Describing Mathematical Figures with METAPOST

Jeffrey Clark
Elon University

October 30, 2004

Introduction

Mathematicians use diagrams to convey information, demonstrate algorithms, and convey geometric relationships. Such diagrams need to accurately convey all kinds of information. This talk will discuss a program called METAPOST that is easy to learn and use, and which supports a variety of styles for constructing diagrams.

METAPOST is a graphics programming language invented by John D. Hobby, based on the METAFONT system for font creation invented by Donald Knuth. Using a very concise but clear syntax, it produces output in Postscript (a language for printing and displaying documents). I will try in a brief time to demonstrate a number of different styles for describing diagrams that are possible in using METAPOST.

This talk is available at *http://frodo.elon.edu* under the link *Presentations*.

I will discuss the following different styles of for creating diagrams with METAPOST:

- Coordinates
- Related Points
- Line Intersections
- Curve Intersections
- Transformations

Coordinates

Some mathematical diagrams are based on specific coordinates; for example, most of the statistical plots that I construct for my classes require the placement of points at specific coordinates within the plot.

In METAPOST, a point with coordinates x and y is represented as an ordered pair, (x, y). It is presumed that x and y contain appropriate information about the units used.

Example: Box-plot

Suppose that I have a data set with the following five-number summary:

Minimum	110
First Quartile	208
Median	373
Third Quartile	579
Maximum	1016

I would like to construct a box-plot for this summary, with proportionate lengths and a labeled axis underneath it.

The box-plot consists of two whiskers and a box, with the box split at the median. I will place the bottom of the box at height 100, the whiskers at height 150, and the top of the box at height 200.

In a METAPOST file, Comment lines begin with %.

A METAPOST file can contain several figures. Each begins with a *beginfig* command and ends with an *endfig* command. After all figures are complete, the file ends with an *end* command.

In METAPOST, ":=" is used to assign numerical parameters.

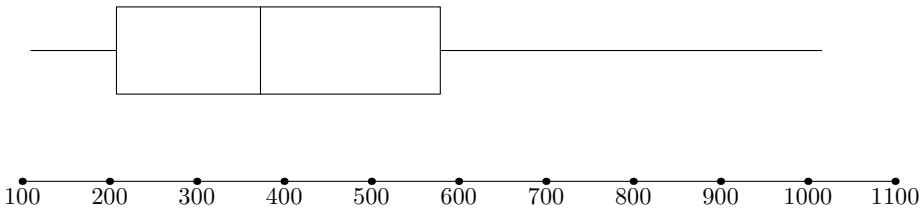
To create the graph with given dimensions, we start by defining width and height and then determine units in the horizontal and vertical dimensions. For the sake of convenience, we define a macro *pt* early on to keep track of the units.

The command *draw (a, b) -- (c, d);* draws a line segment between the points *(a, b)* and *(c, d)*. When concluding a closed polygon, the word *cycle* is used for the last point.

METAPOST contains for-loops and other control structures.

The *dotlabel* command is used to place a labeled dot at a point. The *decimal* command is used to convert a number to a string label.

```
% boxplot
beginfig(0);
  width := 5in; height := 1in;
  xmin := 100; xmax := 1100; deltax := 100;
  ymin := 0; ymax := 200;
  xunit := width/(xmax - xmin);
  yunit := height/(ymax - ymin);
  def pt(expr x, y) = (x*xunit, y*yunit)
  enddef;
% axis
  draw pt(xmin, 0) -- pt(xmax, 0);
  for x = xmin step deltax until xmax:
    dotlabel.bot(decimal(x), pt(x, 0));
  endfor;
% boxplot
  draw pt(110, 150) -- pt(208, 150);
  draw pt(208, 100) -- pt(208, 200)
    -- pt(579, 200) -- pt(579, 100) -- cycle;
  draw pt(373, 100) -- pt(373, 200);
  draw pt(579, 150) -- pt(1016, 150);
endfig;
end;
```



Related Points

In many diagrams, once one point is specified, the rest of the diagram's control points are determined by their relations to each other.

METAPOST will take a system of linear equations relating the coordinates of the control points, as long as it has a unique solution, and solve for the coordinates. This provides remarkable freedom in describing a diagram.

Example: Regular Heptagon

We will construct a regular heptagon, with radius (distance from the center to one of the vertices) equal to 1 inch.

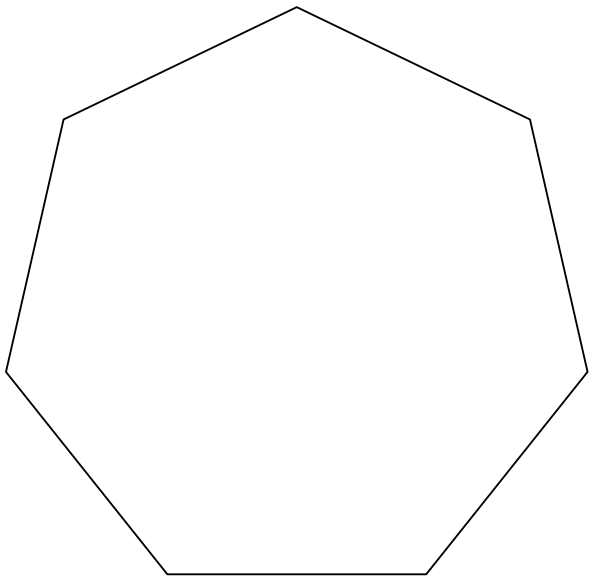
We will set the center at the origin, and define all of the other vertices relative to the center.

Points in METAPOST are commonly identified as *z1*, *z2*, ...

The word *origin* is a synonym for the coordinates *(0, 0)*.

The command *dir(theta)* produces a unit vector pointing in the direction *theta*, where *theta* is measured in degrees.

```
% heptagon.mp
beginfig(0);
  theta := 360/7;
  radius := 1in;
  z0 = origin;
  z1 = z0 + dir(90)*radius;
  z2 = z0 + dir(90 + theta)*radius;
  z3 = z0 + dir(90 + 2*theta)*radius;
  z4 = z0 + dir(90 + 3*theta)*radius;
  z5 = z0 + dir(90 + 4*theta)*radius;
  z6 = z0 + dir(90 + 5*theta)*radius;
  z7 = z0 + dir(90 + 6*theta)*radius;
  draw z1 -- z2 -- z3 -- z4 -- z5 -- z6 -- z7
    -- cycle;
endfig;
end;
```



Example: Nested Rectangles

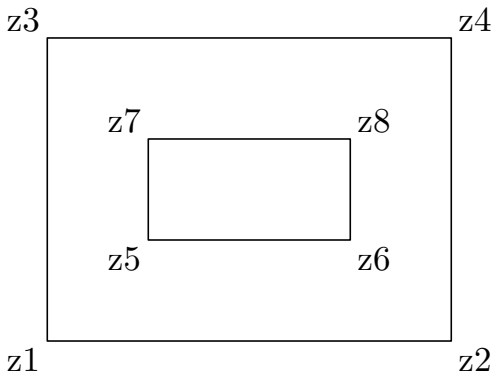
I would like to construct two nested rectangles, one that is 3 inches by 4 inches, the other that is 1 inch by 2 inches, with the smaller rectangle nested symmetrically within the larger.

Once we specify the location of one of the control points in this diagram, it is not hard to solve for the locations of the other points, but we will let METAPOST handle the computations for us.

When working with related points, it is easier to work with points labeled as *z1*, *z2*, *z3*, etc. Their x-coordinates are *x1*, *x2*, *x3*, etc., and their y-coordinates are *y1*, *y2*, *y3*, etc.

label is used to place labels in diagrams.

```
% nested.mp:  nested rectangles
beginfig(0);
  scale = 0.4;
  z1 = origin;
  x2 - x1 = 4in*scale; y2 = y1;
  x3 = x1; y3 - y1 = 3in*scale;
  x4 = x2; y4 = y3;
  draw z1 -- z2 -- z4 -- z3 -- cycle;
  label.llft("z1", z1);
  label.lrt("z2", z2);
  label.ulft("z3", z3);
  label.urt("z4", z4);
  x6 - x5 = 2in*scale; x5 - x1 = x2 - x6;
  y7 - y5 = 1in*scale; y5 - y1 = y3 - y7;
  x7 = x5; x8 = x6;
  y6 = y5; y8 = y7;
  draw z5 -- z6 -- z8 -- z7 -- cycle;
  label.llft("z5", z5);
  label.lrt("z6", z6);
  label.ulft("z7", z7);
  label.urt("z8", z8);
endfig;
end;
```



Line Intersections

Often control points are determined as the intersections of lines or curves.

In parametric equations, lines are given as $P + t\vec{A}$, where P is a point on the line and $\vec{A}$ is a non-zero vector indicating the direction of the line.

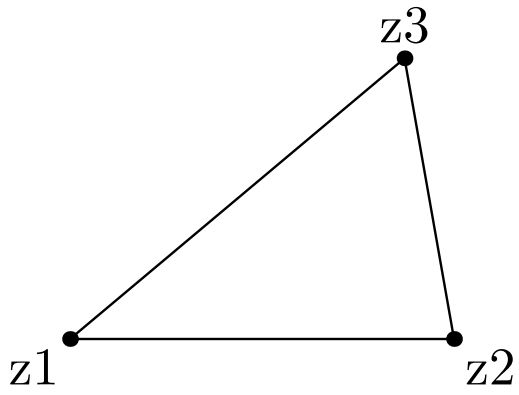
In METAPOST, an arbitrary point on such a line is written as P + *whatever*A*. Setting a point equal to two such expressions for intersecting lines is enough to determine the point.

Example: Triangle with Given Base and Base Angles

We will construct a triangle whose base is of length 2 inches, and with base angles 40° and 80°. The left base angle will point in the direction *dir(40)*, and the right base angle will point in the direction *dir(100)* of the supplement of 80°.

```
% triangle.mp
beginfig(0);
  scale := 0.5;
  z1 = origin;
  z2 = (2in*scale, 0);
  z3 = z1 + whatever*dir(40);
  z3 = z2 + whatever*dir(180 - 80);
  draw z1 -- z2 -- z3 -- cycle;
  dotlabel.llft("z1", z1);
  dotlabel.lrt("z2", z2);
  dotlabel.top("z3", z3);
```

```
endfig;
end;
```



Example: Circumcenter

We will now take the same triangle and find its circumcenter, i.e., the center of the circle that goes through its vertices.

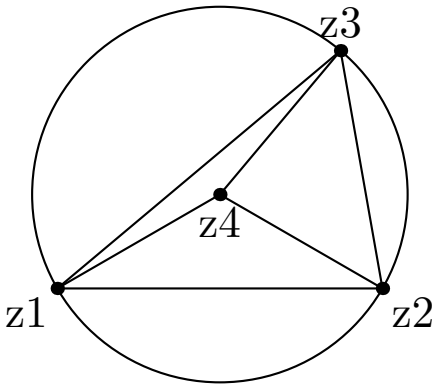
The center lies on the intersection of the perpendicular bisectors of the three sides; we need only intersect two of the bisectors.

The command *v rotated theta* takes the vector v and rotates it by the angle *theta*. If we have two distinct points A and B determining a line, their midpoint is *(A + B)/2*, and their perpendicular bisector is *(A + B)/2 + whatever*(B - A) rotated 90*.

fullcircle is a circle of diameter 1 centered at the origin; *fullcircle scaled d* is a circle of diameter d centered at the origin; *fullcircle scaled d shifted c* is a circle of diameter d centered at c.

```
% circumcenter.mp
beginfig(0);
  scale := 0.5;
  z1 = origin;  z2 = (2in*scale, 0);
  z3 = z1 + whatever*dir(40);
  z3 = z2 + whatever*dir(180 - 80);
  dotlabel.llft("z1", z1);
  dotlabel.lrt("z2", z2);
  dotlabel.top("z3", z3);
  z4 = (z1 + z3)/2 + whatever*(z3 - z1) rotated 90;
  z4 = (z2 + z3)/2 + whatever*(z3 - z2) rotated 90;
  dotlabel.bot("z4", z4);
```

```
  draw z1 -- z2 -- z3 -- cycle;
  draw z4 -- z3;  draw z4 -- z1;  draw z4 -- z2;
  draw fullcircle scaled (2*abs(z4 - z3)) shifted z4;
endfig;
end;
```



Example: Incenter

We will now take the same triangle and find its incenter, i.e., the center of the circle inscribed in the triangle.

The center lies on the intersection of the angle bisectors of the three sides; we need only intersect two of the bisectors.

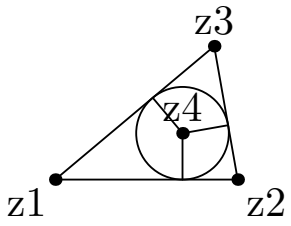
The points of contact between the circle and the triangle are the intersections of the perpendiculars drawn from the incenter to the sides.

angle(v) yields the direction angle for a vector v. To find the angle bisector for vectors v and w, we use the direction vector *dir((angle(v) + angle(w))/2)*.

The command *r[A, B]* finds the point with barycentric coordinate r on the line determined by A and B. *0[A, B]* is A and *1[A, B]* is B. Setting a point equal to *whatever[A, B]* means that it is on the line determined by A and B.

```
% incenter.mp
beginfig(0);
  scale := 0.3;
  z1 = origin;  z2 = (2in*scale, 0);
  z3 = z1 + whatever*dir(40);
  z3 = z2 + whatever*dir(180 - 80);
  dotlabel.llft("z1", z1);
  dotlabel.lrt("z2", z2);
  dotlabel.top("z3", z3);
```

```
  draw z1 -- z2 -- z3 -- cycle;
  z4 = z1 + whatever*dir((angle(z3 - z1)
      + angle(z2 - z1))/2);
  z4 = z2 + whatever*dir((angle(z1 - z2)
      + angle(z3 - z2))/2);
  dotlabel.top("z4", z4);
  z12 = whatever[z1, z2];
  z12 = z4 + whatever*(z2 - z1) rotated 90;
  z13 = whatever[z3, z1];
  z13 = z4 + whatever*(z1 - z3) rotated 90;
  z23 = whatever[z3, z2];
  z23 = z4 + whatever*(z2 - z3) rotated 90;
  draw z4 -- z12; draw z4 -- z13; draw z4 -- z23;
  draw fullcircle scaled (2*abs(z4 - z12))
      shifted z4;
endfig;
end;
```



Curve Intersections

METAPOST can draw Bezier curves through points to produce many curves in addition to circles. Once the curves have been specified, METAPOST can determine any points of intersection.

Example: Fixed Point

We will find the fixed point of the cosine function in the first quadrant by graphing $y = \cos(x)$ and $y = x$, and letting METAPOST solve for their point of intersection and its coordinates.

METAPOST treats all angles in degrees, and appends a "d" to the names of its trigonometric functions accordingly. We can define cosine as *cosd(x*pi/180)*.

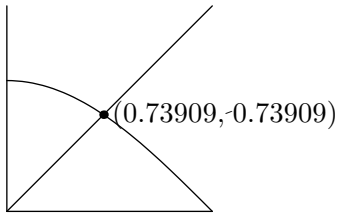
Now that we will be trying to find the intersection of two curves, we need to give them names after declaring them to be *paths*.

A Bezier curve through points P, Q, R is written P .. Q .. R. If there are many points defining the curve, it is simpler to use a for-loop.

Once the curves *C1* and *C2* have been specified, we can find their point of intersection as *C1 intersectionpoint C2*.

In constructing the label for the point of intersection, we use *&* to concatenate the parts of the label.

```
% fixedpoint.mp
beginfig(0);
  pi := 3.14159; width := 1in;
  minimum := 0; maximum := pi/2;
  delta := (maximum - minimum)/10;
  unit := width/(maximum - minimum);
  def pt(expr x, y) = (x*unit, y*unit) enddef;
  def f(expr x) = cosd(x*180/pi) enddef;
  draw pt(minimum, minimum) -- pt(maximum, minimum);
  draw pt(minimum, minimum) -- pt(minimum, maximum);
  path curve, line;
  curve =  pt(minimum, f(minimum))
  for x = minimum step delta until maximum:
    .. pt(x, f(x)) endfor;
  draw curve;
  line =  pt(minimum, minimum)
    -- pt(maximum, maximum);
  draw line;
  z1 = curve intersectionpoint line;
  string lab;
  lab = "(" & decimal(x1/unit) & ", "
    & decimal(y1/unit) & ")";
  dotlabel.rt(lab, z1);
endfig;
end;
```



Transformations

METAPOST has affine transformations as one of its data types. An affine transformation consists of a linear transformation followed by a translation; in two dimensions this depends on six parameters.

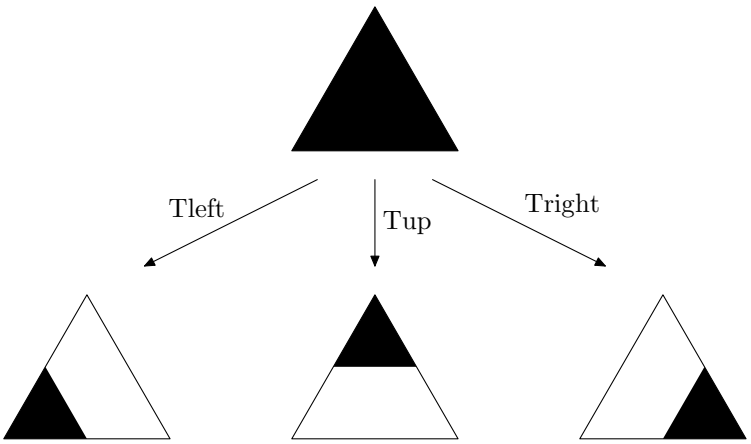
Every time you specify the image of a point under a transformation, you provide two linear equations to METAPOST. Therefore a transformation is specified if you give the images of three points.

rotated and *shifted* are special cases of affine transformations.

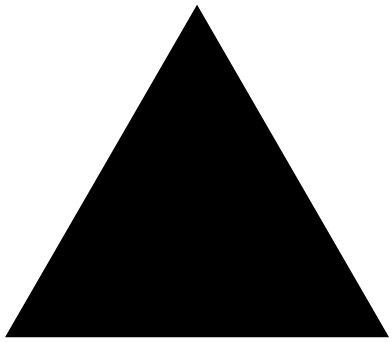
Example: Sierpinski's Triangle

Sierpinski's triangle is formed by starting with a solid triangle, and then removing the middle fourth, and continuing to do so ad infinitum to all remaining triangles.

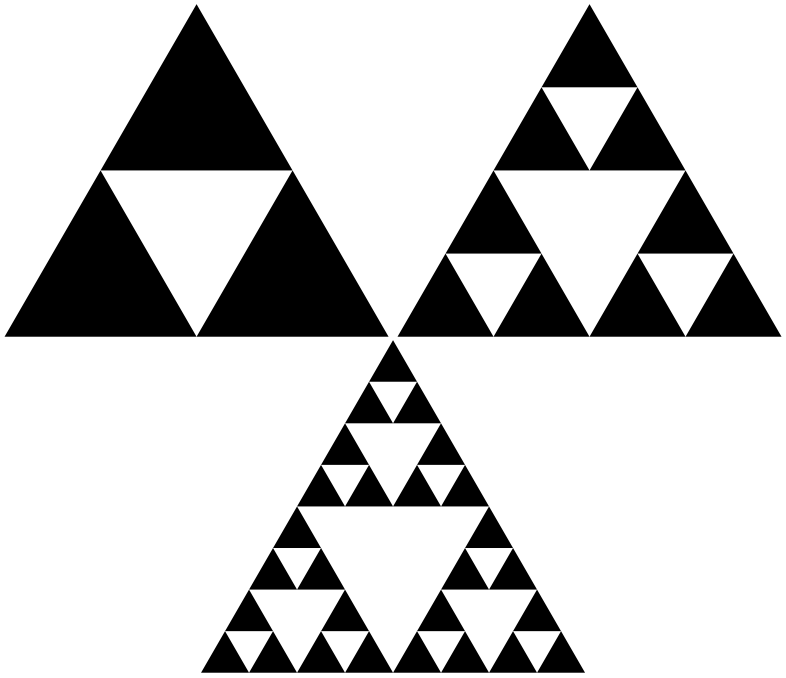
We can form finite approximations to Sierpinski's triangle by defining three transformations: for each vertex we contract the triangle halfway toward that vertex.



We will define the finite approximations to Sierpinski's triangle recursively. The first approximation is a solid triangle.

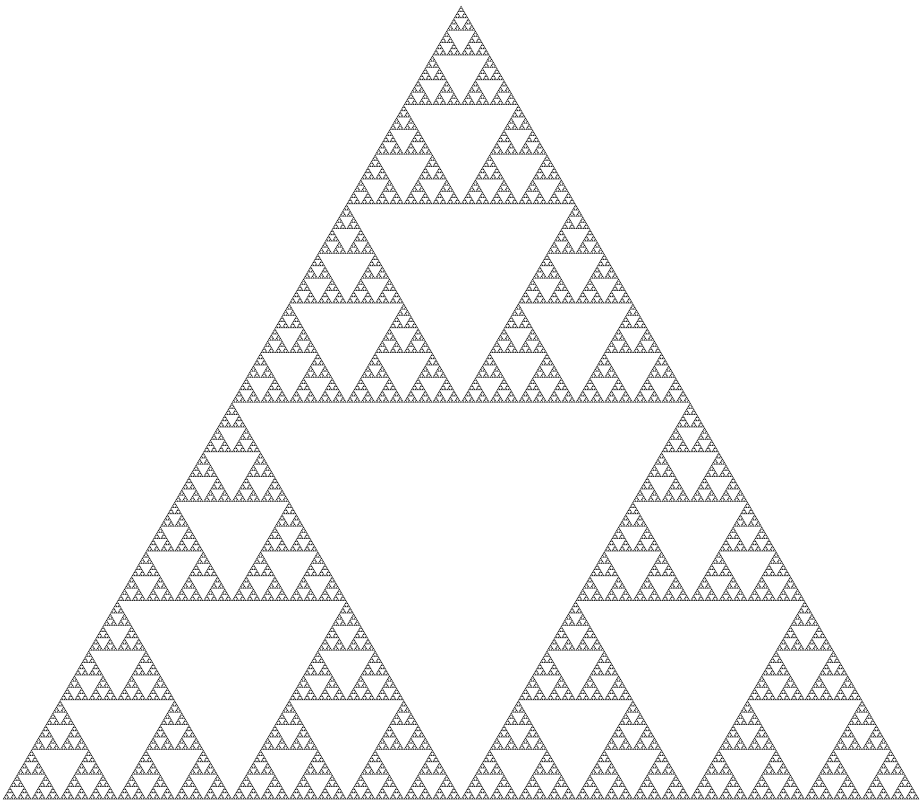


Successive approximations are found by applying each of the three transformations separately, and taking the union of the result.



```
% sierpinksi.mp
beginfig(0);
  width := 2in; transform Tleft, Tup, Tright;
  z1 = origin; z2 = z1 + width*dir(60);
  z3 = z1 + width*right;
  z1 transformed Tleft = z1;
  z2 transformed Tleft = 1/2[z1, z2];
  z3 transformed Tleft = 1/2[z1, z3];
  z1 transformed Tup = 1/2[z1, z2];
```

```
  z2 transformed Tup = z2;
  z3 transformed Tup = 1/2[z2, z3];
  z1 transformed Tright = 1/2[z1, z3];
  z2 transformed Tright = 1/2[z2, z3];
  z3 transformed Tright = z3;
  def sierpinski(expr a, b, c, n) =
    if n = 1: fill a--b--c--cycle;
    else: sierpinski(a transformed Tleft,
        b transformed Tleft,
        c transformed Tleft, n - 1);
      sierpinski(a transformed Tup,
        b transformed Tup,
        c transformed Tup, n - 1);
      sierpinski(a transformed Tright,
        b transformed Tright,
        c transformed Tright, n - 1);
    fi;
  enddef;
  sierpinski(z1, z2, z3, 10);
endfig;
end;
```



There are many approaches to constructing mathematical diagrams; which you use depends on the nature of what you are trying to convey.

Data-oriented diagrams will rely heavily upon coordinates.

Many geometric figures are determined by control points which are most easily described in relation to each other.

Others involve determining points found on the intersections of linesor curves.

Fractal pictures are transformation oriented; the key to their construction lies in identifying the correct transformation(s).

METAPOST supports all of these approaches and more.

References

- *http://cm.bell-labs.com/who/hobby/MetaPost.html* is John Hobby's home page for METAPOST and a good source for manuals, tutorials, etc.
- *http://tex.loria.fr/prod-graph/zoonekynd/metapost/metapost.html* contains an amazing repertory of METAPOST examples of code and diagrams.