MetaPlot, MetaContour, and Other Collaborations with METAPOST

Brooks Moses
Mechanical Engineering
Building 520
Stanford University
Stanford, CA 94305
USA
bmoses@stanford.edu

Abstract

Most methods of creating plots in METAPOST work by doing all of their calculations in METAPOST, or by doing all of their calculations in a preprocessing program. There are advantages to dividing the work more equitably, by doing the mathematical and data-visualization calculations in a preprocessing program and doing the graphical and layout calculations in METAPOST. The MetaPlot package provides a standard, flexible, interface for accomplishing such a collaboration between programs, and includes a general-purpose set of formatting macros that are applicable to a wide range of plot types. Examples are shown of linear plots with idiosyncratic annotation and two-dimensional contour plots with lines and filled contours on a non-Cartesian mesh.

1 Introduction

One of the challenges of scientific writing in TEX (or in LATEX) is producing figures that are of comparable quality to the typesetting. These figures often include plots and graphs that represent mathematically-intense visualization of large data files, implying that some form of specialized program must be used to create them. They also typically contain labels, notes, and other text that should be typeset in a manner consistent with the rest of the document, which requires using TEX's typesetting engine.

Traditionally, programs that meet these goals have taken one of two approaches. The first approach, used by programs such as ePiX [1] and Gnuplot [2], is to implement the program in a "traditional" programming language such as C++ or Fortran, and produce the complete figure as output in TEX/eepic or METAPOST code, which is then post-processed. The other approach, taken by METAPOST's `graph` package and m3D [3], is to implement the program directly in METAPOST's macro language.

There are advantages and tradeoffs to both of these approaches. Programming in METAPOST allows one to work directly with the language features such as declarative equations and ability to measure the size of typeset text, and thus to specify the figure layout in an intuitive, simple, and flexible manner. On the other hand, programming in a traditional language allows one to write mathematically-intensive programs that use floating-point numbers and can be compiled rather than run slowly through an interpreter; in addition, it may allow one to take advantage of existing visualization libraries, or to provide an interactive user interface.

This paper describes an intermediate approach, which combines the benefits of METAPOST and traditional-language programs. The initial data processing is done with a program written in a traditional language, which produces a METAPOST source file containing the processed data in an encapsulated form. This processed data is then fed into a set of METAPOST formatting macros, and the scaling, drawing, and annotation of the plots is all done by user-written commands within METAPOST.

Creating plots in two steps in this manner has several advantages:

- The initial data visualization can be done in a special-purpose program that uses a programming language and code libraries intended for substantial computations, with no need to implement more than a very simple output routine.
- The METAPOST macros for formatting plots and arranging them within a figure are largely independent of the details of the plots they are working with, and can be written in a generic manner suitable for widespread distribution.
- The layout of any given figure can be done using the same processes as a native-METAPOST

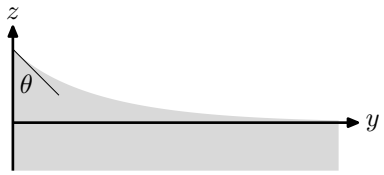

Figure 1: A capillary surface on a liquid touching a solid wall, after Batchelor [4].

drawing.

2 A simple example

Consider, by way of example, a plot of the shape of a meniscus formed by a liquid surface meeting a solid wall, as shown in Figure 1. The surface curve is given by a somewhat complicated expression involving inverse hyperbolic cosines,[1] and is representative of calculations that would be easier to do in a traditional programming language.

The C++ program to produce this curve in a METAPOST format is straightforward. The most complicated part is the function to generate a string containing a METAPOST representation of a point, which we accomplish using the <*sstream*> standard library.

```
string mpoint(double x, double y) {
      ostringstream pointstring;
  pointstring.setf(ios_base::fixed, ios_base::
      floatfield);
  pointstring.precision(5); pointstring << '(' << x
      << ',' << y <<
  ')'; return pointstring.str(); }
```

The **setf** and **precision** commands set the numeric format for the stream (fixed-precision, five decimal places), and then the coordinates are fed into the **ostringstream** with the appropriate punctuation, producing a result like `(0.01556,0.75006)`.

Given this and a *capillary*() function that computes the equation for the surface, creating the METAPOST command for the curve is simply a matter of looping through the points and dumping them to the standard output, with appropriate text before and after the loop to define the picture variable and close the curve into a cyclic path.

```
int main() { double theta = pi/4.0; double d =
      1.0; double h =
  sqrt(2.0 * d*d * (1.0 - sin(theta))); double y, z;

  cout << "picture capillary;\n"; cout << "
      capillary :=
  nullpicture;\n"; cout << "addto capillary
      contour " << mpoint(0.0,
  h); for(int i = 99; i > 2; i-- ) { z = (i/100.0) * h;
      y =
  capillary(z,h,d); cout << " .. " << mpoint(y, z); }
      cout << " -- "
  << mpoint(y, -0.5); cout << " -- " << mpoint
      (0.0, -0.5); cout << "
  -- cycle;\n"; }
```

This produces the following METAPOST code as output:

```
picture capillary; capillary :=
    nullpicture; addtocapillary contour
    (0.00000,0.76537) .. (0.00772,0.75771)..
    (0.01556,0.75006)
        % [ ... and so forth ... ]
.. (3.39322,0.02296) --
    (3.39322,-0.50000) --(0.00000,-0.50000)
    -- cycle;
```

We can then follow this with additional METAPOST commands to scale the figure to an appropriate size for printing on the page, and draw axes and labels, in order to produce the plot shown in Figure 1.

```
beginfig(1) draw (capillary scaled 0.5in) withcolor
  0.85white; linecap := butt; pickup pencircle scaled
      1pt; drawarrow
  (0,-0.25in) -- (0, 0.5in); label.top(btex $z$ etex
      ,(0,
  0.5in)); x1 := (xpart(lrcorner capillary) * 0.5in, 0)
      + (0.1in, 0);
  drawarrow (0,0) -- x1; label.rt(btex $y$ etex, x
      1);
  pickup pencircle scaled 0.25pt; x2 := ulcorner
      capillary scaled
  0.5in; draw ((0,0) -- (0.24in, -0.24in)) shifted x2;
      label(btex
  $\theta$ etex, x2 + (0.07in, -0.18in)));
  endfig; end
```

Although this example produces a perfectly serviceable result, it has some noteworthy drawbacks. The scale factor of 0.5 in does not have a clear relationship to the size of the plot, and producing a plot of a particular size would require measurement of the *capillary* picture and explicit computation of the scale factor. The locations of the annotations

[1] For those who are curious, the equation (from [4]) is

$$\frac{y}{d} = \cosh^{-1}\frac{2d}{z} - \cosh^{-1}\frac{2d}{h} + \left(4 - \frac{h^2}{d^2}\right)^{\frac{1}{2}} - \left(4 - \frac{z^2}{d^2}\right)^{\frac{1}{2}},$$

where $h^2 = 2d^2(1 - \sin\theta)$ is the height of the meniscus, θ is the contact angle, and d is a scaling parameter related to the surface tension and liquid density.

are likewise determined by explicit measurement, or by being typed in directly. If we were to change one of the parameters in the C++ program and re-run it, many of the values in the METAPOST code would need to be changed as well.

3 A more general example: The MetaPlot package

The MetaPlot package is designed to address many of the shortcomings of the example given in Section 2. It provides a consistent way of transferring the plot commands and associated metadata from the generating program into METAPOST, and direct handles for manipulating the plots within METAPOST using its normal idiom of declarative equations rather than procedural assignments.

To accomplish this in a general manner, we define two types of METAPOST data structures: *plot objects* and *plot instances*. A plot object is a plot "in the abstract", containing paths, filled contours, and metadata that make up the plot (or a set of related plots), represented in a manner that is independent of the details of how the plot is positioned. By contrast, a plot instance is a plot "on the page", containing parameters for the scaling and positioning of a given plot, and a reference to a parent plot object that gives the actual pictures to be drawn.

A typical preamble for a figure using MetaPlot will consist of an **input** `metaplot` command to load the MetaPlot macros, an **input** command to load the METAPOST file that contains the plot objects (typically an output file from the preprocessing program), and calls to the MetaPlot macros to generate plot instances from the plot objects.

3.1 The concept of a plot-object

Suffix arguments and multi-token variable names in METAPOST allow us to define data structures that approximate structures or objects in more traditional programming. The correspondence is not exact; in particular, there is no data type associated with the overall object. METAPOST is simply passing around a fragment of a variable name and constructing complete variable names from it, so any arbitrary element can be added to the class without changing its type. Thus, the MetaPlot macros can deal with arbitrary types of plots in a generic manner, so long as they meet a few minimal requirements that allow them to be scaled and positioned.

The paths and contours that make up a plot object are not defined in terms of the native data coordinates, but are rescaled to fit within a unit box (that is, extending from 0 to 1 in both coordinate directions), which is treated as the bounding box of the plot for purposes of scaling and positioning.[2] As a result, the possibility of coordinates too small or too large for METAPOST's fixed-point number representation is avoided; in addition, positioning the plot on the page is a simple matter of scaling by the final width and height and shifting by the final position of the lower-left corner. The original data scales are stored in four numeric components that record the values corresponding to the extents of the bounding box, and can be used later to rescale the plot to an appropriate size and aspect ratio.[3]

The remaining details of the format can be shown by rearranging the example from Section 2 into a plot object, as follows. For purposes of later examples, we will presume that this has been saved as `capillary.mp`.

```
% Definition of capillary plot-object
% Picture components
picture capillary.fplot; capillary.fplot := nullpicture;
      addto
  capillary.fplot contour (0.00000,1.00000) ..
        (0.00227,0.99395)
  .. (0.00459,0.98790)
        % [ ... and so forth ... ]
  .. (1.00000,0.41329) -- (1.00000, 0.00000) --
        (0.00000, 0.00000) --
  cycle; picture capillary.lplot; capillary.lplot :=
        nullpicture;
  addto capillary.lplot doublepath
        (0.00000,1.00000)
  .. (0.00227,0.99395)
        % [ ... and so forth ... ]
  .. (1.00000,0.41329);

% Required metadata
numeric capillary.xleft; capillary.xleft = 0.0;
      numeric
capillary.xright; capillary.xright = 3.39322; numeric
      capillary.ybot;
capillary.ybot = -0.5; numeric capillary.ytop;
      capillary.ytop =
0.76537;

% Plot-specific metadata
pair capillary.contactpoint; capillary.contactpoint =
      (0.0, 1.0);
```

[2] Using a box from −4096 to +4096 would make better use of METAPOST's fixed-point number range, but even on a unit box expanded to a 2-meter-wide poster, the granularity is only 0.03 mm — which is better than most printers. A larger box is probably not worth the inconvenience.

[3] Although these variables are represented here as numerics and thus are still vulnerable to under- or overflow, it would be a simple matter to replace them with string-represented numbers from the `sarith` package.

```
numeric capillary.contactangle; capillary.
        contactangle = 45.0;
```

In this case, I have also added an additional component: this version of *capillary* contains a path for the liquid surface line (*capillary.lplot*), as well as the original filled contour (now *capillary.fplot*); the decision about which of them to draw can be made later. A plot object can contain any number of these pictures (even zero), with arbitrary names.

The four required scale variables are *capillary.xleft*, *.xright*, *.ybot*, and *.ytop*; these, for purposes of the MetaPlot macros, must be named thus.

Finally, there are two metadata variables for this particular plot, *capillary.contactpoint* and *capillary.contactangle*, which will be useful in drawing the annotations. Any number of additional variables can be present, with arbitrary names. Of note is that *.contactpoint* is given in the same unit-box coordinate system that the paths and contours are in, allowing it to be positioned by the same macros that scale and position the picture components.

3.2 Creation of a plot instance

The next step after creating plot objects is manipulating them on the page by means of plot instances. A plot instance thus needs to contain three sets of components: coordinates and dimensions of the plot as shown on the page, a representation of the plot's internal scale for use in alignment and producing axes, and a means of accessing picture components from its parent plot object. These are created by the *plot_instantiate*() macro, which is part of MetaPlot; the version below is simplified somewhat.

```
% Args: inst is the new plot instance.
% plot_object is the parent plot object.
def plot_instantiate(suffix inst)(suffix plot_object) =

% Define (unknown) parameters for plot-instance
% location on page
  numeric inst.pagewidth, inst.pageheight; numeric
          inst.pageleft,
  inst.pageright, inst.pagetop, inst.pagebottom; inst.
          pageleft +
  inst.pagewidth = inst.pageright; inst.pagebottom +
          inst.pageheight =
  inst.pagetop;

% Define (known) parameters for plot's scaling
  numeric inst.scaleleft, inst.scaleright, inst.scaletop,
          inst.scalebottom; inst.scaleleft :=
                  plot_object.xleft;
          inst.scaleright := plot_object.xright; inst.
                  scalebottom :=
                  plot_object.ybottom; inst.scaletop :=
                  plot_object.ytop;

% Pointer-function to plot_object's plots, scaled
% and positioned.
  vardef inst.plot(suffix name) = plot_object.name
          xscaled
    inst.pagewidth yscaled inst.pageheight shifted (
          inst.pageleft,
    inst.pagebottom) enddef; enddef;
```

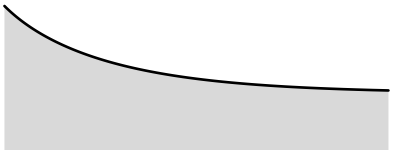

Figure 2: The capillary surface, in its unadorned form as plot object elements scaled to 2.0in by 0.75in.

Note that, immediately after a plot instance is created, the page information is unknown while the scale information is known.

We can now start putting plot objects on the page in a limited fashion, by assigning known values to the unknown page information, and then drawing the scaled picture elements.

```
input metaplot     % MetaPlot macros
input capillary    % capillary plot object

plot_instantiate(plotA, capillary); plotA.pageleft =
        0.0;
plotA.pagebottom = 0.0; plotA.pagewidth = 2.0in;
        plotA.pageheight =
0.75in; beginfig(2) draw plotA.plot(fplot)
        withcolor
0.85white; draw plotA.plot(lplot) withpen pencircle
        scaled 1pt;
endfig; end
```

The result of this is shown in Figure 2. Note that the color of the filled plot and the line size for the line plot are specified in the draw command, rather than in the plot object.

3.3 Manipulation of plot-objects

The bare plot instances are of little use without a set of macros for manipulating them. We start with a macro to set the x-axis and y-axis scales to equal values:

```
def plot_setequalaxes(suffix inst) = inst.pagewidth =
        inst.pageheight
```

```
∗ ((inst.scaleright − inst.scaleleft) / (inst.scaletop
      −
inst.scalebottom)); enddef;
```

This is written so that the page-related variables do not appear in the denominator of fractions, because either one (or both) of them may be unknown when the macro is called, and METAPOST can only solve linear equations.

There is also a set of macros for converting between locations expressed in the plot's coordinates and locations on the page. For example,

```
def plot_xpageloc(suffix inst)(expr scalex) = inst.
      pageleft + (scalex
− inst.scaleleft) ∗ (inst.pagewidth / (inst.scaleright
      −
inst.scaleleft)); enddef;
```

The additional macros in this series are *ypageloc*, *zpageloc* (which takes an x and a y coordinate as input, and returns a point), and *xscaleloc* and *yscaleloc* for the reverse direction of converting from a page location to a plot coordinate.

With these, we have most of what we need to manipulate plots in an intuitive way. For instance, consider the figure from Section 2, which can now (with some small changes) be written in a much more general way as

```
input metaplot     % MetaPlot macros
input capillary    % capillary plot object

plot_instantiate(plotB, capillary);
plot_setequalaxes(plotB); plotB.pageleft = 0.0; plotB.
      pagebottom =
0.0; plotB.pageheight = 0.75in; beginfig(3) draw
plotB.plot(fplot) withcolor 0.85white; linecap :=
      butt; pickup
pencircle scaled 1pt;
  % z-axis (vertical)
  z1 = (plotB.pageleft, plotB.pagebottom); z2 = (
        plotB.pageleft,
  plotB.pagetop + 0.1in);
  % y-axis (horizontal)
  z3 = (plotB.pageleft, plot_ypageloc(plotB,0.0)); z4
        =
  (plotB.pageright + 0.1in, plot_ypageloc(plotB,0.0));
        drawarrow z1 --
  z2; label.top(btex $z$ etex, z2); drawarrow z3
        -- z4;
  label.rt(btex $y$ etex, z4); pickup pencircle
        scaled
  0.25pt;
  % Label for contact angle
```

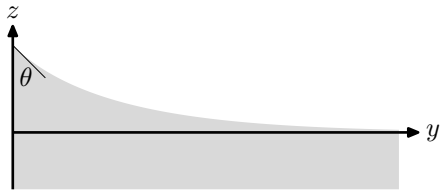

Figure 3: The capillary surface, with equal y and z scales, a page height of 0.75in, and appropriate annotations.

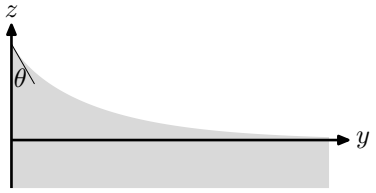

Figure 4: The capillary surface with parameters and page height as in Figure 3, but with $\theta = \pi/6$.

```
  z5 = plotB.plot(contactpoint); z6 = z5 + 0.24in ∗
        dir(−90 +
  capillary.contactangle); z7 = z5 + 0.18in ∗ dir(−90
        +
  0.5∗capillary.contactangle); draw z5 -- z6; label(
        btex
  $\theta$ etex, z7); endfig; end
```

The result of this is shown in Figure 3. We can demonstrate that this is flexible by changing the value of θ to $\pi/6$ rather than $\pi/4$, and recreating the figure using exactly the same files; the result is shown in Figure 4. Note that changing the contact angle raises the contact point, making the plot taller in scale coordinates; thus, it is drawn at a smaller scale to maintain the 0.75-inch page height.

Having two figures in this way is not the clearest way to compare the two plots, particularly with the differences in scale. A better approach is to overlay them at the same scale, making use of the existence of the filled plot from one plot object and the line plot from the other to provide a visually clear result. A simple way of placing both plots on the same coordinate axes is to require that their (0,0) and (1,1) points coincide on the page, which we do by means of the *plot_zpageloc* command; the remainder of the file is as much in the previous plots, although there is a little additional code to make certain that the axis-arrows cover both plots.

```
input metaplot     % MetaPlot macros
input capillary    % capillary plot object
input capillary2   % capillaryb plot object
```

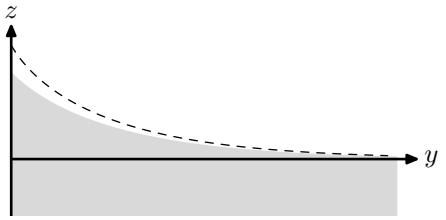

Figure 5: Two capillary surfaces, as in Figure 3 and Figure 4, showing the difference in the curves as a result of varying θ.

```
plot_instantiate(plotB, capillary);
plot_setequalaxes(plotB); plotB.pageleft = 0.0; plotB.
      pagebottom =
0.0; plotB.pageheight = 0.75in;

plot_instantiate(plotC, capillaryb); plot_zpageloc(plotB
      , 0.0, 0.0) =
plot_zpageloc(plotC, 0.0, 0.0); plot_zpageloc(plotB, 1.0,
      1.0) =
plot_zpageloc(plotC, 1.0, 1.0);

beginfig(5) linecap := butt; pickup pencircle scaled
  1pt; draw plotB.plot(fplot) withcolor 0.85white;
      draw
  plotC.plot(lplot) dashed evenly withpen pencircle
      scaled 0.5pt;
  % z-axis (vertical)
  z1 = (plotB.pageleft, plotB.pagebottom); x2 = plotB
      .pageleft; y2 =
  max(plotB.pagetop, plotC.pagetop) + 0.1in;
  % y-axis (horizontal)
  z3 = (plotB.pageleft, plot_ypageloc(plotB,0.0)); x4
      =
  max(plotB.pageright,plotC.pageright) + 0.1in; y4 =
  plot_ypageloc(plotB,0.0); drawarrow z1 -- z2;
      label.top(btex
  $z$ etex, z2); drawarrow z3 -- z4; label.rt(
      btex
  $y$ etex, z4); endfig; end
```

The result of this is shown in Figure 5.

3.4 Creation of axes

Any quantitative graph is meaningless without grid-labels for the coordinate axes, and so MetaPlot includes macros to create them. Unlike METAPOST's `graph.mp` package, MetaPlot's axis-drawing functionality requires that the user specify most of the details of the formatting, with the benefit of having a much more flexible implementation.[4]

The core of the axis-drawing functionality is a set of macros for creating generic tickmarks, labeled tickmarks, rows of tickmarks, and so forth, which are included with MetaPlot in a `axes.mp` file (and thus, for consistency, are prefaced with *axes_* rather than *plot_*). These are interfaced to the plot object coordinates by the *plot_xtickscale* and *plot_ytickscale* macros.

```
def plot_xtickscale (suffix inst) (expr startpoint,
      endpoint,
  ticklength , tickspace , tickdir , tickzero , tickstep
      , ticklabelformat )
  =
  axes_tickscale (
    startpoint , % First endpoint of the tickrow
    endpoint,    % Second endpoint of the tickrow
    ticklength , % Length of tickmarks
    tickspace , % Space between tickmark and label
    tickdir ,    % Tickmark direction
    plot_xscaleloc (inst)(xpart(startpoint)),
                 % Coordinate value at first endpoint
    plot_xscaleloc (inst)(xpart(endpoint)),
                 % Coordinate value at second
                      endpoint
    tickzero ,   % Coordinate value for a known
                 %  tick location
    tickstep ,   % Coordinate space between ticks
    ticklabelformat
                 % Format for tick labels
                 %  (syntax from format.mp package)
                 %  (use "" for no tick labels)
  ) enddef;
```

The *plot_ytickscale* definition is nearly identical. Note that these macros do not actually draw the tickmarks; they return a picture object, which can then be explicitly drawn or otherwise manipulated.

A simple way of adding grid labels to the previous example would be the following:

```
beginfig(6)
  % [ ... repeat of definitions from fig(4) ... ]
  x5 = plotB.pageleft; x6 = x4; y5 = y6 = plotB.
      pagebottom; draw
  plot_xtickscale(plotB)(z5, z6,
    0.08in, 0.06in, down, 0.0, 1.0, "%3f")
    withpen pencircle scaled 0.5pt; y7 = plotB.
        pagebottom; y8 = y2; x7
  = x8 = plotB.pageleft; draw plot_ytickscale(plotB)(
      z7, z8,
```

[4] There is, of course, no need for flexible implementations and simple interfaces to be mutually exclusive, and functions for more automated axes may be included in MetaPlot as it continues to be developed.

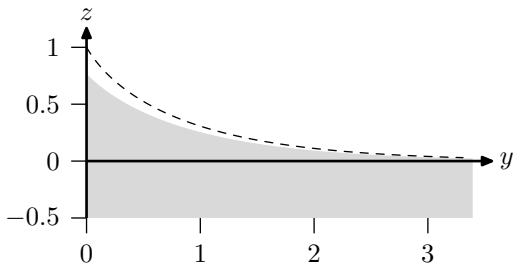

Figure 6: A repeat of Figure 5, with simple grid labels added.

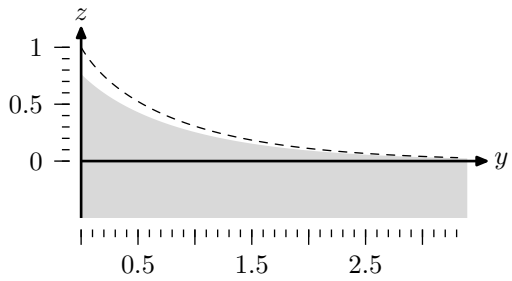

Figure 7: A repeat of Figure 5 again, with more advanced grid labels.

```
    0.08in, 0.06in, left, 0.0, 0.5, "%3f")
    withpen pencircle scaled 0.5pt; endfig;
```

The results of this are shown in Figure 6. As can be seen with the x-axis, the *tickscale* macros do not include the axis-lines themselves, thus allowing the user to draw them with a different line style than that used for the ticks, or to leave them off entirely.

For a more polished look, we can move the grid ticks a small distance away from the plot, limit the y-axis range to the region that has meaningful significance, and add intermediate ticks without labels. In addition, this example illustrates the use of the *tickzero* parameter to start the labeled x-axis ticks at 0.5 rather than zero.

```
beginfig(7)
  % [ ... repeat of definitions from fig(4) ... ]

  x5 = plotB.pageleft; x6 = x4 − 0.1in; y5 = y6 =
      plotB.pagebottom −
  0.06in; draw plot_xtickscale(plotB)(z5, z6,
    0.08in, 0.06in, down, 0.5, 1.0, "%3f")
    withpen pencircle scaled 0.5pt; draw
      plot_xtickscale(plotB)(z5,
  z6, 0.08in, 0.06in, down, 0.0, 1.0, "") withpen
      pencircle scaled
  0.5pt; draw plot_xtickscale(plotB)(z5, z6, 0.04in,
      0.06in, down,
  0.0, 0.1, "") withpen pencircle scaled 0.5pt; y7 =
      y4; y8 = y2 −
  0.1in; x7 = x8 = plotB.pageleft − 0.06in; draw
  plot_ytickscale(plotB)(z7, z8,
     0.08in, 0.06in, left, 0.0, 0.5, "%3f")
    withpen pencircle scaled 0.5pt; draw
      plot_ytickscale(plotB)(z7,
  z8, 0.04in, 0.06in, left, 0.0, 0.1, "") withpen
      pencircle scaled
  0.5pt; endfig;
```

The result is shown in Figure 7.

3.5 MetaContour: A C++ program for contour plots

Now that the METAPOST side of the collaboration has been described in some detail, we return to the matter of programs that generate plot objects as output. One of the particular reasons for developing MetaPlot was to have a way of producing contour plots, and so the MetaPlot package comes with a C++ program, MetaContour, for creating them.

The internals of MetaContour are beyond the scope of this paper, but it does make use of one additional capability of plot objects that is worth noting — the ability to include color information. The plot object is defined with commands like the following, with color directives.

```
picture contplotA.LinePlot; contplotA.LinePlot :=
      nullpicture; addto
contplotA.LinePlot doublepath (0.48075,0.50000)
      -- (0.48163,0.50597)
withcolor contourcolor27; addto contplotA.LinePlot
      doublepath
(0.48420,0.50000)-- (0.48492,0.50490) withcolor
      contourcolor28; addto
contplotA.LinePlot doublepath (0.45994,0.50000)
      -- (0.46169,0.51245)
withcolor contourcolor23;
    % [ ... and so forth ... ]
```

Then, before the plot object file is read into the main METAPOST file, the *contourcolor* array is defined as desired.

```
% Contour colors for grayscale scheme
color contourcolor[ ]; contourcolor0 = 1white;
      contourcolor1 =
0.98white;
    % [ ... and so forth ... ]
contourcolor30 = 0.4white;
```

Thus, each line of the contour plot is associated with a color, and it will be drawn in that color unless it is overridden by another color directive; for instance,

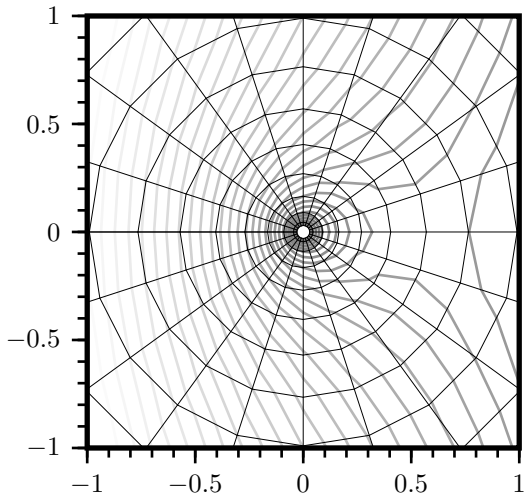

Figure 8: Sample graph created by MetaContour and MetaPlot, showing potential lines for a combination of a linear gradient and a point source, plotted on a polar grid.

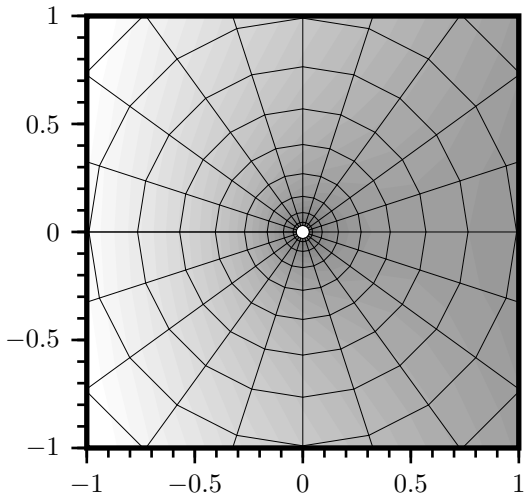

Figure 9: Another sample graph created by MetaContour and MetaPlot, illustrating a filled contour-plot style rather than using contour lines.

if we wanted to plot the contour lines all in black, we could do so simply by specifying:

draw *continstA.plot*(*LinePlot*) **withcolor** black;

Aside from the color contour-line plot just described, the MetaContour output contains a filled contour plot, and an image of the mesh of data points. Some examples of these are shown in Figure 8 and Figure 9; although these are much more complex than the examples from preceding sections, the MetaPlot commands to generate them are nearly identical.

4 Conclusion

The examples that have been shown illustrate only a small sampling of the capabilities of MetaPlot. In using METAPOST to generate the figures, it provides an easily extensible layout capability that is not limited by the imagination of the package author. The standardized plot-object interface simplifies the process of writing plot-generation programs, as they can leave the details of layout and annotation to the MetaPlot postprocessing.

MetaPlot and MetaContour are available from CTAN in the `/graphics/metaplot` directory. They are still very much works in progress; I look forward to suggestions and improvements, and hope that others will find them to be useful tools.

References

[1] Hwang, A., ePiX, `http://mathcs.holycross.edu/~ahwang/current/ePiX.html`.

[2] Gnuplot, `http://www.gnuplot.info`.

[3] Phan, A., m3D, `http://www-math.univ-poitiers.fr/~phan/m3Dplain.html`.

[4] Batchelor, G. K., *An Introduction to Fluid Dynamics*, Cambridge University Press, 1967.