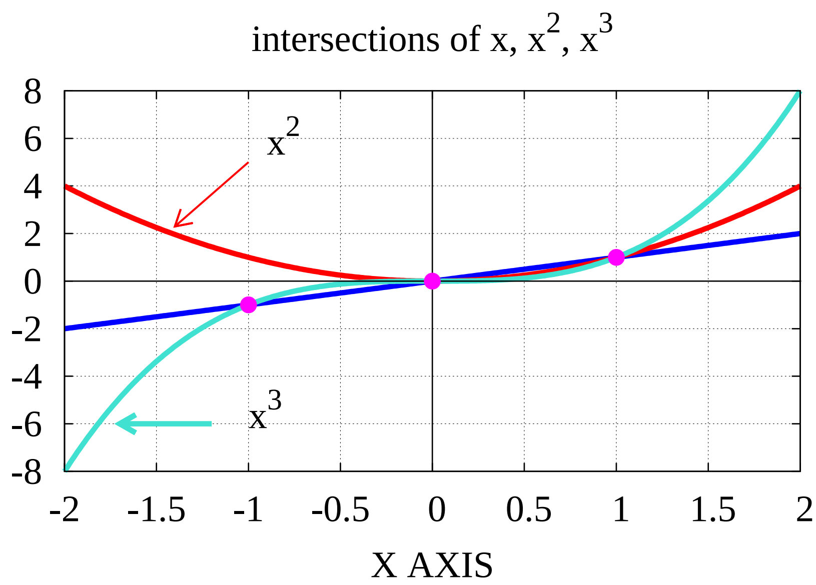

Figure 87: Using more(...) for title and x-axis Label

14 Basic Elements of the draw Program

From the webpage

```
http://riotorto.users.sourceforge.net/Maxima/gnuplot/index.html
```

we provide a list of the links which lead to more information about the basic elements of the **draw** package.

14.1 Introduction

```
Introduction

This is a Maxima-Gnuplot interface.

There are three functions to be used at Maxima level: draw2d, draw3d and draw.
To read the available documentation about functions, variables and graphic options,
     type, for example, ? point_type for information about point_type, etc.

More or less, this package works as follows. Scenes are described in gr2d or
     gr3d objects, which are then passed to function draw. If more than one
         scene is described, a multiplot will be generated,
         as in draw(gr2d(...),gr3d(...)) but if you want only one scene,
         draw2d(...) and draw3d(...) are equivalent to draw(gr2d(...)) and
         draw(gr3d(...)), respectively. See examples bellow.
```

14.2 Graphic objects

```
Graphic objects

Click on the items below to see examples of graphic objects plotted
      with the VTK libraries.

    bars
    cylindrical
    elevation_grid
    ellipse
    errors
    explicit
    geomap
    image
    implicit
    label
    mesh
    parametric
    parametric_surface
    points
    polar
    polygon
    rectangle
    region
    spherical
    triangle
    tube
    vector
```

14.3 Global options

```
Global options

Global options are those which are related to the whole plot.
                   They can be written anywhere in the scene description.

allocation. Used in multiplots. Some examples: (1, 2)
axis_3d. Removes all the axes in 3D scenes. Example: (1, 2, 3)
axis_top, axis_right. Show and hide axes. Some examples: (1, 2, 3)
background_color. Sets the background color. Example: (1, 2, 3, 4, 5)
cbrange. Sets the range of the color box. Example: (1)
cbtics. Sets the tics of the color box. Example: (1)
colorbox. Shows and hides the color box. Some examples: (1, 2, 3, 4)
columns. Number of columns in multiplots. Some examples: (1, 2, 3, 4)
contour. Plots contour lines on explicit surfaces. Some examples: (1)
contour_levels. Defines the levels to be plotted. Some examples: (1)
delay. Used in animations. Some examples: (1, 2, 3)
dimensions. Sets the dimensions of the plot in format [width, height].
                       Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
file_name. Sets the name of the graphic file. Some examples: (1, 2, 3, 4, 5)
font. Sets the font type. Example: (1, 2, 3, 4)
font_size. Sets the size of the fonts. Example: (1, 2, 3, 4)
grid. Used to draw grid lines on the plane. Some examples: (1, 2, 3, 4, 5, 6, 7, 8)
logcb. Sets the logarithmic scale in the color box. Example: (1)
logx, logy, logz. Indicates which axes must be transformed in
logarithmic scales. Some examples: (1, 2, 3)
palette. In 3D scenes, selects the palette. Some examples: (1, 2, 3, 4, 5, 6)
proportional_axes. Proportional axes. Example: (1, 2, 3, 4, 5, 6, 7)
surface_hide. Hides the surface. Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
terminal. By default, the output terminal is the screen.   Other terminals are:
     PNG (1, 2), SVG (1, 2), WXT (1, 2, 3, 4, 5),  EPS (1, 2, 3, 4),
     EPS_COLOR (1, 2, 3, 4, 5, 6), GIF (1),  animated GIF (1, 2, 3, 4),
     Multimode plots (1), EPSLATEX_STANDALONE (1)
title. Writes a title on the scene. Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
user_preamble. Let’s the user write his own Gnuplot code. Some examples: (1, 2, 3)
view. Positions the viewer in 3D scenes. Some examples: (1, 2, 3)
xaxis, xaxis_secondary, yaxis, yaxis_secondary, zaxis. Show and hide the axes.
              Some examples: (1, 2, 3, 4, 5)
xaxis_color, yaxis_color, zaxis_color. Set axes colors. Some examples: (1, 2)
xaxis_type, yaxis_type, zaxis_type. Set axes types. Some examples: (1, 2)
xaxis_width, yaxis_width, zaxis_width. Set axes widths. Some examples: (1, 2)
xlabel, ylabel, zlabel. Sets axes labels. Some examples: (1, 2, 3, 4, 5, 6, 7)
xrange, yrange, zrange. Sets the ranges of the axes. Example: (1, 2, 3, 4)
xtics, xtics_secondary, ytics, ytics_secondary, ztics. Show and hide axes tics.
                Some examples: (1, 2, 3, 4, 5, 6, 7)
xtics_axis. Show and hide axes tics. Some examples: (1, 2)
xtics_rotate, ytics_rotate. Rotates tics. Example: (1)
xy_file. Name of file where coordinates are saved. Example: (1)
```

14.4 Local Options

```
Local options

Local options are those which affect the appearance of individual graphic objects.
          They must be declared before the objects in the scene description.

    border. Shows and hides borders of polygons and ellipses. Example: (1, 2, 3)
    capping. Declares if circles must be drawn at the extrems of a tube. Example: (1)
    color. Sets the plotting color.
          Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14)
    enhanced3d. In 3D scenes, defines a colored pattern to project on a surface.
          Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
    error_type. Option for error plots. Some examples: (1)
    filled_func. Indicates whether a 2d function must be filled or not.
          Default is not (false). Some examples: (1, 2, 3)
    fill_color. Sets the color to fill an area.
          Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    fill_density. Sets the color density. Example: (1, 2)
    head_both. Option for vectors. Example: (1)
    head_angle. Option for vector heads. Example: (1)
    head_length. Option for vector heads. Example: (1, 2, 3)
    key. Defines the label of an object to be written in the legend.
          Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
    label_alignment. Sets label alignment. Example: (1, 2)
    label_orientation. Sets label orientation. Example: (1)
    line_type. Sets the type of lines. Example: (1, 2, 3, 4)
    line_width. Sets the width of lines. Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9)
    nticks. Declares the number of points to be calculated for plotting curves.
          Some examples: (1, 2, 3, 4, 5, 6)
    points_joined. true, false, or impulses. Some examples: (1, 2, 3, 4, 5)
        point_size. Sets the size of points. Some examples: (1, 2, 3)
    point_type. Sets the type of points. Some examples: (1, 2, 3, 4, 5)
    transform. Allows to perform geometric transformations.
          Example: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13)
    transparent. Makes 2D polygons transparent. Example: (1, 2, 3, 4, 5, 6, 7)
    unit_vectors. Plot vector of unit length. Example: (1)
    x_voxel, y_voxel, z_voxel. Sets the resolution in 2D regions and implicit
          3D objects. Example: (1, 2, 3, 4)
    xu_grid, yv_grid. Defines the resolution for plotting surfaces.
          Some examples: (1, 2, 3, 4, 5, 6, 7, 8, 9)
```

15 Programming Homework Exercises

General Comments

The file **qdraw.mac** is a text file which you can modify with a good text editor such as the freely available **notepad++**. 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 parenthsis is a fatal error which can nevertheless creep in; this may happen if you delete a debug printout placed inside and at the end of a do loop.

You may find it useful to insert some special debug printouts, such as **`print("in blank, a = ",a)`** or **`display(a)`**, perhaps in the middle (or the end) of a do loop:

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

When you are finished debugging a section, you can either comment out the debug printout or delete it to clean up the code.

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 all 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. Looking at this list is an excellent way to debug this program.

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.

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. When you want to edit your previous "try", use **`Alt-p`** to enter your previous code entry, and backspace over either the **`;`** or **`$`** which ends the code. You can then left-cursor and up-cursor to an area where you want to add a new line of code, and with the cursor placed just after a comma, press **`ENTER`** to create a new (blank) line. Since the block of code has not been properly concluded with either a **`);`** or **`)$`**, Maxima will not try to "run" 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`** cause Maxima to execute the entry.

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, or use the two key command **`SHIFT-END`**.

Then press **`CNTRL-C`** to copy the selected code to the 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.

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 "scan 3" 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 **notepad++**, **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**. You should first experiment with **draw2d** directly, as in

```
(%i3) draw2d( line_width = 5,
              line_type = dots,
             explicit(1 + x^2,x,-1,1),
             line_type = solid, /* default */
             explicit(2 + x^2,x,-1,1))$
```

which produces

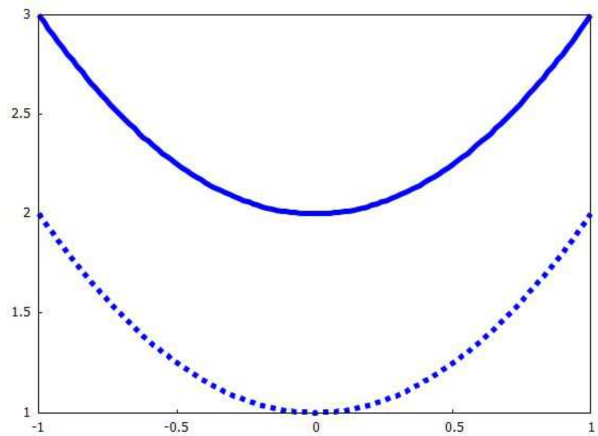

Figure 88: solid and dot choices for line type

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(f)** in the **qdensity** function (**f** for **false**). 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. You should start by experimenting first directly with **draw2d**.

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".

16 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** page

```
http://riotorto.users.sourceforge.net/Maxima/gnuplot/index.html
```

These examples go far beyond the simple graphics in this chapter.