1 Start here

This document contains a collection of ideas and techniques for producing attractive technical drawings with John Hobby's METAPOST language. I'm assuming that you already know the basics of the language, that you have it installed as part of your up to date TEX ecosystem, and that you have established a reasonable workflow that let's you write a METAPOST program, compile it, and include the results in your TEX document. If not, you might like to start at the METAPOST page on CTAN, and read some of the excellent tutorials, including `mpintro.pdf`. If you have already done this, please read on.

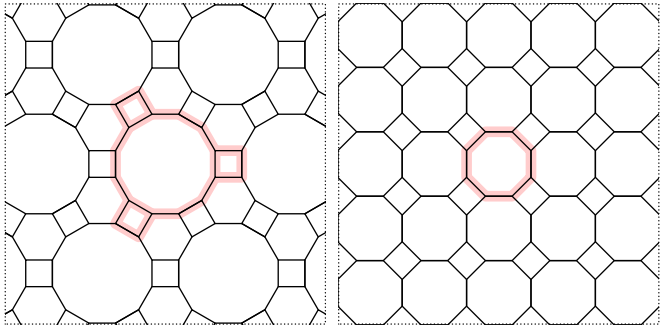
These notes are partly based on the examples I have developed as answers to questions about technical drawing on the TEX Stack Exchange site. In accordance with their terms and conditions, I've only included material here that I've written myself — if you want other people's code then visit the site; while most answers there focus on writing LATEX documents, there are a great many questions about drawing, and some of the answers are very illuminating.

My approach here will be to explore plain METAPOST, with examples grouped into themes. One approach to using this document would be to read it end to end. Another would be to flick through until you see something that looks like it might be useful and then see how it's done.

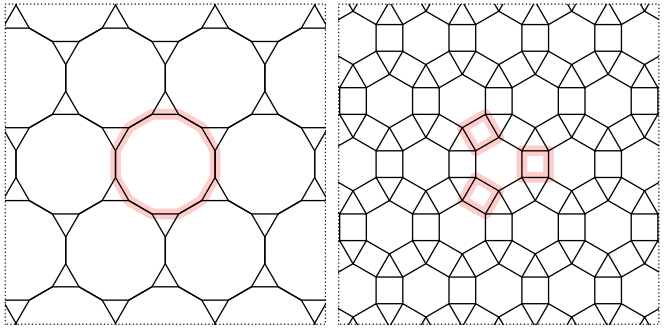
And when I say *plain* METAPOST I mean METAPOST with the default format (as defined in the file `plain.mp`) loaded and only a few simple external packages (like `colorbrewer`) occasionally. Nearly all of the examples here are supposed to be self contained, and any macros are defined locally so you can get to grips with what's going on. METAPOST is a very subtle language, and it's possible to do some very clever and completely inscrutable things with it; but here I have tried to be as clear as possible in my examples.

Drawing with Metapost

Toby Thurston — March 2017 – October 2024



$(4,6,12)$ $(4,8^2)$



$(3,12^2)$ $(3,4,6,4)$

2 Some features of the syntax

- Assignment or equation: the equation `a=3;` means "`a` is the same as `3` throughout the current scope"; the assignment `a:=3;` means "update the value of `a` to the value `3` immediately". The difference becomes apparent when you try to update a variable in the same scope.

  This difference also lets you write linear equations like `a=-b;`. After this, as soon as you give a value to `a`, METAPOST immediately works out the value of `b`. This is clever but has its limitations. As the following snippet reveals:

```
% if you run this       you will get this in the log
a + b = 0; show (a,b);  % >> (a,-a)
a=42;      show (a,b);  % >> (42,-42)
a:=43;     show (a,b);  % >> (43,-42)
```

  As soon as you assign to variable with `:=` METAPOST breaks any previously established equations.

- Variable types:

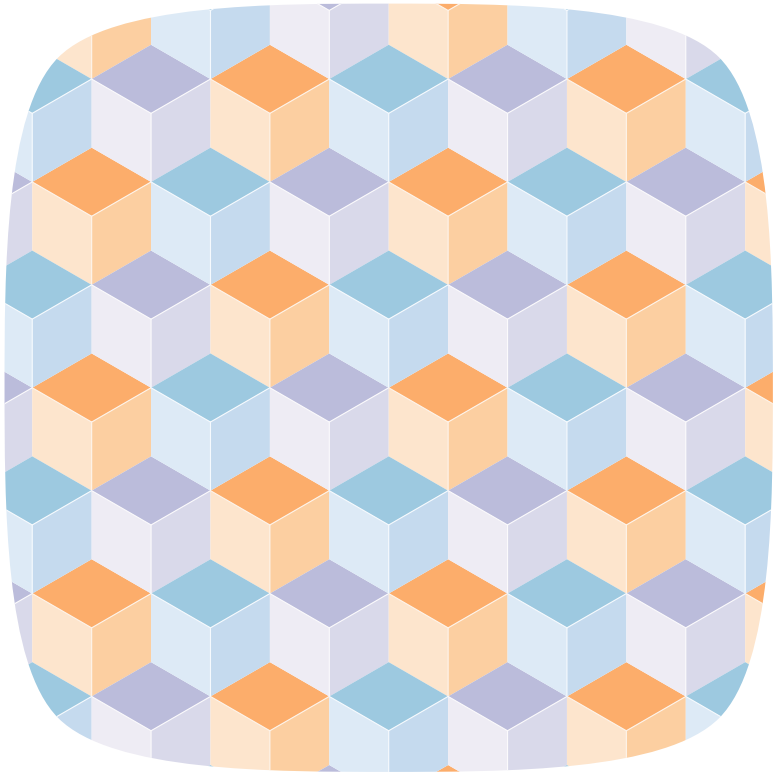
  — ⟨numeric⟩ `a`, ⟨pair⟩ `(a,b)`

  — ⟨color⟩ `(r,g,b)`, ⟨cmykcolor⟩ `(c,m,y,k)` ⟨transform⟩ `(x,y,xx,xy,yx,yy)`

  — ⟨string⟩, ⟨path⟩, ⟨picture⟩

  If you don't declare a variable, it's assumed that it's a ⟨numeric⟩. When you do declare a variable — ⟨numeric⟩ or otherwise — any value that it already had in the current scope is removed.

- Implicit multiplication: METAPOST inherits a rich set of rules about numerical expressions from METAFONT, and of special interest is the scalar multiplication operator. Any simple number, like `42`, `3.1415`, or `.6931`, or any simple fraction like `1/2` or `355/113` standing on it's own (technically at the primary level) and not followed by `+` or `-` becomes a scalar multiplication operator that applies to the next token (which should be variable of some appropriate type). So you can write things like `3a`, or even `1/2 a` (the space between the number and the variable name is optional). This lets you write very readable mathematical expressions. It's quite addictive after a while.



The `sqrt` operator is defined at the same (top) level of precedence, so that `sqrt2+1` is read as `(sqrt2)+1` and not `sqrt(2+1)`, but fractions trump even that, so `sqrt 1/2 = 0.7071` is true.

3 Workflow

This document is not meant for beginners, so you won't find step by step tutorials for something so simple as running METAPOST. But since you might not find it all that simple, and since the basic tutorials can go out of date, here are descriptions of my own workflows that you might find helpful. You might also think I'm being really inefficient; if so please drop me a line and suggest an improvement.

The common features of each of these workflows are: Mac OS, the MacVim editor to edit METAPOST source code, and Skim.app to view PostScript and PDF files. I have the complete MacTeX distribution installed; any commands mentioned below are supplied by MacTeX.

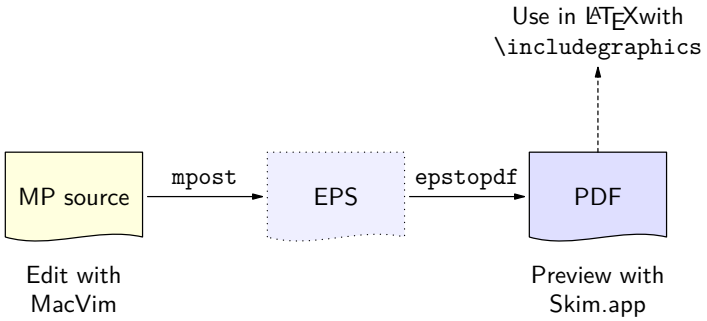
3.1 Stand alone graphics with plain METAPOST

METAPOST source files have the extension `.mp`, when I open a file in MacVim that matches `*.mp`, my editor profile sets the file type to `mp` (which picks up the highlight and indentation rules supplied with MacVim), and adds some relevant directories to the search tree. Finally, if the file is a new file, then the profile loads this template:

```
prologues := 3;
outputtemplate := "%j%c.%{outputformat}";
beginfig(1);

endfig;
end.
```

The first two lines are important: `prologues := 3;` makes METAPOST put the full font details in the output so that the files are self-contained; the `outputtemplate` line means that the output will be written to files with an extension that matches the chosen output format, which will be `png`, `svg`, or more usually `eps`, which is the default (and suggests that the output is Encapsulated PostScript).

I then add drawing and label commands, using all the traditional facilities for typesetting labels described in section 11. I compile the source with `mpost`. I usually do this from within MacVim using the command line `:!mpost %` where `!` means "this is an external command" and the `%` picks up the current file name. Usually I need several attempts to get a diagram right, so I open Skim to preview the output.



Until recent (2024) versions of Mac OS, it was possible to get Skim to view the PostScript output directly, with automatic updates on recompile, but the conversion from PostScript no longer works properly, so I now prefer to convert the EPS to PDF using `epstopdf`. This slightly complicates the edit, compile, and preview loop. On the other hand the PDFs are generally more useful files to create, so it is worth the extra effort.

I use a small Python script to automate the process: run `mpost` with the `-recorder` option; scan the list of files to see what got produced; check which ones are PostScript; call `epstopdf` to make each one into a PDF file; remove each EPS file if successful. Your mileage may vary.

3.2 Stand alone graphics with LuaLaTeX

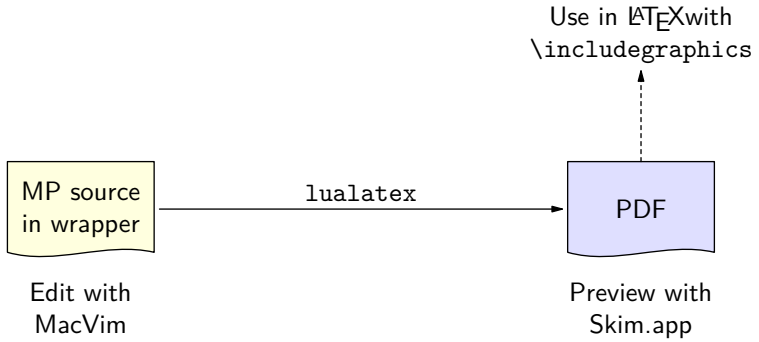
For graphics with more complicated text formatting, I prefer now to use `lualatex` with the `luamplib` package. The work flow is a bit simpler because there are no intermediate EPS files to worry about. Instead of compiling with plain `mpost` I use `lualatex` with the `luamplib` package, which calls METAPOST from within the Lua environment. The METAPOST engine actually used is exactly the same. Here is the template I use:

```
\documentclass[border=5mm]{standalone}
\usepackage{luamplib}
\mplibtextextlabel{enable}
\begin{document}
\begin{mplibcode}
 beginfig(1);

 endfig;
\end{mplibcode}
\end{document}
```

As you can see, we have METAPOST source code wrapped up in a minimal LaTeX document using the `standalone` class, which automatically adjusts the page size to fit the contents of document, so is ideal for single diagrams. One small disadvantage is that you can only produce a single PDF output file, so you need to have a separate file for each picture, but the good news is that you get a much simpler and more effective integration with LaTeX, in particular with the font environment, but as this only works with `lualatex` you have to use the `fontspec` package, as explained in section 12.

3.3 Integrated graphics with LuaLaTeX

If you are ready to use `lualatex` for processing your entire document, then you can directly embed your METAPOST drawings in a series of `mplibcode` environments. Each one produces a horizontal-mode box. For details try `texdoc luamplib`. The only drawback of this all-in-one approach is that you have to compile all the drawings every time you compile the document, which might slow you down — although on a modern machine this is not really an issue any more.



☞ If you like the fancier METAPOST format provided with ConTexT, you can use it directly with this `luamplib` approach. Just add this option to your preamble:

```
\mplibsetformat{metafun}
```

4 Making and using paths

In METAPOST there are two sorts of paths: open and closed. A closed path is called a cycle, and is created with the `cycle` primitive like this:

```
path t; t = origin -- (55,0) -- (55,34) -- cycle;
```

You can think of `cycle` as meaning 'connect back to the start and close the path'. Note that you have explicitly put `cycle` to make a closed path. If you wrote

```
path u; u = origin {right} .. (55,0) .. (55,34) .. {-2,-1} origin;
```

then u would be an open path even though the last point is the same as the first. Any path that does not have `cycle` at the end is an open path.

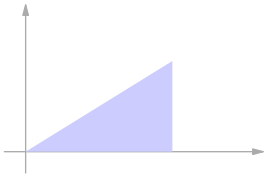
You can use `draw` with either sort of path, but you can only use `fill` with a cycle. This concept is common to most drawing languages but it's often hidden: an open path might be automatically closed for you when you try to fill it. METAPOST takes a more cautious approach; if you pass an open path to `fill` you will get an error that says 'Not a cycle', even if the first and last points are the same like path u above.

If you want to write a macro that deals differently with the two types of path, then you can use `cycle` in a boolean context to test whether a given path p is closed:

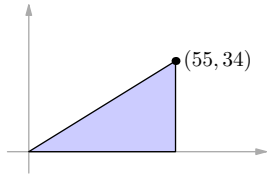
```
if cycle p:
    % do something for closed path p
else:
    % do something for open path p
fi
```

METAPOST inherits the rich path-making syntax directly from METAFONT, so if you want a general refresher, or you are not quite sure what the five joiners do, ⟶ or you would like to bone up on exactly what `curl` and `tension` are for, then you are recommended to review Chapter 14 of *The METAFONTbook*.

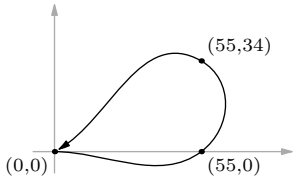
Most of the examples in this document use only the two simple joiners `--` and `..` with the occasional use of a direction-specifying pair before or after a point.



```
fill t withcolor 0.8[blue,white];
```




```
draw t;
```




```
drawarrow u cutafter fullcircle scaled 4;
```

`..`	free curve
`...`	bounded curve
`--`	straight line
`---`	tense line
`&`	splice.

4.1 Predefined closed paths

There are several closed paths defined for you in plain METAPOST.

- `unitsquare` is defined as the path `(0,0)--(1,0)--(1,1)--(0,1)--cycle`. It runs counter-clockwise from the origin, and you can use it to draw any rectangle with appropriate use of `xscaled` and `yscaled`, or a parallelogram with `slanted`, or a diamond with `rotated` — but note that the definition means that is it centred on point `(1/2, 1/2)` so you might want to shift it by `-(1/2,1/2)` before you transform it.

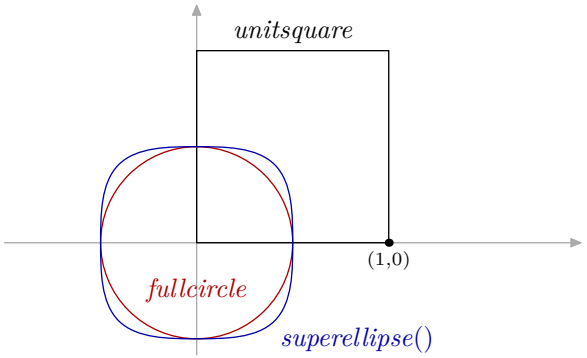
- `fullcircle` which you can use to draw any circle or ellipse with appropriate use of `xscaled` and `yscaled`. Unlike the square, it is defined so that it is centred at the origin. But beware that it has unit *diameter*, so its radius is 0.5 bp long. The path runs counter-clockwise and starts at 3 o'clock; which means `point 0 of fullcircle = 1/2 right` is true.

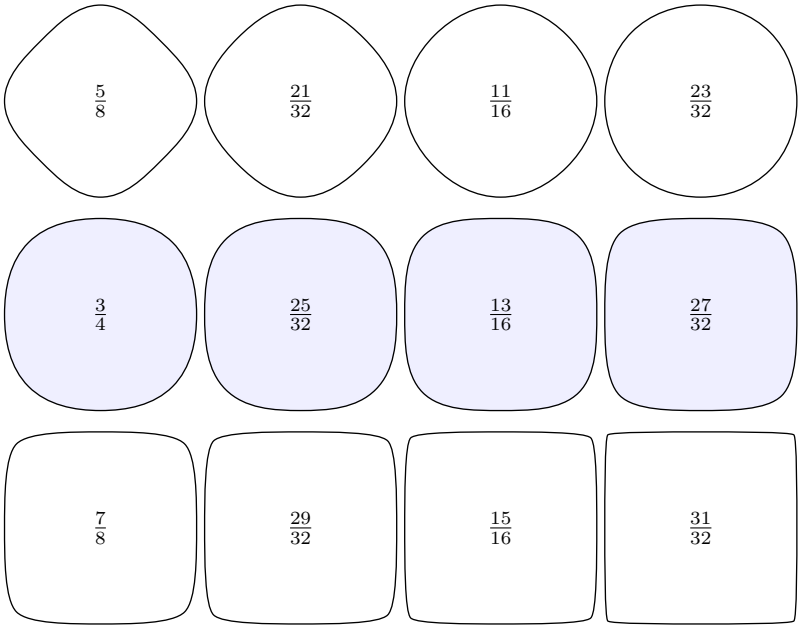
- `superellipse()` which creates the shape beloved of the Danish designer Piet Hein. Unlike the other two, this one is defined as a function rather than a ⟨path⟩ constant, so you need to call it like this:

  ```
  path s;
  s = superellipse(1/2 right, 1/2 up, 1/2 left, 1/2 down, 13/16);
  ```

  to create a 'unit' shape that matches `fullcircle` as shown above. The fifth parameter is the 'superness': the value 1 makes it look almost square; $\frac{1}{2}$ gives you a diamond; a value somewhere between $\frac{3}{4}$ and $\frac{7}{8}$ looks about right. ⟶ Values outside the range $(0.5, 1)$ give you rather weird propeller shapes.

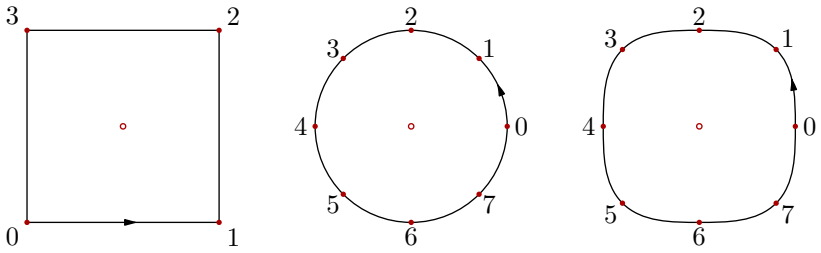
  ❖ Note that, unaccountably, *superellipse*() is defined in `plain.mp` with a **def** rather than a **vardef**. This means you need to enclose it in a group before you can transform it in any way. One way to do this is to use parentheses; or you can assign it to a ⟨path⟩ variable, as shown above.





4.2 Points on the standard closed paths

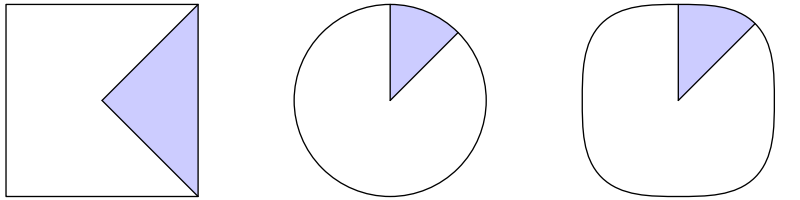
HERE ARE THE THREE SHAPES centred on the origin and labelled to show the points along them. **Note** that the *unitsquare* shape has been shifted so that it is centred on the origin in all of these examples. The small red circle marks the *origin*, and the labelled red dots are the points of each path. The *unitsquare* has four points, while the other two shapes both have eight. The small arrows between point 0 and point 1 of each shape indicate the direction of the path that makes up the shape.



If you want to highlight a segment of your shape, there's a neat way to define it using `subpath`. Assuming `p` is the path of your shape, then this:

```
center p -- subpath(1,2) of p -- cycle
```

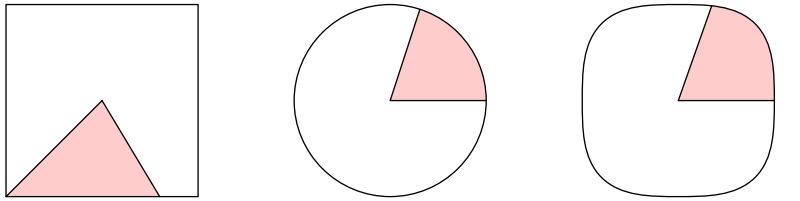
creates a useful wedge shape which looks like this in our three 'standard' shapes.



Better still, you are not limited to integer points along the path of your closed shape. So if you wanted a wedge that was exactly 1/5 of the area of your shape, you could try

```
center p -- subpath(0,1/5 length p) of p -- cycle
```

Clearly this works rather better with more circular shapes. Indeed for a circle you can convert directly between circumference angle and points along the path. So you have defined path c to be scaled copy of `fullcircle`, then `point 1 of c` is 45° round and 1 radian is `point 1.27324 of c`, (because $4/\pi \simeq 1.27324$).

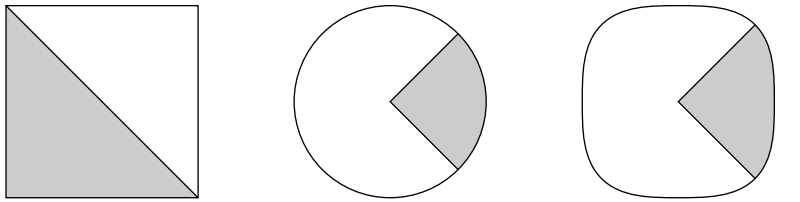


In a closed path, the point numbering in METAPOST wraps round: so in a circle, point n is the same as point $n + 8$; and in general point n is the same as point $n + \mathbf{length}\, p$. This works with negative numbers too, so we could use

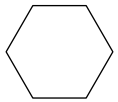
```
center p -- subpath(-1,1) of p -- cycle
```

to get wedge that extends either side of point 0. The same idea was used to draw the arrows in the first row:

```
drawarrow subpath(1/2, length p + 1/2) of p;
```



4.3 Regular polygons of a given radius



REGULAR POLYGONS with a given radius can be defined or drawn directly with a simple inline loop:

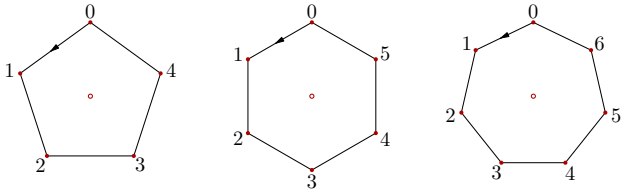
```
draw for i=0 upto 5: 20 dir 60i -- endfor cycle;
```

which works because `dir d` expands to `right rotated d`. But you might prefer to make a macro:

```
vardef polygon(expr n, r) =
    for i=0 upto n-1: (r, 0) rotated (360/n * i) -- endfor cycle
enddef;
```

This produces a closed path to represent an n-sided polygon that fits in a circle of radius r centred at the origin and that starts at `(r, 0)`, like the corresponding circular path, as shown in this polygonal version of the previous segment chart. → If you need polygon paths that start at the top, you can just swap the coordinates:

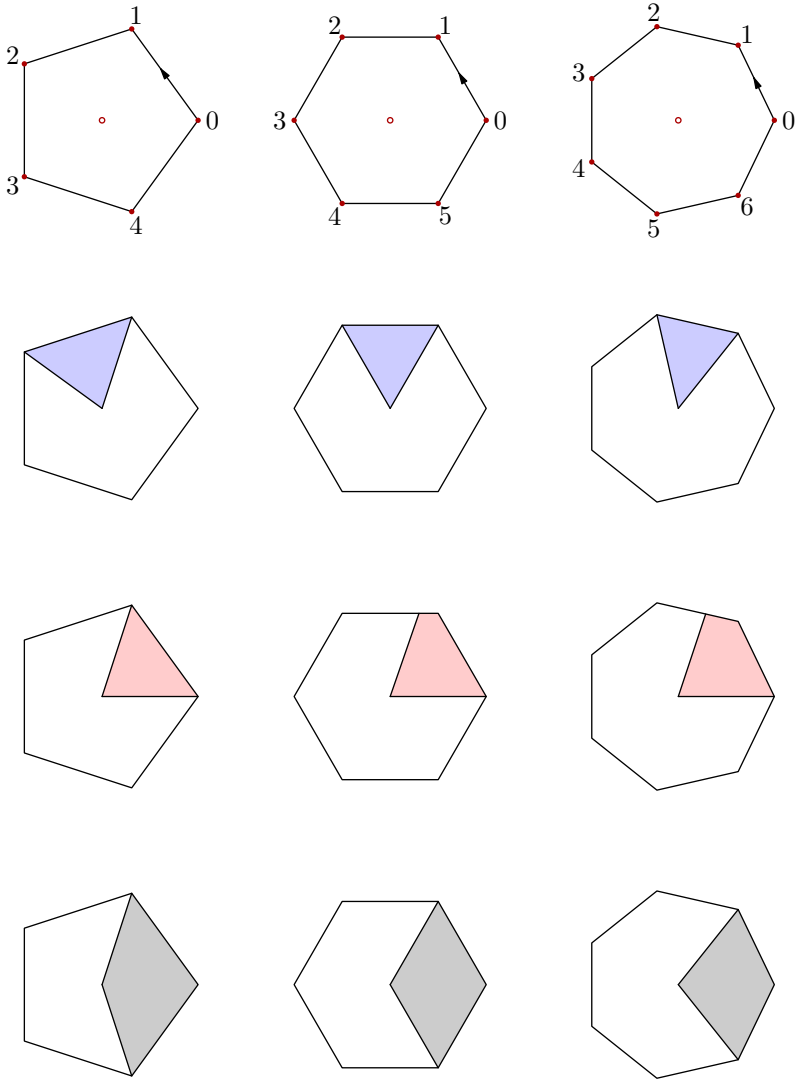
```
vardef polygon(expr n, r) =
    for i=0 upto n-1: (0, r) rotated (360/n * i) -- endfor cycle
enddef;
```



☞ Note also that some extra care is required to find the centres of these shapes. The `center` macro defined in `plain.mp` gives you the centre of the bounding box, but this is not the same as the centre of the polygon when the number of sides is odd. What you need instead is the geometric center or *centroid*:

```
vardef centroid primary p =
  origin for i=1 upto length p: + point i of p / length p endfor
enddef;
```

This should work for any closed path, not just regular polygons. For ways to label the vertices neatly, as shown above, see §12.5.



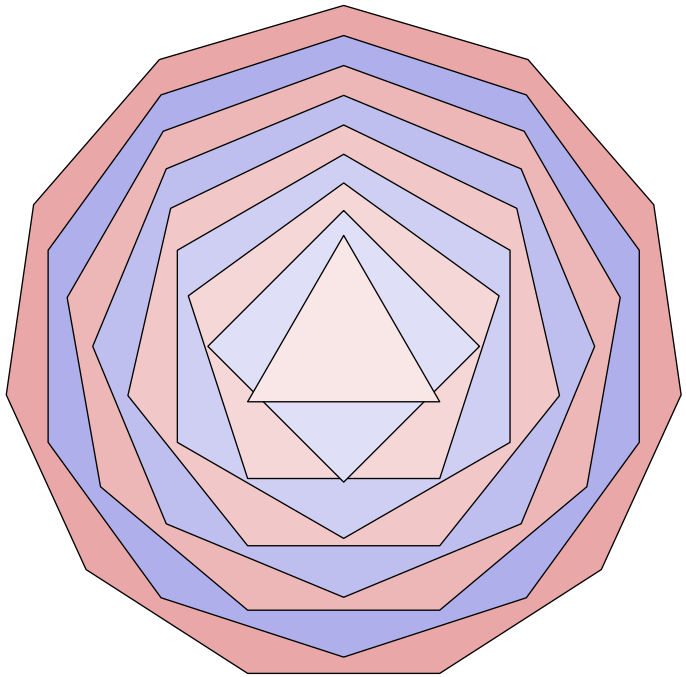
4.4 Regular polygons of a given side length

But you might want a polygon with a fixed side instead of a fixed radius. This needs a little trigonometry, using the sine rule:

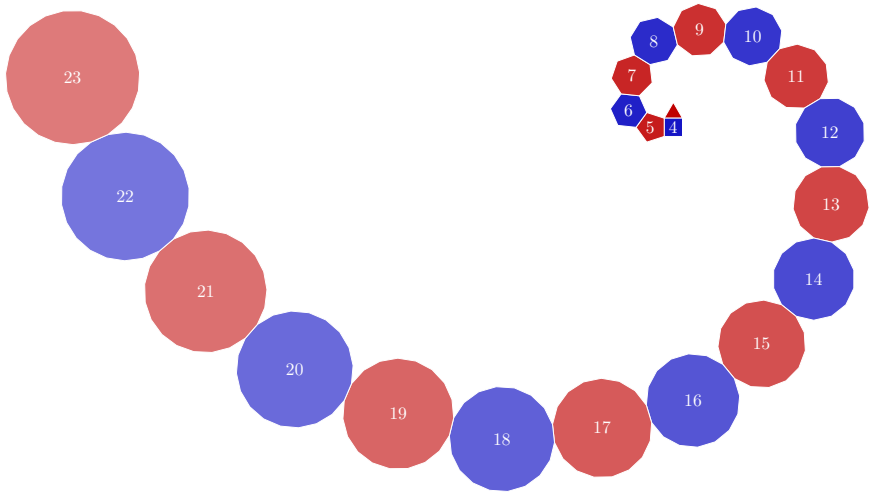
```
vardef polygon_with_side(expr n, s) =
    save a, b, r; numeric a, b, r;
    a * n = 360; a + 2b = 180; r = s * sind(b) / sind(a);
    for i = 0 upto n-1: (0, r) rotated (a * i) -- endfor cycle
enddef;
```

Which you can use like this to produce a nest of polygons ⟶

```
for n = 11 downto 3:
    path p; p = polygon_with_side(n, 72);
    fill p withcolor (n/32)[white, 3/4 if odd n: red else: blue fi];
    draw p;
endfor
```



These polygon paths are centred on $(0, 0)$ but sometimes it is more convenient to construct a polygon on a known segment rather than working out how to rotate and shift it into place.



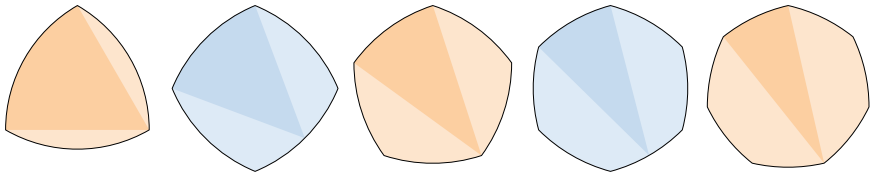
Here is a way to do that using the "of" syntax in the macro construction ⟶

```
vardef poly expr n of p =
  clearxy; z0 = point 0 of p; z1 = point 1 of p;
  for i=2 upto n-1:
    z[i] = z[i-2] rotatedabout(z[i-1], 360/n-180);
  endfor
  for i=0 upto n-1: z[i] -- endfor cycle
enddef;
beginfig(1);
  path P[]; P3 = for i=0 upto 2: 6 up rotated 120i -- endfor cycle;
  fill P3 withcolor 3/4 red; undraw P3;
  for n = 4 upto 23:
    numeric m; m = floor(n / 2);
    P[n] = poly n of subpath (m, m-1) of P[n-1];
    fill P[n] withcolor (n/48)[3/4 if odd n: red else: blue fi, white];
    undraw P[n]; label(decimal n, center P[n]) withcolor white;
  endfor
endfig;
```

4.5 Curved polygons

The regular polygons above are all defined with straight edges using the `--` connector that makes a tense path. If you changed each connector to `..` you would get a circle, and contrariwise, if you try `tensepath(fullcircle scaled 20)` you will get a regular octagon. But we can also adjust the directions at the corners to make a variety of closed polygon shapes with closed edges.

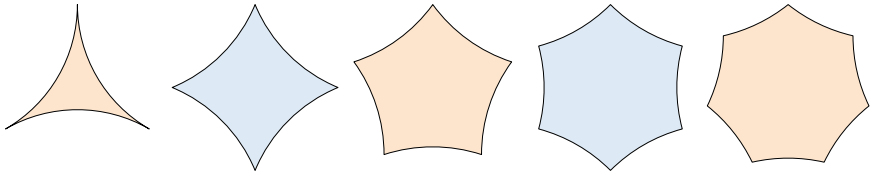
One of the most pleasing is the Reuleaux polygon, with circular arcs for edges.

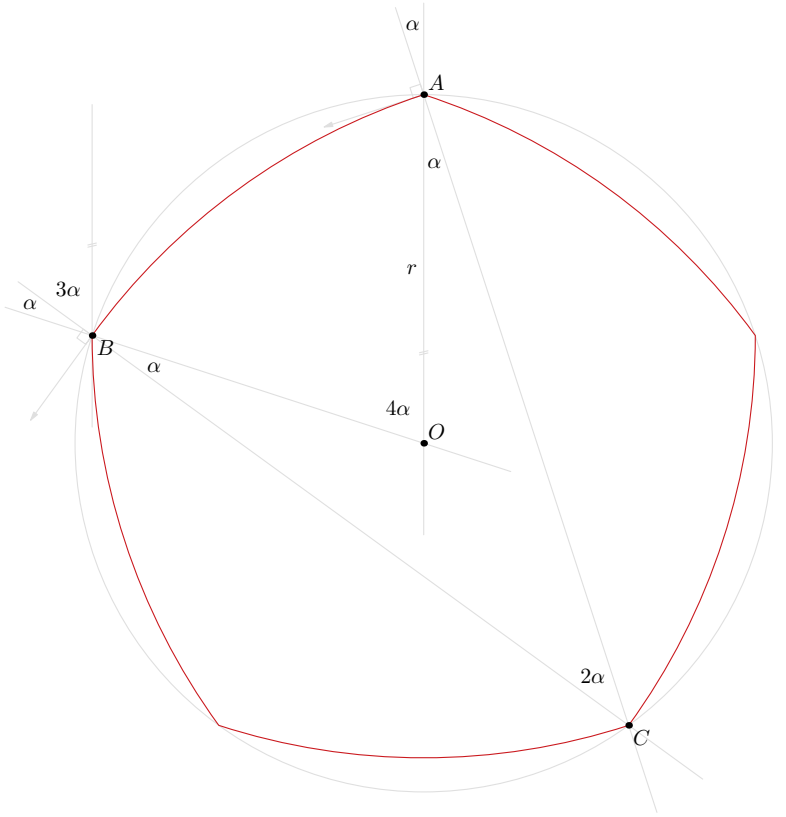


The figure on the right attempts to explain the geometry.

```
vardef reuleaux(expr n, r) =
  save a; numeric a; a = 90/n;
  for t = 0 step 4a until 359:
    (0,r) rotated t {left rotated (a+t)} .. {left rotated (3a+t)}
  endfor cycle
enddef;
```

If you swap the directions at each point you get shapes that are not quite like hypocycloids; play about a bit more to get flower shapes or windmills.





This proof only works for Reuleaux polygons with an odd number of sides, because otherwise the point C does not (quite) lie on the circle.

4.6 A triangle of Schläfli polygons

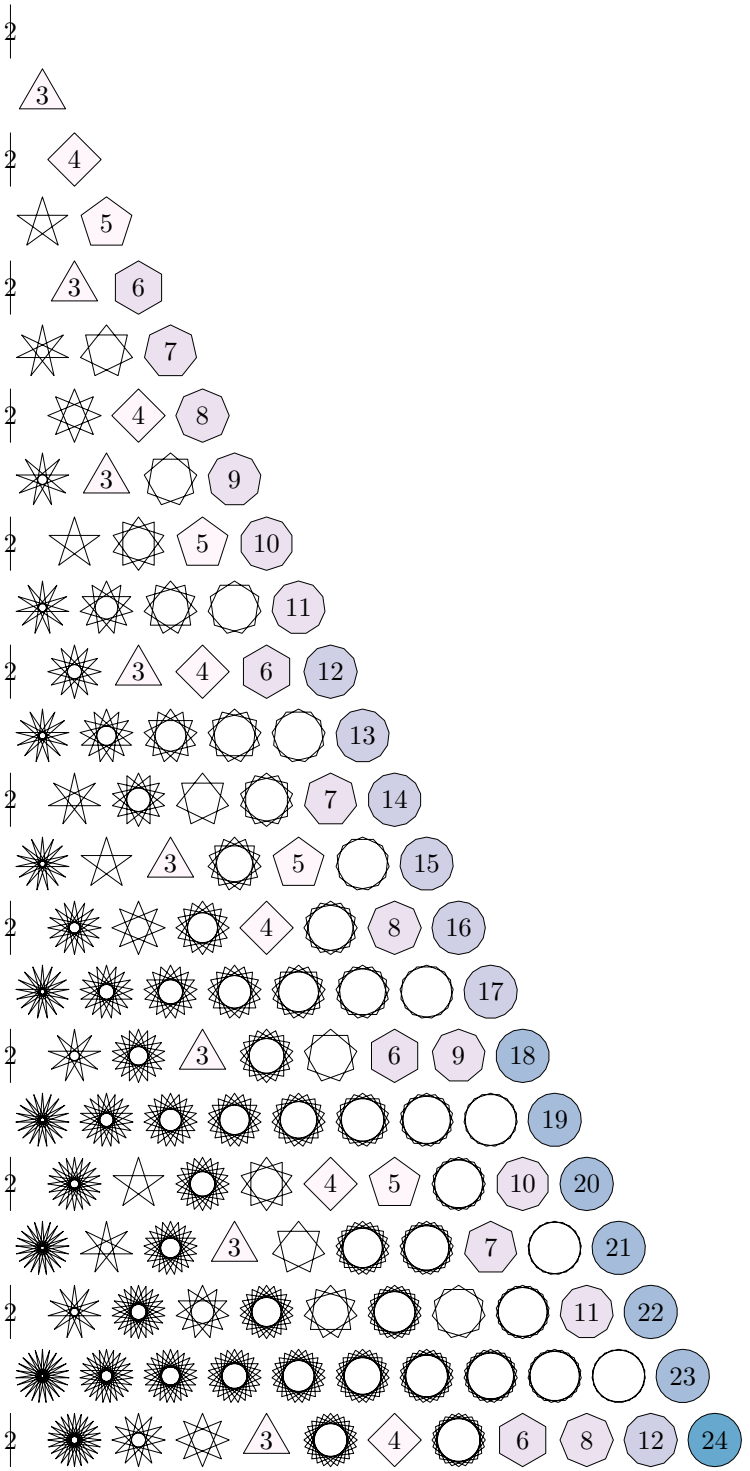
Apart from the curious polygon patterns in the display, the main METAPOST point of interest is the recursive `gcd` macro to find the greatest common divisor.

```
input colorbrewer-rgb

vardef gcd(expr a, b) =
  if b = 0: a else: gcd(b, a mod b) fi
enddef;

beginfig(1);
  for n=2 upto 24:
    for s=1 upto floor n/2:
      pair p; p = (12n - 24s, -24n);
      path gon; gon = for t=0 upto n/gcd(s,n) - 1:
        10 up rotated (360/n * s * t) --
      endfor cycle;
      if (n mod s = 0):
        fill gon shifted p withcolor PuBuGn[9][1+floor (n/s/6)];
        label("$" & decimal (n/s) & "$", p);
      fi
      draw gon shifted p withpen pencircle scaled 1/8;
    endfor
  endfor
endfig;
```

The macro also leads directly to an efficient way to find the least common multiple:

```
vardef lcm(expr a, b) = a / gcd(a, b) * b enddef;
```

As always in METAPOST, it is safer to divide as early as possible to reduce the chance of arithmetic overflow.

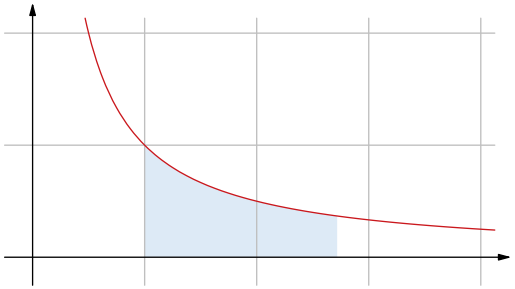


4.7 Building cycles from parts of other paths

Plain METAPOST has a built-in function to compute the intersection points of two paths, and there's a handy high level function called `buildcycle` that uses this function to create an arbitrary closed path. The arguments to the function are just a list of paths, and providing the paths all intersect sensibly, it returns a closed path that can be filled or drawn. This is often used for colouring an area under a function in a graph. Here is an example. The red line has been defined as path `f` and the two axes as paths `xx`, and `yy`. The light blue area was defined with

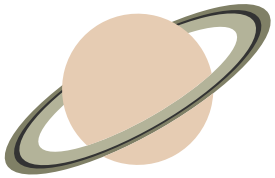
```
buildcycle(yy shifted (u,0), f, yy shifted (2.71828u,0), xx)
```

Note the re-use of the y-axis path shifted along by different amounts.



There are similar examples in the METAPOST manual, but `buildcycle` can also be useful in more creative graphics. Here's a second example that uses closed paths to give an illusion of depth to a simple graphic of the planet Saturn.

```
path globe, gap, ring[], limb[];
globe = fullcircle scaled 2cm;
gap   = fullcircle xscaled 3cm yscaled .8cm;
ring1 = fullcircle xscaled 4cm yscaled 1.2cm;
ring2 = ring1 scaled 0.93;
ring3 = ring1 scaled 0.89;
limb1 = buildcycle(subpath(5,7) of ring1, subpath(8,4) of globe);
limb2 = buildcycle(subpath(5,7) of gap, subpath(-2,6) of globe);
picture saturn; saturn = image(
  fill ring1 withcolor .1 red + .1 green + .4 white;
  fill ring2 withcolor .2 white;
  fill ring3 withcolor .1 red + .1 green + .6 white;
  unfill gap;
  fill limb1 withcolor .2 red + .1 green + .7 white;
  fill limb2 withcolor .2 red + .1 green + .7 white;
);
draw saturn rotated 30;
```



Notes

- The first five paths are just circles and ellipses based on `fullcircle`.
- The drawing is done inside an `image` simply so that the final result can be drawn at an angle
- `unfill gap` means: `fill gap withcolor background`
- The subpaths passed to `buildcycle` are chosen carefully to make sure we get the intersections at the right points and so that the component paths all run in the same direction. Note that `subpath (8,4) of globe` runs clockwise (that is backwards) from point 8 to point 4.

4.8 The implementation of `buildcycle`

The implementation of `buildcycle` in plain METAPOST is interesting for a number of reasons. Here it is copied from `plain.mp` (with minor simplifications) ⟶

```
vardef buildcycle(text input_path_list) =
  save ta, tb, k, j, pp; path pp[];
  k=0;
  for p=input_path_list: pp[incr k]=p; endfor
  j=k;
  for i=1 upto k:
    (ta[i], length pp[j]-tb[j])
      = pp[i] intersectiontimes reverse pp[j];
    if ta[i]<0:
      errmessage("Paths " & decimal i &
                 " and " & decimal j & " don't intersect");
    fi
    j := i;
  endfor
  for i=1 upto k:
    subpath (ta[i],tb[i]) of pp[i] ..
  endfor cycle
enddef;
```

Notice how freely the indentation can vary; this is both a blessing (because you can line up things clearly) and a curse (because the syntax may not be very obvious at first glance). Notice also the different ways we can use a `for`-loop. The first two are used at the 'outer' level to repeat complete statements (that end with semi-colons); the third one is used at the 'inner' level to build up a single statement.

The use of a `text` parameter allows us to pass a comma-separated list as an argument; in this case the list is supposed to be a list of path expressions that (we hope) will make up a cycle. The first `for` loop provides us with a standard idiom to split a list; in this case the comma-separated value of `input_path_list` is separated into into a more convenient array of paths called `pp` indexed by `k`. Note that the declaration of the array as ⟨path⟩ forces the argument to be a list of paths.

The second `for` loop steps through this array of paths looking for intersections. The index `j` is set to be `k` when `i=1`, and then set to the previous value of `i` at the end of the loop; in this way `pp[j]` is the path before `pp[i]` in what is supposed to be a cycle. The macro uses the primitive operator `intersectiontimes` to find the intersection points, if any. Note that we are looking for two path times: the time to start a subpath of the current path and the time to end a subpath of the previous path; the macro does this neatly by reversing the previous path and setting the *b*-point indirectly by subtracting the time returned from the length of the path.

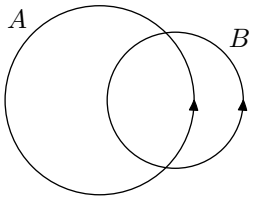
If all has gone well, then `ta` will hold all the start points of the desired subpaths, and `tb` all the corresponding end points. The third and final `for` loop assumes that this is indeed the case, and tries to connect them all together. Note that it uses `..` rather than `&` just in case the points are not quite co-incident; finally it finishes with a `cycle` to close the path even though point `tb[k] of pp[k]` should be identical (or at least very close) to point `ta[0] of pp[0]`.

This implementation of `buildcycle` works well in most cases, provided that there are enough components to the cycle of paths. If you only have two paths, then the two paths need to be running the same direction, and the start of each path must not be contained within the other. This is explored in the next section.

4.9 Strange behaviour of `buildcycle` with two closed paths

The implementation of `buildcycle` in plain METAPOST can get confused if you use it with just two paths. Consider the following example:

```
path A, B;
A = fullcircle scaled 2.5cm;
B = fullcircle scaled 1.8cm shifted (1cm,0);
fill buildcycle(A,B) withcolor .8[blue,white];
drawarrow A;
drawarrow B;
label.ulft(btex $A$ etex, point 3 of A);
label.urt(btex $B$ etex, point 1 of B);
```

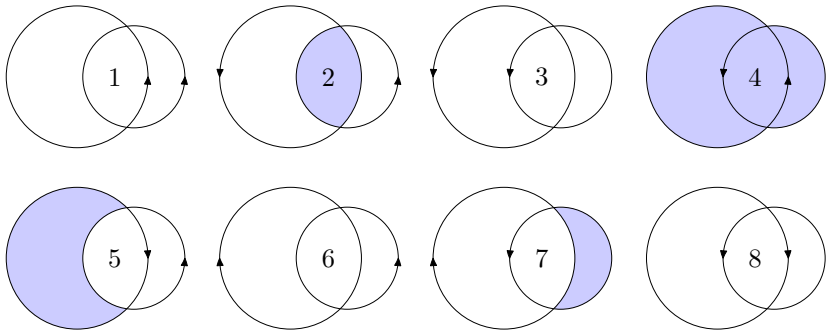


Where has the fill colour gone?

When we compile this example, we get no error message from `buildcycle`, but there is no fill colour visible in the output. The problem is that the points found by `buildcycle` are the same both times that it steps through the middle loop, so the closed path it returns consists of two identical (or very close) points and the so the fill has zero area.

Now observe what happens when we rotate and reverse each of the paths in turn. Number 1 corresponds to the example shown above; point 0 of A is inside the closed path B. In 2 we have rotated path A by 180° so that the start of path A is no longer inside B, and now `buildcycle` works 'properly' — but this is the only time it does so. In 3, we've rotated B by 180° as well, so that B starts inside A and as expected `buildcycle` fails. In 4 we've rotated A back to its original position, so that both paths start inside each other; and we get the union of the two shapes. In 5–8, we've repeated the exercise with path A reversed, and `buildcycle` fails in yet more interesting ways.

You could use this behaviour as a feature if you need to treat A and B as sets and you wanted to fill the intersection, union, or set differences, but if you just wanted the overlap, then you need to ensure that both paths are running in the same direction and that neither of them starts inside the other.



❖ To rotate a circular path, you can use: `p rotatedaround(center p, 180)`

4.10 Find the overlap of two closed paths

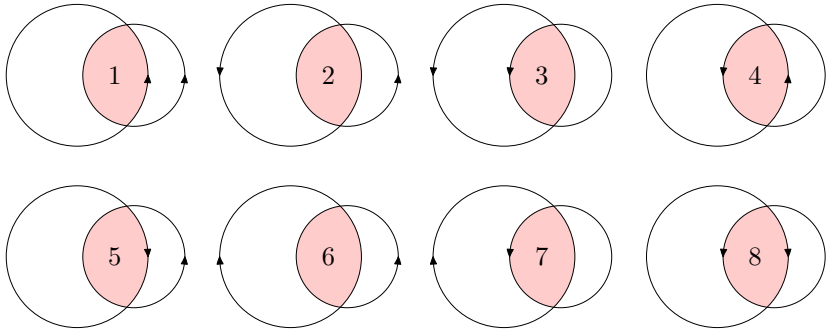
As we have seen, in order to get the overlap of two closed paths from `buildcycle`, we need both paths to be running in the same direction, and neither path should start inside the other one. It's not hard to create an `overlap` macro that does this automatically for us. The first element we need is a macro to determine if a given point is inside a given closed path. Following Robert Sedgwick's *Algorithms in C* we can write a generic `inside` function that works with any simple closed path. The approach is to extend a horizontal ray from the point towards the right margin and to count how many times it crosses the closed path; if the number is odd, the point must be inside.

Equipped with this function we can create an `overlap` function that first uses the handy `counterclockwise` function to ensure the given paths are running in the same direction, and then uses `inside` to determine where the start points are.

```
vardef front_half primary p = subpath(0, 1/2 length p) of p enddef;
vardef back_half primary p = subpath(1/2 length p, length p) of p enddef;
% a and b should be closed paths...
vardef overlap(expr a, b) =
  save A, B, p, q;
  path A, B; boolean p, q;
  A = counterclockwise a;
  B = counterclockwise b;
  p = not inside(point 0 of A, B);
  q = not inside(point 0 of B, A);
  if (p and q):
    buildcycle(A,B)
  elseif p:
    buildcycle(front_half B, A, back_half B)
  elseif q:
    buildcycle(front_half A, B, back_half A)
  else:
    buildcycle(front_half A, back_half B, front_half B, back_half A)
  fi
enddef;
```

Using this `overlap` macro in place of `buildcycle` produces less surprising results.

```
vardef inside(expr p, ring) =
  save t, count, test_line;
  count := 0;
  path test_line;
  test_line = p -- (infinity, ypart p);
  for i = 1 upto length ring:
     t := xpart(subpath(i-1,i) of ring
                  intersectiontimes test_line);
     if ((0<=t) and (t<1)): count := count + 1; fi
  endfor
  odd(count)
enddef;
```



5 Numbers

This section discusses plain METAPOST scalar numeric variables and what you can do with them. METAPOST inherits its unusual native system of scaled numbers from METAFONT; like many of Knuth's creations it is slightly quirky, but works very well once you get the hang of it. The original objective was to make METAFONT produce identical results on a wide variety of computers. By default all arithmetic is carried out using 28-bit integers in units of 1/65536. This is done automatically for you, so you don't need to worry about it, but you should be aware of a couple of practical implications:

- All fractions are rounded to the nearest multiple of $\frac{1}{65536}$, so negative powers of 2 ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ...) are exact, but other common fractions are not: for example $\frac{1}{3}$ is represented as $\frac{21845}{65536} \simeq 0.333328$, and $\frac{1}{10}$ as $\frac{6554}{65536} \simeq 0.100006$. You should bear this in mind particularly when you choose fractional step-values in a `for` loop; the errors can accumulate so that you may miss your expected terminal value.
- The system limits you to numbers that are less than 4096 in absolute value. This can be an irritation if you are trying to plot data with large values, but the solution is simple: scale your values to a reasonable range first.
- Intermediate calculations are allowed to be up to 32768 in absolute value before an error occurs. You can sometimes avoid problems by using the special Pythagorean addition and subtraction operators, but the general approach should be to do your calculations before you scale a path for filling or drawing.
- You can turn a number up to 32768 into a string using the `decimal` command, and then you could append zeros to it using string concatenation.

If you are using a recent version of METAPOST you can avoid all these issues by choosing one of the three new number systems: double, binary, or decimal, with the `numbersystem` command line switch. But beware that if you write programs that depend on these new systems, they might not be so portable as others. It's nice to have these new approaches just in case, but you will not need to use them very often.

Compare the following two snippets:

Code	Output

`for i = 0 step 1/10 until 1:`
`  show i;`
`endfor`
	`>> 0`
	`>> 0.1`
	`>> 0.20001`
	`>> 0.30002`
	`>> 0.40002`
	`>> 0.50003`
	`>> 0.60004`
	`>> 0.70004`
	`>> 0.80005`
	`>> 0.90005`

`for i = 0 step 1 until 10:`
`  show i/10;`
`endfor`
	`>> 0`
	`>> 0.1`
	`>> 0.2`
	`>> 0.3`
	`>> 0.4`
	`>> 0.5`
	`>> 0.6`
	`>> 0.7`
	`>> 0.8`
	`>> 0.9`
	`>> 1`

Unless you run this with `-numbersystem=decimal`, you will get 11 iterations in the second but only 10 with the first.

5.1 Numeric constants

Alongside the quirky number system, plain METAPOST also inherits three numeric constants from METAFONT: *infinity*, *epsilon*, and *eps*:

- *eps* is defined to be a small amount that is noticeable to METAFONT's rounding algorithms, namely $\frac{32}{65536} = \frac{1}{2048} \simeq 0.00049$. As a distance on the page or screen, it's invisible at any resolution less than 150,000 dots per square inch. If you were designing fonts in METAFONT, *eps* could help you avoid bad choices of pixels at low resolutions, but in METAPOST it's only really useful in comparisons that might suffer from rounding errors. *eps* is tiny, but it's bigger than any rounding error you may encounter, so you can safely test for equality with: $\mathbf{abs}(a-b) < \mathit{eps}$.
- *epsilon* is defined to be $\frac{1}{65536}$, the smallest positive scaled number.
- *infinity* is defined to be $4096 - \mathit{epsilon}$, which is the largest number you will normally deal with. This is useful when you just want a quantity larger than any other in the immediate vicinity. For an example, look at the definition of the `inside` function in section 4.10.

These three quantities retain (approximately) the same value even if you choose one of the alternative, higher precision, number systems. This is probably the most sane approach, but the constants lose their status as the smallest and largest numbers you can have.

The messy set of results shown on the right arises because `plain.mp` defines these constants like this (in version 1.005, which is current at the time of writing):

```
eps := .00049;    % this is a pretty small positive number
epsilon := 1/256/256;   % but this is the smallest
infinity := 4095.99998;    % and this is the largest
```

If you want cleaner constants, feel free to redefine the two decimals as:

```
eps := 1/2048;
infinity := 64*64-epsilon;
```

These definitions are equivalent with `scaled` numbers, but more consistent at higher precision. In particular they ensure that we always have $4096 = \mathit{infinity} + \mathit{epsilon}$ whichever number system is in use.

Running the toy program:

```
show numbersystem, eps, epsilon, infinity; end.
```

gives the following results with the different number systems:

```
>> "scaled"
>> 0.00049
>> 0.00002
>> 4095.99998

>> "double"
>> 0.0004899999999999998
>> 1.52587890625e-05
>> 4095.9999800000001

>> "binary"
>> 0.00048999999999999999999999999999999993
>> 0.0000152587890625
>> 4095.9999800000000000000000000000001

>> "decimal"
>> 0.00049
>> 0.0000152587890625
>> 4095.99998
```

5.2 Units of measure

In addition to the very small and very large numeric variables, plain METAPOST inherits eight more that provide a system of units of measure compatible with TEX. The definitions in `plain.mp` are very simple: ⟶

```
mm=2.83464;     pt=0.99626;     dd=1.06601;     bp:=1;
cm=28.34645;    pc=11.95517;    cc=12.79213;    in:=72;
```

When the output of METAPOST is set to be PostScript, then the basic unit of measure is the PostScript point. This is what TEX calls a `bp` (for 'big point'), and it is defined so that 1 inch = 72 bp. The traditional printers' point, which TEX calls a `pt`, is slightly smaller so that 1 inch = 72.27 pt.

Normal use of these units relies on METAPOST's implicit multiplication feature. If you write '$w = 10\ cm$;' in a program, then the variable w will be set to the value 283.4645. The advantage is that your lengths should be more intuitively understandable, but if you are comfortable thinking in PostScript points (72 to the inch, 28.35 to the centimetre) then there is no real need to use any of the units.

Bizarrely, 28.35 is also the number of grammes to the ounce.

It is sometimes useful to define your own units; in particular many METAPOST programs define something like '$u = 1$ cm;' near the start, and then define all other lengths in terms of u. If you later wish to make a smaller or larger version of the drawing then you can adjust the definition of u accordingly. Two points to note:

- If you want different vertical units, you can define something like '$v = 8\,mm$' and specify horizontal lengths in terms of u, but verticals in terms of v.
- If you want to change the definition of u or v from one figure to the next, you will either have to use '**numeric** u, v;' at the start of the your program in order to reset them, or use the assignment operator instead of the equality operator to overwrite the previous values.

The unit definitions in `plain.mp` are designed for use with the default scaled number system; if you want higher precision definitions, then you can update them by including something like this at the top of your program: ⟶

```
% exact values to re-define the plain.mp units
numeric bp, in, mm, cm, pt, pc, dd, cc;
72 = 72 bp = 1 in;
800 = 803 pt = 803/12 pc;
3600 = 1270 mm = 127 cm;
1238 pt = 1157 dd = 1157/12 cc;
```

The effect of the **numeric** keyword is to remove the previous definitions; the four equation lines then re-establish the units with very slightly more accurate definitions. You can safely use these definitions with `scaled`, as they are equivalent to the decimals currently given in `plain.mp`, but the main point of the example is to show how you can do implicit definitions with equations.

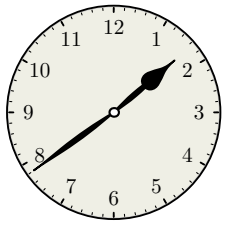
5.3 Integer arithmetic, clocks, and rounding

Native METAPOST provides nothing but a `floor` function, but `plain.mp` provides several more useful functions based on this.

- '**floor** x' returns $\lfloor x \rfloor$, the largest integer $\le x$. You can use `x=floor x` to check that x is an integer.
- '**ceiling** x' returns $\lceil x \rceil$, the smallest integer $\ge x$.
- 'x **div** y' returns $\lfloor x/y \rfloor$, integer division.
- 'x **mod** y' returns $x - y \times \lfloor x/y \rfloor$, integer remainder.

Note that **mod** preserves any fractional part, so $355/113$ **mod** $3 = 0.14159$.

This behaviour is usually what you want. For example we can use it to turn the time of day into an appropriate rotation for the hands of a clock. In the program given on the right, this idea is used to define functions that convert from hours and minutes to degrees of rotation on the clock. METAPOST provides two internal variables *hour* and *minute* that tell you the time of day when the current job started. The clock face shown here was generated using



beginfig(1); **draw** *clock*(*hour*, *minute*); **endfig**;

to give a sort of graphical time stamp.

There is also a `round` function that rounds a number to the nearest integer. It is essentially defined as **floor**$(x + 0.5)$ except that it is enhanced to deal with ⟨pair⟩ variables as well. If you round a pair the x-part and the y-part are rounded separately, so that **round**$(3.14159, 2.71828) = (3, 3)$.

The `round` function only takes a single argument, but you can use it to round to a given number of places by multiplying by the precision you want, rounding, and then dividing the result. So to round to the nearest eighth you might use '**round**$(x \times 8)/8$', and to round to two decimal places '**round**$(x \times 100)/100$'. The only restriction is that the intermediate value must remain less than 32767 if you are using the default number system.

```
path hand[];
hand1 = origin .. (.257,1/50) .. (.377,1/60)
        & (.377,1/60) {up} .. (.40,3/50)
        .. (.60, 1/40) .. {right} (.75,0);
hand1 := (hand1 .. reverse hand1 reflectedabout(left,right)
        .. cycle) scaled 50;
hand2 = origin  .. (.60, 1/64) .. {right} (.925,0);
hand2 := (hand2 .. reverse hand2 reflectedabout(left,right)
        .. cycle) scaled 50;

% hour of the day to degrees
vardef htod(expr hours)   = 30*((15-hours)   mod 12) enddef;
vardef mtod(expr minutes) =  6*((75-minutes) mod 60) enddef;

vardef clock(expr hours, minutes) = image(
  % face and outer ring
  fill fullcircle scaled 100 withcolor 1/256(240, 240, 230);
  draw fullcircle scaled  99 withcolor .8 white;
  draw fullcircle scaled 100 withpen pencircle scaled 7/8;
  % hour and minute marks
  for t=0 step  6 until 359:
    draw ((48,0)--(49,0)) rotated t;
  endfor
  for t=0 step 30 until 359:
    draw ((47,0)--(49,0)) rotated t withpen pencircle scaled 7/8;
  endfor
  % numerals
  for h=1 upto 12:
    label(decimal h infont "bchr8r", (40,0) rotated htod(h));
  endfor
  % hands rotated to the given time
  pickup pencircle scaled 7/8;
  filldraw hand1 rotated htod(hours+minutes/60);
  filldraw hand2 rotated mtod(minutes);
  % draw the center on top
    draw origin withpen pencircle scaled 5;
  undraw origin withpen pencircle scaled 3;
) enddef;
```

5.4 Integer powers

Occasionally you might get caught out by the implementation of the ** operator. As the table on the right shows, you may get an approximate answer from `x ** y` even when x and y are both integers. Note that the squares are all integers, and the powers of two appear to be ok (although if the page was wider you would see that `2**9` is 512.00002), but that with a couple of exceptions cubes and higher powers are all slightly off. Changing to one of the new number systems makes it worse; even x^1 is not always an integer. The reason can be found in the way that the ** operator is defined in `plain.mp`.

```
primarydef x ** y = if y = 2: x * x else: takepower y of x fi enddef;
def takepower expr y of x =
  if x > 0:
    mexp(y * mlog x)
  elseif (x = 0) and (y > 0):
    0
  else:
    if y = floor y:
      if y >= 0: 1 for n=1 upto y: * x endfor
      else:      1 for n=-1 downto y: / x endfor
      fi
    else:
      hide(errmessage "Undefined power: " & decimal x & "**" & decimal y)
    fi
  fi
enddef;
```

This is inherited directly from plain METAFONT, and as it says in the *The METAFONTbook*, it is optimized for x^2 and takes care to handle correctly negative numbers and zeros. But for all positive values of x other than 2 it is implemented using logs, and the results are therefore only approximate. To avoid confusion where this might matter (such as a particular offset into a recursively defined path) you could simply use `round(7**3)` to get a whole number, or if you are sure that your y values are all non-negative integers, you could temporarily replace the definition:

```
primarydef x ** y = 1 for n=1 upto y: * x endfor enddef;
```

x	x^2	x^3	x^4	x^5	x^6	x^7
2	4	8	16	32	64	128
3	9	27	81	243.00003	729.00009	2187.00024
4	16	64	256	1024.00003		
5	25	124.99998	624.99992	3124.99944		
6	36	216.00002	1296.0001			
7	49	343.00002	2401			
8	64	512.00002				
9	81	728.99995				
10	100	999.99992				
11	121	1331.00002				
12	144	1728.00012				
13	169	2196.99977				
14	196	2744				
15	225	3374.9998				
16	256					
17	289					
18	324					
19	361					

Results of `x**y` for small values, using the default `scaled` number system

6 Pairs, triples, and other tuples

METAPOST inherits a generalized concept of number from METAFONT that includes ordered pairs. Pairs are primarily used as Cartesian coordinates, but can also be used as complex numbers, as discussed below. METAPOST extends this generalization with 3-tuples and 4-tuples. Just like pairs, the elements in these tuples can take any numeric value, so in theory it would be possible to use them for three- and four-dimensional coordinates, but there are no built-in facilities for this in plain METAPOST, so some external library is needed. *All of the various attempts at three dimensions in METAPOST are rather difficult to use, so none of them is discussed in this document.*

Unlike simple numerics, the extended tuple variables are not automatically declared for you, so if you want to define points A and B you need to explicitly write '**pair** A, B;' before you assign values to them. Once you have declared them, you can equate them to an appropriate tuple using = as normal.

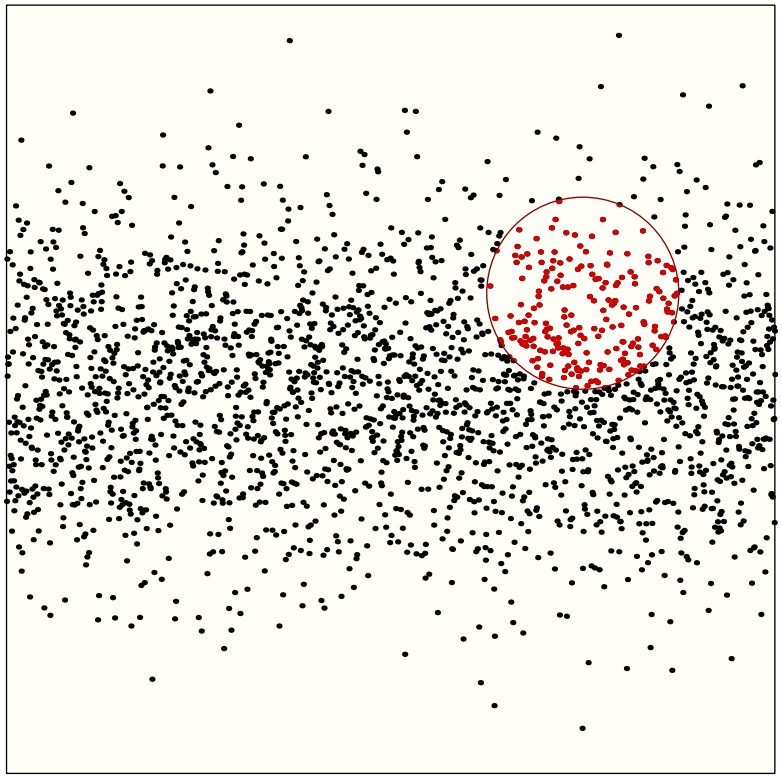
```
pair A,B; A = B = (1,2);
color R;  R = (1,2,3);
cmykcolor C; C = (1,2,3,4);
```

The normal use of triples and quads is for colours (RGB colours and CMYK colours); Triples are type **color**, quads are type **cmykcolor**. You can't have tuples of any other length, not even as constants, except for transforms.

A transform is how METAPOST represents an affine transformation such as `rotated 45 shifted (10,20)`. They are represented as 6-tuples, but if you try to write:

```
transform T; T = (1,2,3,4,5,6); % <-- doesn't work
```

you will get a parsing error (that complains about a missing parenthesis after the 4). You can examine and assign the individual parts using '**xpart** T' etc. More details below, and full details in the METAFONT book.



6.1 Pairs and coordinates

Now **pairs**: if you enclose two numerics in parentheses, you get a ⟨pair⟩. A pair generally represents a particular position in your drawing with normal, orthogonal Cartesian x- and y-coordinates, but you can use a pair variable for other purposes if you wish. As far as METAPOST is concerned it's just a pair of numerics.

METAPOST provides a simple, but slightly cumbersome, way to refer to each half of a pair. The syntax '**xpart** A' returns a numeric equal to the first number in the pair, while '**ypart** A' returns the second. The names refer to the intended usage of pair variable to represent pairs of x and y-coordinates. Note that they are read-only; you can't assign a value to an **xpart** or a **ypart**. So if you want to update only one part of a pair, you have to do something like this: $A := (42, \textbf{ypart}\ A)$;

In addition there is a neat macro definition in plain METAPOST that allows you do deal with the x- and y-parts of pairs rather more succinctly. The deceptively simple definition of z as a subscripted macro allows you to write `z1 = (10,20);` and have it automatically expanded into the equivalent of `x1=10;` and `y1=20;`. You can then use `x1` and `y1` as independent numerics or refer to them as a pair with `z1`. A common usage is to find the orthogonal points on the axes in graphs, like so ⟶

There is also a simple way to write coordinates using a polar notation using `dir`. This macro is defined so that `dir 30` expands to `right rotated 30` and then to `(1,0) rotated 30`, which becomes `(cosd(30), sind(30)` or `(0.86603, 0.5)`. So to get the polar notation point (r, θ), where r is the radius and θ is the angle in degrees counter-clockwise from the positive x-axis, you can write '`r * dir theta`'. As usual, with a constant you can omit the multiplication sign, so '`2 dir 30`' provides another way to define the point `(sqrt(3),1)`.

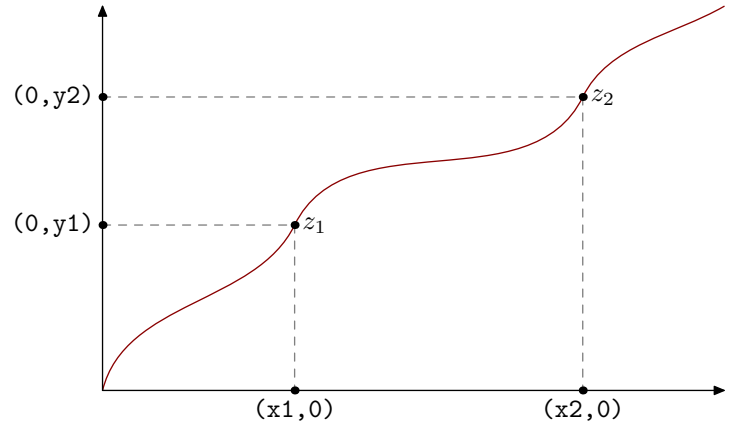
Plain METAPOST defines five useful pair variables: *origin*, *right*, *up*, *left*, and *down*. As so often, Knuth-Hobby definitions in `plain.mp` are quite illuminating ⟶ As you can see, pair variables can be used in implicit equations.

They can also be scaled using implicit multiplication, so writing '144 *right*' is equivalent to writing '$(144, 0)$' but possibly a bit more readable. In particular the idiom '**shifted** 200 *up*;' works well when applied to a point, a path, or an image. Unfortunately, this convenient notation does not work well with units of measure. This is because implicit multiplication only works between a numeric constant and a variable. So '2 *in right*' does not work as you might expect; you can write '2 *in* $*$ *right*' but by that stage it's probably simpler to write '$(2\ in, 0)$' or even just '$(144, 0)$'.

Plain METAPOST provides this definition

```
vardef z@#=(x@#,y@#) enddef;
```

which you can use to find orthogonal points.




```
% pair constants
pair right,left,up,down,origin;
origin=(0,0); up=-down=(0,1); right=-left=(1,0);
```

6.2 Pairs as complex numbers

As you might expect in a language designed by mathematicians, METAPOST's pair variables work rather well as complex numbers. To represent the number $3 + 4i$ you can write `(3,4)`. To get its modulus, you write `abs (3,4)` (which gives 5 in this case), and to get its argument, you write `angle (3,4)` (which gives 53.1301). Note that `angle` returns the argument in degrees rather than radians, and that the result is normalized so that $-180 < \mathbf{angle}(x, y) \leq 180$.

The standard notation for points supports this usage. You can write `z0=(3,4);` and then extract or set the real part with `x0` and the imaginary part with `y0`. If you want to use other letters for your variable names, you can use `xpart` and `ypart` to do the same thing. So after '`pair w; w=(3,4);`' you can get the real part with `xpart w` and the imaginary part with `ypart w`. You can also use the polar notation shown above to write complex numbers. For $re^{i\theta}$ you can write '`r * dir theta`' where `r` is the modulus and `theta` is the argument in degrees.

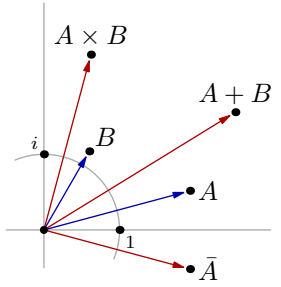
The predefined constants `up`, `down`, `left`, and `right` also provide points on the unit circle corresponding to i, $-i$, -1, and $+1$ respectively. It's tempting to define '`pair i; i=(0,1);`', so that you can write constants like `4i` directly, but this is not very helpful, because `3+4i` will give you an error since METAPOST does not let you add a `numeric` to a `pair`.

However METAPOST does let you add (and subtract) two pairs, so complex addition and subtraction are just done with the normal operators. To get the complex conjugate you could use `reflectedabout(left,right)`, but it's probably easier just to write `(x0,-y0)` or define a simple function:

```
def conj(expr z) = (xpart z, -ypart z) enddef;
```

Complex multiplication is provided as part of the core language by the `zscaled` operator. This is defined with the same precedence as `scaled` or normal scalar multiplication (which is what you usually want). So `(3,4) zscaled (1,2)` gives `(-5,10)` because $(3 + 4i) \times (1 + 2i) = 3 + 6i + 4i - 8 = -5 + 10i$. `zscaled` is only defined to work on two `pair` variables, so you can't write `(3,4) zscaled 4`. To get that effect with `zscaled` you would have to write `(3,4) zscaled (4,0)`, but this is the same as `(3,4) scaled 4`, which is usually simpler to write. If your pair is stored as a variable you can write (for example) `4 z0` to get the same effect. Or `1/4 z0` or `z0/4` for scalar division.

There are no other complex operators available, but it is not hard to implement the usual operations when they are required…




```
beginfig(1);
  numeric u; u = 1cm;
  z1 = 2 dir 15; z2 = 1.2 dir 60;
  z3 = z1+z2; z4 = z1 zscaled z2; z5 = (x1,-y1);
  drawoptions(withcolor 2/3 white);
  draw (1/2 left -- 3 right) scaled u ;
  draw (1/2 down -- 3 up   ) scaled u ;
  draw subpath (0,3) of fullcircle scaled 2u rotated -22.5;
  drawoptions();
  dotlabel.lrt (btex $\scriptstyle 1$ etex, (u,0));
  dotlabel.ulft(btex $\scriptstyle i$ etex, (0,u));
  interim ahangle := 30;
  forsuffixes @=1,2,3,4,5:
    x@ := x@ * u; y@ := y@ * u;
    drawarrow origin -- z@
      cutafter fullcircle scaled 5 shifted z@
      withcolor 2/3 if @ < 3: blue else: red fi;
  endfor
  fill fullcircle scaled dotlabeldiam;
  dotlabel.rt (btex $A$ etex, z1);
  dotlabel.urt(btex $B$ etex, z2);
  dotlabel.top(btex $A+B$ etex, z3);
  dotlabel.top(btex $A \times B$ etex, z4);
  dotlabel.rt (btex $\bar{A}$ etex, z5);
endfig;
```

6.2.1 Extra operators for complex arithmetic

Since multiplication by z can be thought of as a transformation consisting of rotation by the argument of z and scaling by $|z|$, you can define the complex inverse and complex square root simply using `angle` and `abs`.

First an inverse function. The idea here is to find a function that is the opposite of complex multiplication, so we want something that gives

```
z zscaled zinverse(z) = (1,0)
```

In other words you need to find a complex number with an argument that is the negative of the argument of z and a modulus that will scale $|z|$ to 1. You can use the polar notation with `dir` to write this directly:

```
vardef zinverse(expr z) = 1/abs z * dir - angle z enddef;
```

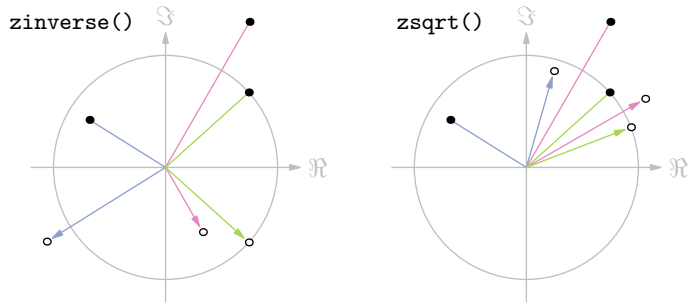
The complex division, z/w, can now be done as: `z zscaled zinverse(w)`. The only difficulty with this function is how it deals with zero, or rather with the point $(0,0)$. Since '`abs (0,0)`' gives 0, the function will give you a 'divide by zero' error if it's called with $(0,0)$. But this is probably what you want it to do, since there is no easy way to represent the point at infinity in the extended complex plane on paper.

For square root, you want a function '`zsqrt(z)`' that returns a complex number with half the argument of z and a modulus that is the square root of the modulus of z, so that '`zsqrt(z) zscaled zsqrt(z) = z`'. This does the trick:

```
def zsqrt(expr z) = sqrt(abs z) * dir 1/2 angle z enddef;
```

This function also has a difficulty with the point $(0,0)$, because `angle (0,0)` is not well defined, and so METAPOST throws an error. If you want a function that correctly returns $(0,0)$ as its own square root, then try something like this:

```
vardef zsqrt(expr z) =
  if abs z > 0: sqrt(abs z) * dir 1/2 angle fi z
enddef;
```

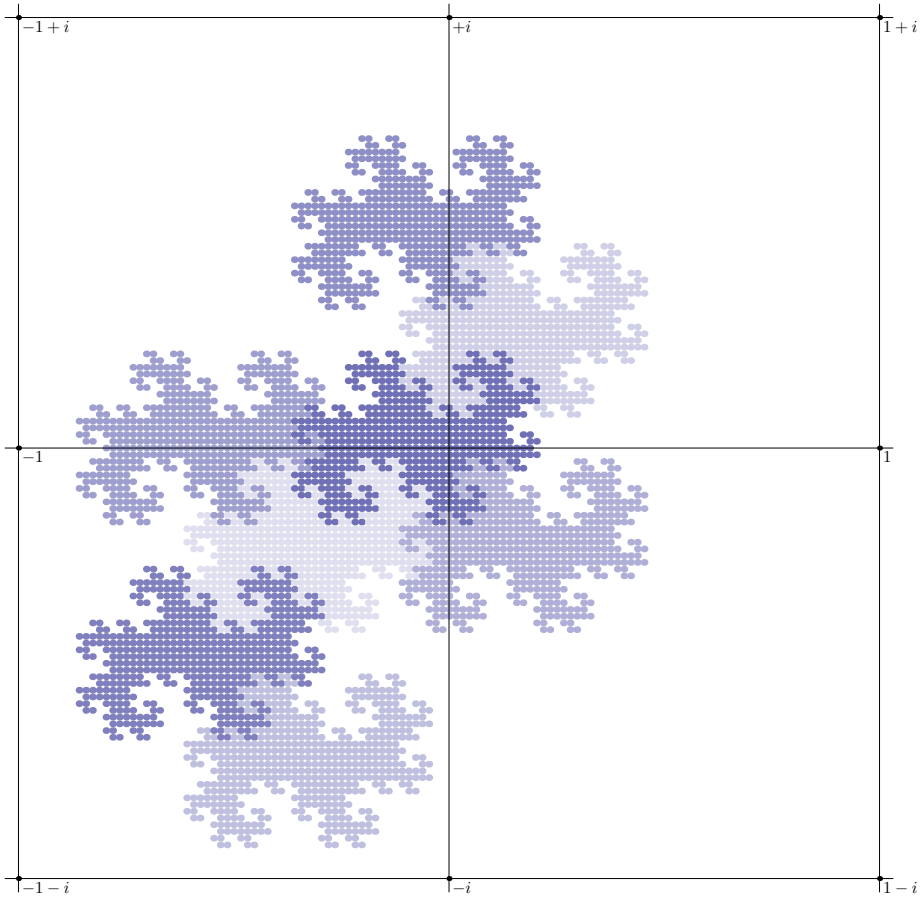


The drawing uses the two functions defined on the left.

```
z1 = 0.8 dir 148; z2 = 1.5 dir 60; z3 = 1.0 dir 42;

input colorbrewer-rgb
numeric u; u = 42; ahangle := 30;
picture axes; axes = image(
  path xx, yy; xx = (left--right) scaled 1.2 u; yy = xx rotated 90;
  draw fullcircle scaled 2u;
  drawarrow xx; label.rt("$\Re$", point 1 of xx);
  drawarrow yy; label.top("$\Im$", point 1 of yy);
);
vardef connect(expr a, b, shade) =
  save A, B; pair A, B; A = a scaled u; B = b scaled u;
  drawarrow A -- origin -- B
     cutafter fullcircle scaled 5 shifted B withcolor shade;
  draw A withpen pencircle scaled dotlabeldiam;
  draw B withpen pencircle scaled dotlabeldiam;
  draw B withpen pencircle scaled 2/3 dotlabeldiam withcolor white;
enddef;
picture P[];
P1 = image(draw axes withcolor 3/4;
  label.lrt("\small\texttt{zinverse()}", ulcorner axes shifted 10 left);
  connect(z1, zinverse(z1), SetTwo 7 3);
  connect(z2, zinverse(z2), SetTwo 7 4);
  connect(z3, zinverse(z3), SetTwo 7 5);
);
P2 = image(draw axes withcolor 3/4;
  label.lrt("\small\texttt{zsqrt()}", ulcorner axes);
  connect(z1, zsqrt(z1), SetTwo 7 3);
  connect(z2, zsqrt(z2), SetTwo 7 4);
  connect(z3, zsqrt(z3), SetTwo 7 5);
);
beginfig(1);
  labeloffset := 12; label.lft(P1, origin); label.rt(P2, origin);
endfig;
```

6.2.2 Using complex numbers to draw fractals

As an example of what you can do with complex arithmetic, here is a version of the diagram from §4.1 of Knuth's *Seminumerical Algorithms* showing S, the set of all points that can be written as $\sum_{k\geq 1} a_k(i-1)^{-k}$.



Note: you can adjust the "resolution" with the parameter t, but don't make it smaller than 1 if you are using the default number system; the diagram looks a bit strange unless t is an integer power of 2.

```
vardef fizz(expr X) =
  pair m, n; m = right; n = origin;
  numeric x; x = X;
  forever:
    exitif x = 0;
    m := m zscaled (-1/2, -1/2);
    if odd x:
      n := n + m;
    fi
    x := x div 2;
  endfor
  n
enddef;
beginfig(1);
  numeric s, t; s = 256; t = 4;
  for n=0 upto (s/t*s/t-1):
    draw fizz(n) scaled s
      withpen pencircle scaled t
      withcolor ((7+n mod 8)/16)[1/2 blue, white];
  endfor;
  path xx, yy;
  xx = (left--right) scaled (s+8);
  yy = xx rotated 90;
  for i=-1 upto 1:
    draw xx shifted (0, s*i) withpen pencircle scaled 1/8;
    draw yy shifted (s*i, 0) withpen pencircle scaled 1/8;
    if i=0:
      dotlabel.lrt("$-i$", (i, -1) scaled s);
      dotlabel.lrt("$+i$", (i, +1) scaled s);
    else:
      dotlabel.lrt("$" & decimal i & "-i$", (i,-1) scaled s);
      dotlabel.lrt("$" & decimal i & "$", (i, 0) scaled s);
      dotlabel.lrt("$" & decimal i & "+i$", (i, 1) scaled s);
    fi
  endfor
endfig;
```

7 Colours

METAPOST implements colours as simple numerics, or tuples of three or four numeric values. Three-tuples (which are type `color`) represent RGB colours; four-tuples (which are type `cmykcolor`) represent CMYK colours. Simple numerics are used to represent grey scale colours.

The numeric values of the colours can take any **numeric** value, but METAPOST only considers the range 0 to 1 — values less than zero are treated as zero, values greater than 1 are treated as 1. So British Racing Green with RGB code `(1,66,37)`, or Pillar Box Red with code `(223,52,57)`, can be defined like this:

```
color brg, pbr;
brg = (0.00390625, 0.2578125, 0.14453125);
pbr = (0.87109375, 0.203125, 0.22265625);
```

or, slightly more idiomatically:

```
brg = 1/256 (1, 66, 37);
pbr = 1/256 (223, 52, 57);
```

As you can see, you can apply implicit multiplication to a `color`, so after the declaration above `2 brg` would be a valid colour, although you have to think a bit to know what that means in terms of colour in your drawings.

Plain METAPOST defines five basic colour constants: `red`, `green`, `blue`, `white`, `black`. These are quite useful with leading fractions: `2/3 red` gives a nice dark red, that is good for drawing lines you want to emphasize; `1/2 white` gives you a shade of grey; and so on. But since `black` is defined as `(0,0,0)`, `1/2 black` just gives you `black`.

You can also add up `colors`. So `red + 1/2 green` gives you a shade of orange; this is more long-winded than writing `(1, 0.5, 0)` but maybe slightly easier to read. Much more usefully, you can use the mediation notation to get a colour that is part way between two others. So `1/2[red, white]` gives you a shade of pink, and `2/3[blue, white]` a sort of sky blue. You can also use this idea to vary colour with data, as in `(r)[red, blue]` where `r` is some calculated value. Here's a toy example:

To use RGB hex strings, you'll need to write a function:

```
vardef hexrgb(expr Spec) =
  save r, g, b;
  numeric r, g, b;
  r = hex substring (1,3) of Spec;
  g = hex substring (3,5) of Spec;
  b = hex substring (5,7) of Spec;
  1/256(r,g,b)
enddef;
brg = hexrgb("#014225");
pbr = hexrgb("#df3439");
```

☞ The `hex` function is a built-in primitive operation.

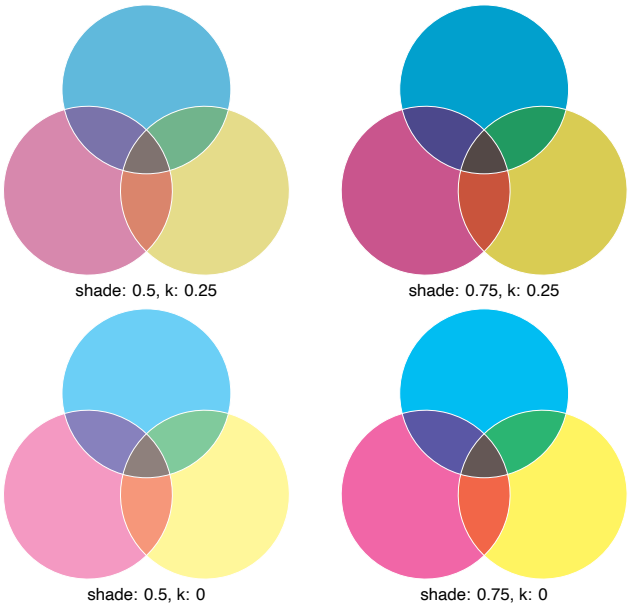
```
color brg, pbr;
brg = 1/256 (1, 66, 37);  % British Racing Green
pbr = 1/256 (223, 52, 57);  % Pillar Box Red
N = 5; n = 0;
for y=1 upto N:
  for x=1 upto N:
    fill fullcircle scaled 16 shifted 20(x,y)
      withpen pencircle scaled 2
      withcolor (n/N/N)[pbr, brg];
    label(decimal incr n infont "phvr8r", 20(x,y))
      withcolor white;
  endfor
endfor
```

7.1 CMYK colours

METAPOST also implements a CMYK colour model, using tuples of four numerics. In this model the four components represent cyan, magenta, yellow, and black. White is `(0,0,0,0)` and black is anything where the last component is 1.

Beware that the five colour constants defined in `plain.mp` are defined as RGB colours, and you can't mix colour models, so anything like `1/2[(1,1,0,0), white]` will not work, unless you redefine the color constants as CMYK colours:

```
cmykcolor black, white, red, green, blue;
black = (0,0,0,1); white = (0,0,0,0);
red = (0,1,1,0); green = (1,0,1,0); blue = (1,1,0,0);
```



☞ The apparent blending of colours here is done by calculating the overlaps and filling them in order. With plain `mpost`, there is no support for transparency in any of the colour models; but `luamplib` gives you access to PDF transparency, see §12.7.

```
path C[], B[];

% arrange each circle so that point 0 is outside the others
C1 = fullcircle scaled 100 rotated 90 shifted 40 up;
C2 = C1 rotated 120;
C3 = C2 rotated 120;

% the illusion of blended colours is helped by buildcycle
B0 = buildcycle(C1, C2, C3);
B1 = buildcycle(C1, C2);
B2 = buildcycle(C2, C3);
B3 = buildcycle(C3, C1);

picture P;
for x=0 upto 1:
  for y=0 upto 1:
    P := image(
      s := 1/2 + x/4;
      k := 0 + y/4;
      fill C1 withcolor s*(1,0,0,k);
      fill C2 withcolor s*(0,1,0,k);
      fill C3 withcolor s*(0,0,1,k);
      fill B3 withcolor s*(1,0,1,k);
      fill B2 withcolor s*(0,1,1,k);
      fill B1 withcolor s*(1,1,0,k);
      fill B0 withcolor s*(1,1,1,k);
      undraw C1; undraw C2; undraw C3;
    ) shifted (200x, 180y);
    draw P;
    label.bot(("shade: " & decimal s & ", k: " & decimal k)
                infont "phvr8r", point 1/2 of bbox P);
  endfor
endfor
```

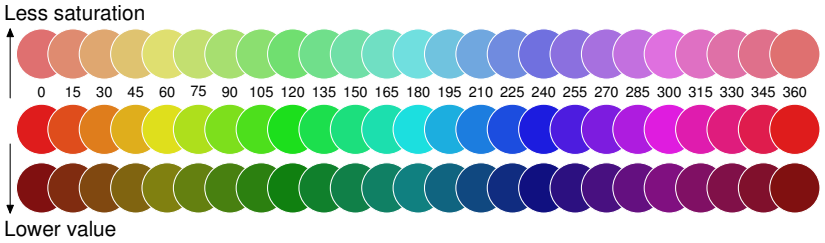
7.2 HSV colours

HSV colours are colours defined by a triple of hue, saturation, and value. Unlike RGB and CMYK colours there is no native support in METAPOST but it is possible to write a routine that maps HSV triples into RGB colours:

```
vardef hsv_color(expr h, s, v) =
  save chroma, hh, x, m;
  chroma = v * s;
  hh = h / 60;
  x  = chroma * (1 - abs(hh mod 2 - 1));
  m  = v - chroma;
  if     hh < 1: (chroma,x,0)+(m,m,m)
  elseif hh < 2: (x,chroma,0)+(m,m,m)
  elseif hh < 3: (0,chroma,x)+(m,m,m)
  elseif hh < 4: (0,x,chroma)+(m,m,m)
  elseif hh < 5: (x,0,chroma)+(m,m,m)
  else:          (chroma,0,x)+(m,m,m)
  fi
enddef;
```

This is based on information from the Wikipedia article on on “HSL and HSV”.

The hue values in HSV colours map nicely to the familiar spectrum of the rainbow. In the model used here 0 is red, 120 green, and 240 blue:



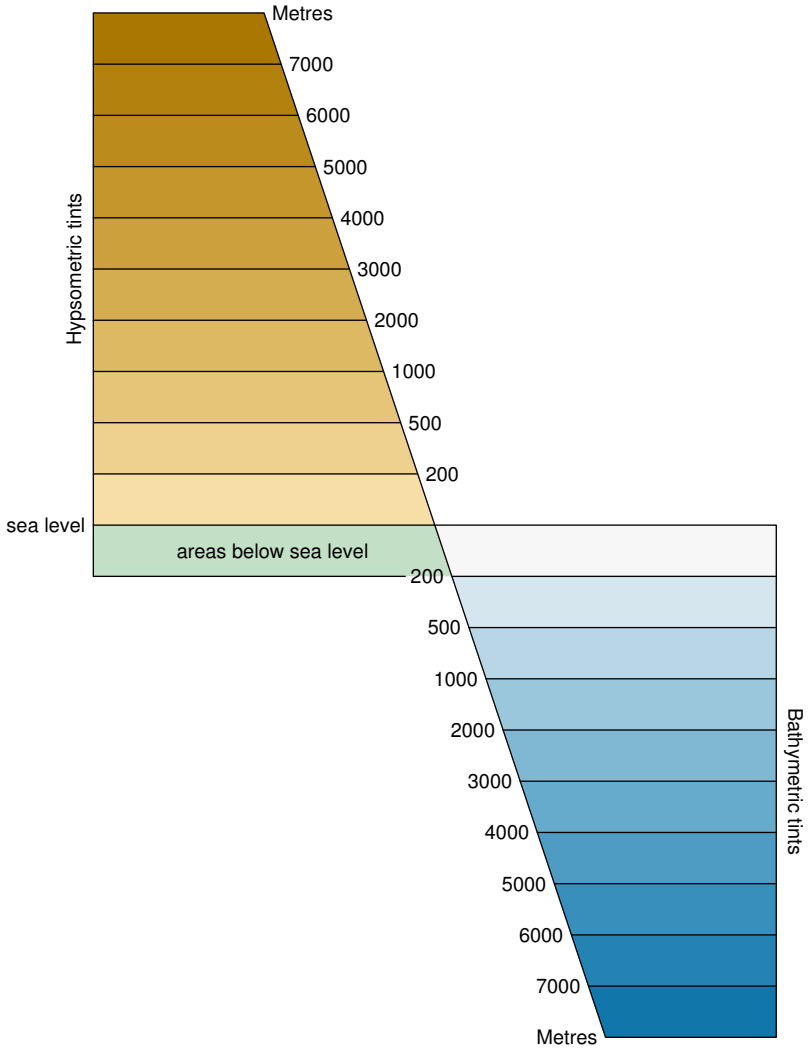
With less saturation the colours look faded; if you lower the value they get darker. Once you get the hang of them, they make choosing colours rather easier. You can produce ranges of colour by changing hue, or make gradations of a single colour by changing the saturation or value.

```
input color-hsv-macro
beginfig(1);
  defaultfont := "phvr8r";

  numeric s[], v[];
  s0 = 1/2; v0 = 7/8;
  s1 = 7/8; v1 = 7/8;
  s2 = 7/8; v2 = 1/2;

  path r;
  for y=0, 1, 2:
    for h=0 step 15 until 360:
      r := fullcircle scaled 24 shifted (h, -36y if y=2: +8 fi);
      fill r withcolor hsv_color(h, s[y], v[y]);
      draw r withcolor white;
      if y=1:
        label(decimal h infont defaultfont scaled 0.7, (h,-18));
      fi
    endfor
  endfor

  label.urt("Less saturation", (-20,14));
  label.lrt("Lower value", (-20,-78));

  drawarrow (-15, -21) -- (-15,12);
  drawarrow (-15, -43) -- (-15,-76);
endfig;
```

An HSV example of a graduated scale

This example requires the `hsv_color` macro from the previous page.

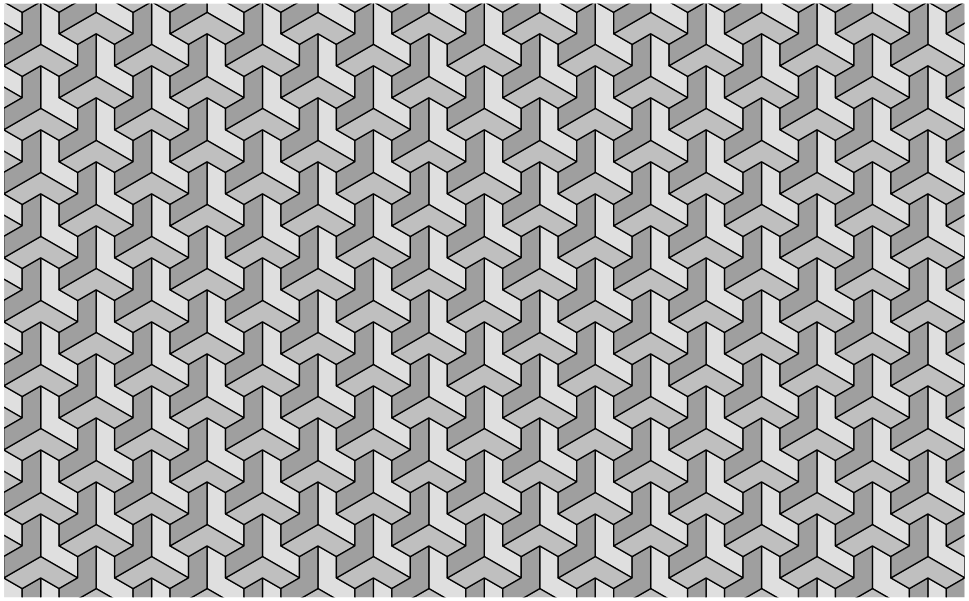
```
defaultfont := "phvr8r"; defaultscale := 3/4;
path h,d,b; numeric n; n = 10;
h = ((-2,0)--(0,0)--(-1,3)--(-2,3)--cycle) scaled 64;
d = h rotated 180;
b = subpath (0,1) of h -- point 1+1/n of d --
    (xpart point 0 of h, ypart point 1+1/n of d) -- cycle;
fill b withcolor hsv_color(123, 1/8, 7/8);
draw subpath (2.13,4) of b;
for i=1 upto n:
  fill point 4-(i-1)/n of h -- point 1+(i-1)/n of h --
       point 1+i/n of h -- point 4-i/n of h -- cycle
       withcolor hsv_color(42, 1/4 + 3/4 * i/n, 1 - i/3n);
  fill point 4-(i-1)/n of d -- point 1+(i-1)/n of d --
       point 1+i/n of d -- point 4-i/n of d -- cycle
       withcolor hsv_color(200, i/n - 1/n, 1 - i/3n);
endfor
string s;
for i=1 upto n-1:
  draw point 4-i/n of h -- point 1+i/n of h;
  draw point 4-i/n of d -- point 1+i/n of d;
  s := decimal if i < 4: (i**2+1) else: (10 + (i-3)*10) fi & "00";
  label.rt(s, point 1+i/n of h);
  label.lft(s, point 1+i/n of d);
endfor
label.rt("Metres", point 2 of h);
label.lft("Metres", point 2 of d);
label.lft("Hypsometric tints" infont defaultfont
    scaled defaultscale rotated 90, point 7/2 of h);
label.rt("Bathymetric tints" infont defaultfont
    scaled defaultscale rotated -90, point 7/2 of d);
label.lft("sea level", point 0 of h);
label("areas below sea level", center b);
draw h; draw d;
```



7.3 Grey scale

The `withcolor` command will also take a single `numeric` instead of a 3-tuple or a 4-tuple. This produces a colour in grey scale (or gray scale if you prefer the Webster spellings). Just as for the other colour types, values below 0 count as zero and values above 1 count as one. And since the smallest possible positive number in plain METAPOST is: *epsilon* = 1/256/256; then you can have at most 65,536 shades in between.

Grey scale is appropriate for some printed media, and can make effective textures and patterns. The pattern below was produced by this program ⟶

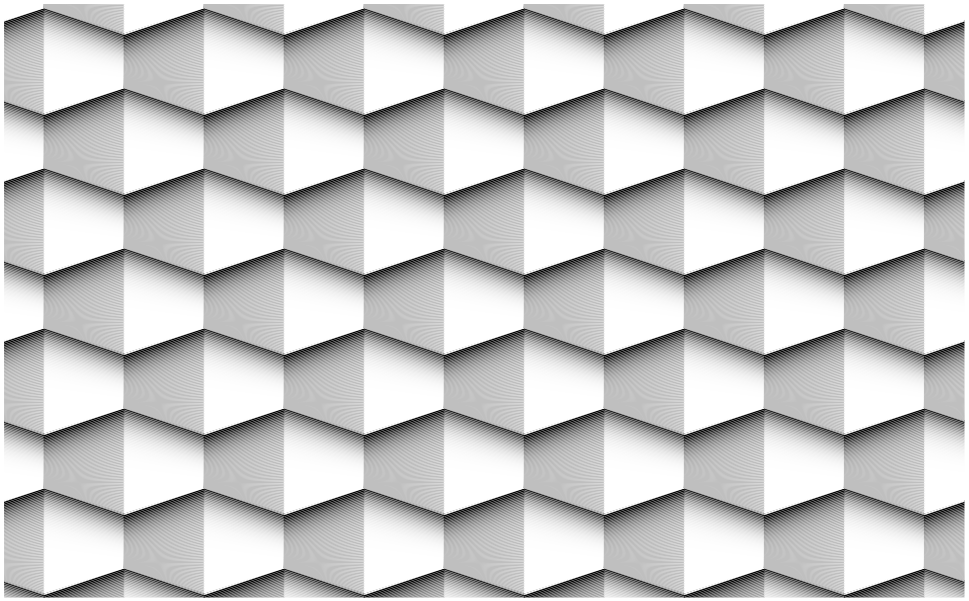


First a basic path (named *atom*) is defined, then in the first loop three picture variables, p_1, p_2, and p_3, are defined, each one rotated 120° from the previous and filled with a slightly darker shade of grey. The double loop then draws the three versions of the shape on an up-and-down grid. Finally the picture is clipped to a neat rectangle.

```
beginfig(1);
  numeric s; s = 8;
  path atom;
  atom = origin
  -- (2s,0) rotated -30 -- (2s,0) rotated -30 + (0,s)
  -- ( s,0) rotated  30 -- ( s,0) rotated  30 + (0,s)
  -- (0,2s) -- cycle;

  picture p[];
  for i=0 upto 2:
    p[i] = image(
      fill atom rotated -120i withcolor (7/8 - 1/8i) ;
      draw atom rotated -120i;
    );
  endfor

  pair u, v;
  u = point 3 of atom - point 1 of atom rotated -120;
  v = u rotated 60;

  n = 13;
  for i=-n upto n:
    for j=-n upto n:
      forsuffixes $=0,1,2:
        draw p$ shifted (i*u + j*v);
      endfor
    endfor
  endfor

  clip currentpicture to
      unitsquare shifted -(1/2,1/2) scaled 5in yscaled 0.618;
endfig;
```

Drawing algorithmic shadows

Here is a more complex pattern, showing one way to create an illusion of shadows with multiple fine lines.



```
beginfig(1);
path b, w;
b = ((-3,-4)--(3,-2)--(3,+2)--(-3,4)--cycle) scaled 5;
w = b reflectedabout(up, down);

picture B, W;
B = image(
  for i=0 step 1/64 until 1:
    draw point 4-i of b -- point 1+i**2 of b withcolor 1-i**8;
  endfor
);

W = image(
  for i=0 step 1/64 until 1:
    draw point 4-i of w -- point 1+i**2 of w withcolor 3/4-i**8;
  endfor
);

for i=-7 upto 7:
  for j=-4 upto 4:
    draw if odd (i+j): W else: B fi shifted ((i,j) scaled 30);
  endfor
endfor

clip currentpicture to
  unitsquare shifted -(.5,.5) scaled 5in yscaled 0.618;

endfig;
```

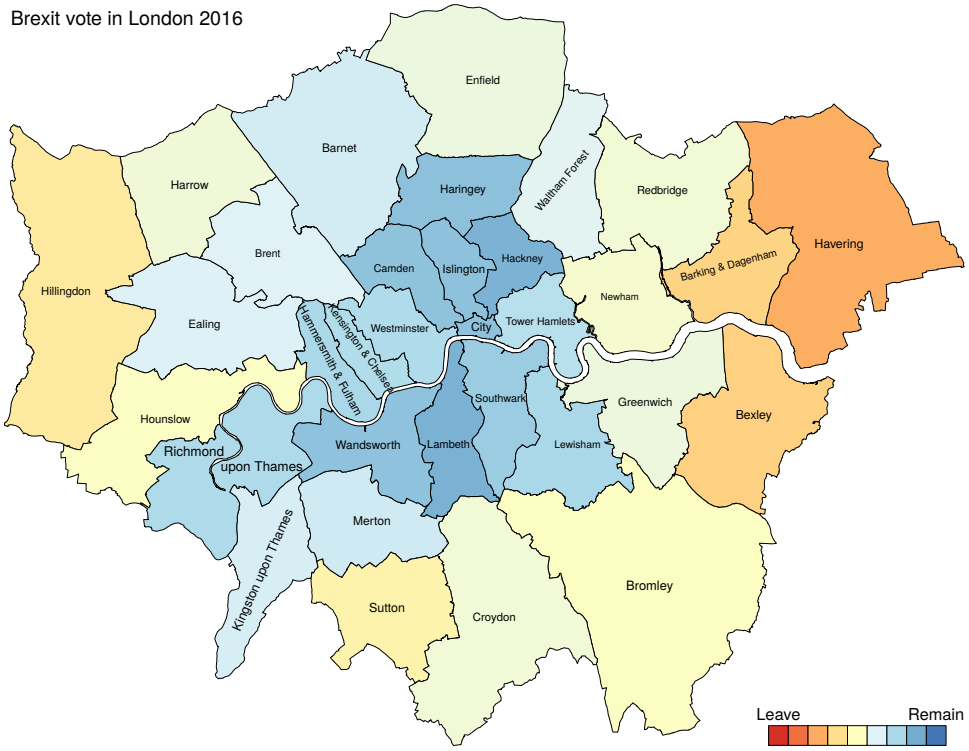
The first part defines two wedge-shaped closed paths, w being the mirror image of b. Like the standard *unitsquare* path, the path b is defined so that point 0 is the bottom left corner.

The two ⟨picture⟩ variables are produced by drawing lines across the shapes from bottom to top. If you set the loop step small enough, these multiple lines blend smoothly to give an even colour. And by using higher powers of the index variable, an effective shadow can be drawn 'bunched up' into the top of each shape. Note that METAPOST likes integer powers of two.

By repeating them alternately in a grid, we get an effective texture, which is clipped at the end to a neat rectangle again.

7.4 Colorbrewer palettes

The well-known Colorbrewer website (`http://colorbrewer2.org`) provides a useful set of colour palettes that are suitable for a wide range of applications. They were originally written for maps, but they are useful for many other types of drawing. If you are using an up-to-date, and complete, TEX distribution, you should find that my implementation of them for METAPOST is already installed on your system, otherwise you can get it from `https://ctan.org/pkg/metapost-colorbrewer`. The package provides two files that define all the colour ranges; one for CMYK and another for RGB; an example of usage is shown below on the right.



This map shows the `RdYlBu[9]` palette in action on a map of the Brexit vote in London. The outlines are the 33 London boroughs, and the colours show how we voted. The size of the labels shows the turnout. The data and the outlines are from public domain sources. They were prepared for METAPOST using various scripts, and they are available in the source for this document.

Here is the code for the palette used as the legend:

```
input colorbrewer-rgb

for i = 1 upto 9:
  path s; s = unitsquare scaled 12 shifted (12i, 0);
  fill s withcolor RdYlBu[9][i]; draw s;
  if i = 1:
    label.top("Leave", point 5/2 of s);
  elseif i = 9:
    label.top("Remain", point 5/2 of s);
  fi
endfor
```

8 Random numbers

METAPOST provides us with two built-in functions to generate random numbers.

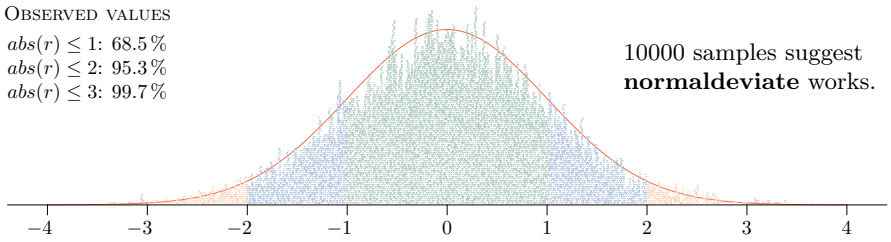
- '**uniformdeviate** n' generates a random real number between 0 and n.

  Note that the n is required. It can be negative, in which case you get negative random numbers; or it can be zero, but then you just get 0 every time. In other words the implementation generates a number r such that $0 \le r < 1$ and then multiplies r by n.

  If you want a random whole number, use '**floor**' on the result. So to simulate six-sided dice, you can use '1 + **floor uniformdeviate** 6'.

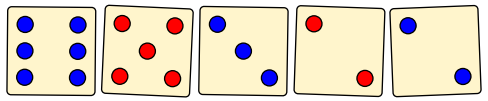
  If you use the new number systems, and you find that the numbers generated are all multiples of $n/4096$, so **uniformdeviate** 8192 (for example) generates even integers instead of random real numbers, then you should update your TEX distribution. This 'feature' was an accident of the original way that the scaled arithmetic routines were adapted.

- '**normaldeviate**' generates a random real number that follows the familiar normal distribution. The algorithm used is discussed in *The Art of Computer Programming*, section 3.4.1. If you generate enough samples, the mean should be approximately zero, and the variance about 1. The chance of getting a number between −1 and 1 is about 68.3%; between −2 and 2, about 95.4%.



  To relocate the mean, just add a constant. To rescale the distribution, multiply by the desired standard deviation (the square root of the desired variance).

```
vardef dice(expr pip_count, pip_color) =
  save d,r,p, ul, ur, lr, ll; r = 1/8;
  path d; d = for i=0 upto 3:
     quartercircle scaled 3 shifted (15,15) rotated 90i --
  endfor cycle;
  picture p; p = image(
     fill fullcircle scaled 6 withcolor pip_color;
     draw fullcircle scaled 6;
  );
  pair ul, ur, ll, lr;
  ul = 1/5[ulcorner d, lrcorner d]; ur = 1/5[urcorner d, llcorner d];
  ll = 4/5[urcorner d, llcorner d]; lr = 4/5[ulcorner d, lrcorner d];
  image(
     fill d withcolor (1, 0.96, 0.8); draw d;
     if odd(pip_count):
        draw p shifted center d;
     fi;
     if pip_count > 1:
        draw p shifted ul; draw p shifted lr;
     fi;
     if pip_count > 3:
        draw p shifted ur; draw p shifted ll;
     fi;
     if pip_count = 6:
        draw p shifted 1/2[ul,ll]; draw p shifted 1/2[ur,lr];
     fi
  )
enddef;
beginfig(1);
for i=0 upto 4:
  draw dice(1+floor uniformdeviate 6, if odd i: red else: blue fi)
    rotated (2 normaldeviate) shifted (36i,0);
endfor
endfig;
\end{mplibcode}
```



8.1 Random numbers from other distributions

The **normaldeviate** function is provided as a primitive METAPOST operation. The implementation is based on the 'Ratio method' presented in *The Art of Computer Programming*, section 3.4.1. It turns out to be very straightforward to implement the algorithm for this method as a user-level program ⟶

This version of `normaldeviate` is of academic interest only, in all real code you should use the primitive operation, but there are a couple of programming notes. If you put '`u = uniformdeviate 1`', then you have $0 \le u < 1$, so v/u might give you a divides-by-zero error; using '`u = 1 - uniformdeviate 63/64`' ensures that $1/64 < u \le 1$, which not only avoids the possibility of a divide-by-zero error, but also ensures that $|(v/u)| < 64$, so that you can square it without overflow. This is a useful general technique, and justified in terms of the algorithm since large values of v/u are rejected anyway. Secondly, the expression `sqrt(8/mexp(256))` is a constant ($\sqrt{8/e} \simeq 1.71553$) and could be replaced by its value, but this does not make an appreciable improvement to the speed of the routine. On a modern machine, this routine is only very slightly slower than using the primitive function.

It is also fairly straightforward to implement random number generators that follow other statistical distributions. The mathematical details are in the section of *TOACP* referenced above. Two examples, for the exponential distribution and the gamma distribution, are shown on the right. In both cases, note the care required to avoid arithmetic overflow with the default scaled number system.

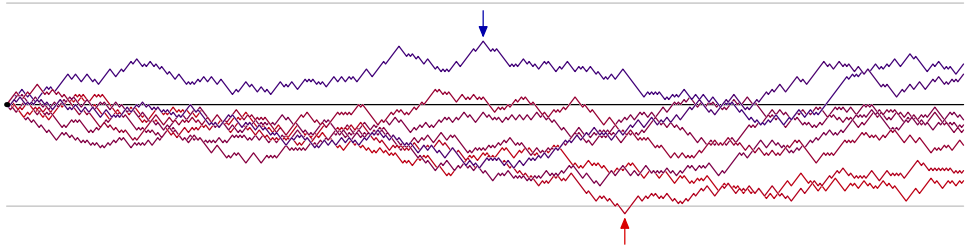
You can also see the special nature of METAPOST's **mexp** and **mlog** functions. They are defined so that $\mathbf{mexp}\, x = \exp(x/256)$ and $\mathbf{mlog}\, x = 256 \log(x)$. This is another artefact of the scaled number system. METAPOST computes x^y using the formula `mexp(y*mlog(x))`, and the adjusted log values give more accurate results. Note that this means that you have $e = \mathbf{mexp}(256)$.

It is sometimes useful to define macros for the usual versions of exp and log as shown on the right. This not only helps you make fewer programming mistakes, and

```
vardef normaldeviate =
  save u, v, xa;
  forever:
    u := 1 - uniformdeviate 63/64;
    v := sqrt(8/mexp(256)) * (-1/2 + uniformdeviate 1);
    xa := v / u;
    exitif (xa * xa <= -mlog(u)/64);
  endfor
  xa
enddef;

vardef exponentialdeviate expr mu =
  save u;
  u = 1 - uniformdeviate 1;  % hence 0 < u <= 1 and so you avoid
  -mu * 1/256 mlog(u)        % the danger of calling mlog(0)
enddef;

vardef tand(expr theta) =
  save x, y;
  (x, y) = dir theta;
  if abs(x) < eps: infinity else: y/x fi
enddef;
vardef exp(expr x) = mexp(256x) enddef;
vardef log(expr x) = 1/256 mlog(x) enddef;

% this is defined only for a > 1
vardef gammadeviate(expr a, b) =
  save y, x, v, s, accept; boolean accept;
  s = sqrt(2a - 1);
  forever:
    forever:
      y := tand(uniformdeviate 180);
      exitif abs(y) < 16;
    endfor
    x := s * y + a - 1;
    accept := false;
    if x > 0:
      v := uniformdeviate 1;
      if v <= (1 + y * y) * exp((a - 1) * log(x / (a - 1)) - s * y):
        accept := true;
      fi
    fi
    exitif accept;
  endfor
  x/b
enddef;
```

8.2 Random walks

You can use the random number generation routines to produce visualizations of random walks, with various levels of analysis.



In this example the random walk lines are coloured according to the final y-value, and the global maximum and minimum points are marked.

Each walk is created with an 'inline' for-loop; the loop is effectively expanded before the assignment, so that each *walk* variable becomes a chain of connected (x, y) pairs. Inside the loop you can conceal yet more instructions in a '**hide**' block. These instructions contribute nothing to the assignment, but can change the values of variables outside the block.

Note the first line of the **hide** block adds ± 1 to y with equal probability. You can (of course) create different kinds of random walks, by changing the way you set this delta value, for example by using a different type of random variate, or scaling the value, or changing the odds in favour of one direction or the other. For example:

```
y := y if uniformdeviate 1 < p: + 2 else: - 1 fi;
```

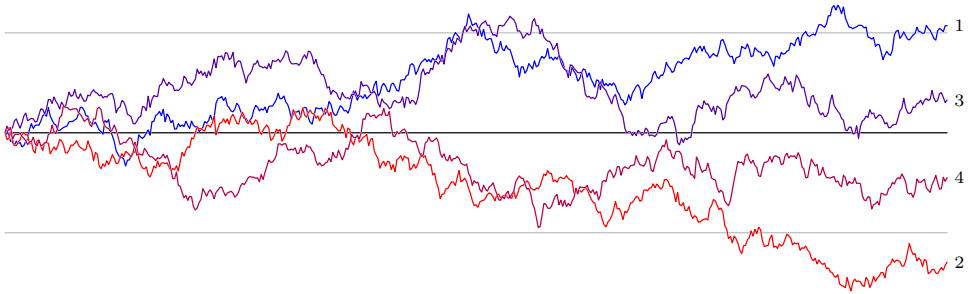
will set the delta to $+2$ with probability p and and to -1 with probability $1 - p$.

```
beginfig(1);
  numeric w, h, n; w = 377; h = 80; n = 500;

  draw (origin--right) scaled w;
  draw (origin--right) scaled w shifted (0,+h/2) withcolor 3/4;
  draw (origin--right) scaled w shifted (0,-h/2) withcolor 3/4;

  pair zenith, nadir; zenith = nadir = origin;
  color gain, lose; gain = .67 blue; lose = .85 red;
  for i=1 upto 8:
    path walk;
    numeric y; y = 0;
    walk = origin for x=w/n step w/n until w:
      hide(
        y := y if uniformdeviate 1 < 1/2: + else: - fi 1;
        if y > ypart zenith: zenith := (x,y) ; fi
        if y < ypart nadir:  nadir  := (x,y) ; fi
      )
      -- (x,y)
    endfor;
    undraw walk withpen pencircle scaled 3/4;
    draw walk withcolor (1/2+y/h)[lose, gain];
  endfor
  draw origin withpen pencircle scaled 2;
  drawarrow (12 up -- 2 up) shifted zenith withcolor gain;
  drawarrow (12 down -- 2 down) shifted nadir  withcolor lose;
endfig;
```

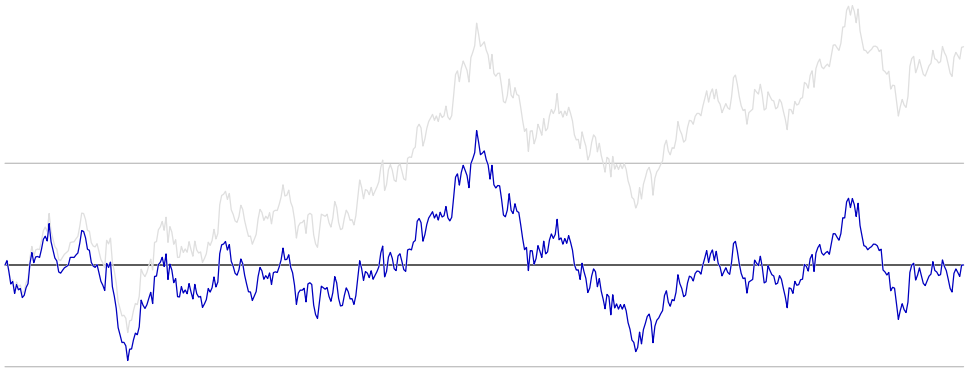
☞ Note the `undraw` line using a slightly thicker pen; this makes it easier to follow the lines as they cross each other.

8.2.1 Random walks with different constraints

FORMALLY, a random walk is constrained to move one unit at a time, but if you relax that constraint and use '**normaldeviate**' in place of '**uniformdeviate**' you may get more "realistic" patterns.



Alternatively you could add an extra constraint that the final value should be zero (or some other desired target value); this is the so-called "Brownian bridge".



To do this, you make a random walk path, as above, with n points, and then copy it into a new path where the i-th point is adjusted by $i/n \times (t - y)$ where t is the target value, and y is the y-value of the last point on the walk path.

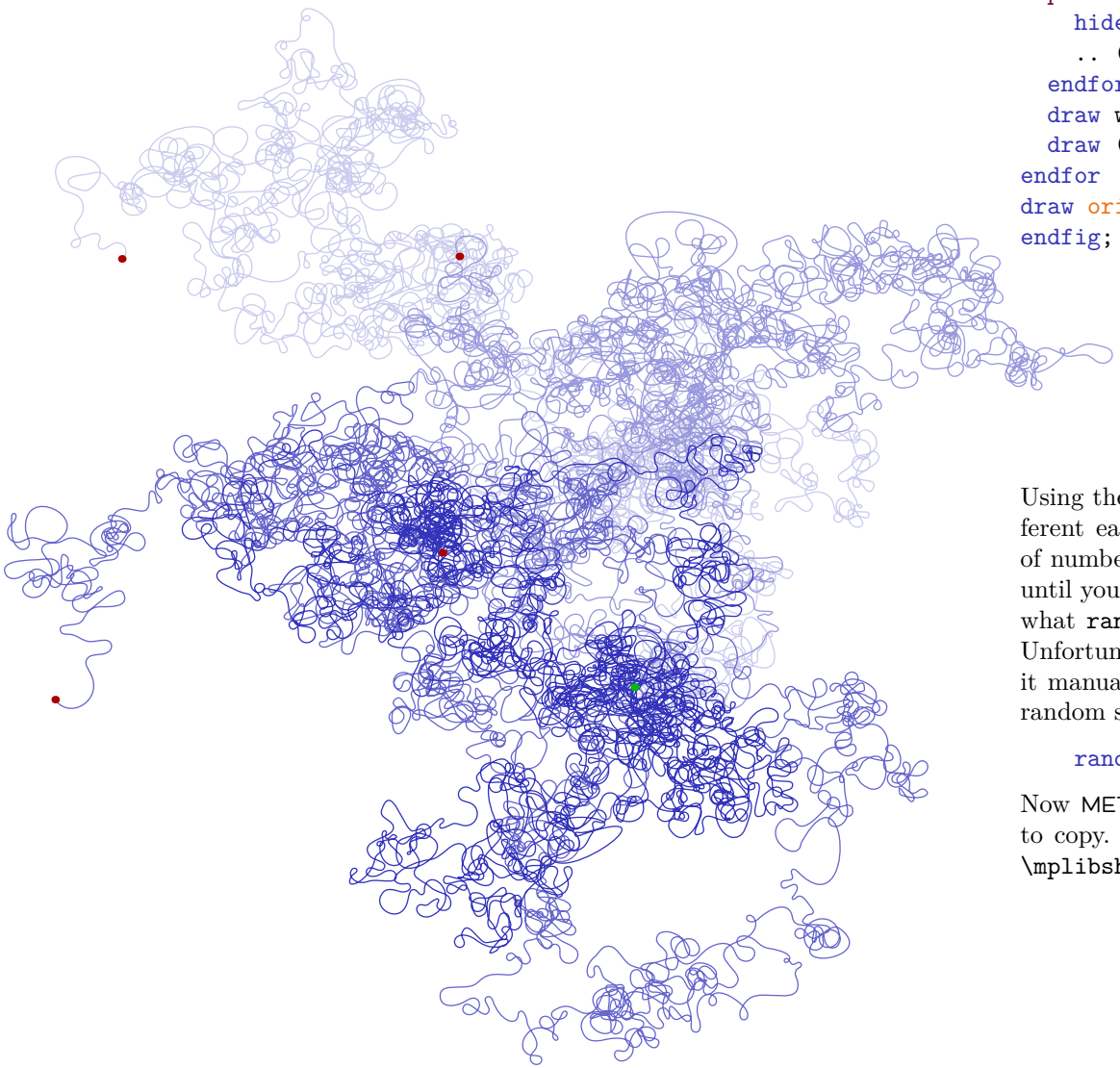
The drawing here is much the same as the previous page, except that the definition of *walk* in the central loop is simplified to this:

```
path walk; walk = (0, y) for x=w/n step w/n until w:
  hide(y := y + 2 normaldeviate)
  -- (x, y)
endfor;
```

```
beginfig(1);
  randomseed := 3612.11746;
  numeric w, h, n; w = 377; h = 80; n = 500;
  draw (origin--right) scaled w;
  draw (origin--right) scaled w shifted (0,+h/2) withcolor 3/4;
  draw (origin--right) scaled w shifted (0,-h/2) withcolor 3/4;
  numeric y; y = 0;
  path walk; walk = (0, y) for x=w/n step w/n until w:
    hide(y := y + 4 normaldeviate)
    -- (x, y)
  endfor;
  path bridge; bridge = point 0 of walk for i=1 upto n:
    -- point i of walk + (0, i/n * -y) endfor;
  draw walk withcolor 7/8;  % so you can see how it works
  undraw bridge withpen pencircle scaled 3/4;
  draw bridge withcolor 3/4 blue;
endfig;
```

8.3 Brownian motion

Relaxing all the constraints, can give you even more interesting patterns. If you also allow the x-coordinates to wander at random as well as the y-coordinates you get two-dimensional random patterns. And if you replace the straight line segments `--` with `..` so that METAPOST draws a smooth curve through the points, then the result is almost artistic.



```
beginfig(1);
% randomseed := uniformdeviate infinity;
randomseed := 2141.34242;
numeric u, v; u = 5; v = 4;
for n=1 upto 4:
  numeric x, y; x = y = 0;
  path w; w = (x, y) for i=1 upto 2048:
    hide(x := x + normaldeviate * u ; y := y + normaldeviate * v)
    .. (x, y)
  endfor;
  draw w withcolor (2n/10)[white, 2/3 blue];
  draw (x, y) withpen pencircle scaled 3 withcolor 2/3 red;
endfor
draw origin withpen pencircle scaled 3 withcolor 3/4 green;
endfig;
```

Using these random number generators means that the output is different each time because METAPOST produces a different sequence of numbers. You may find yourself running the program a few times until you find one you like. At this point you will wish that you knew what `randomseed` had been used, so that you can re-create picture. Unfortunately METAPOST does not log the value used, unless you set it manually. So here's a trick to use in this situation: set your own random seed using a random number at the top of your program.

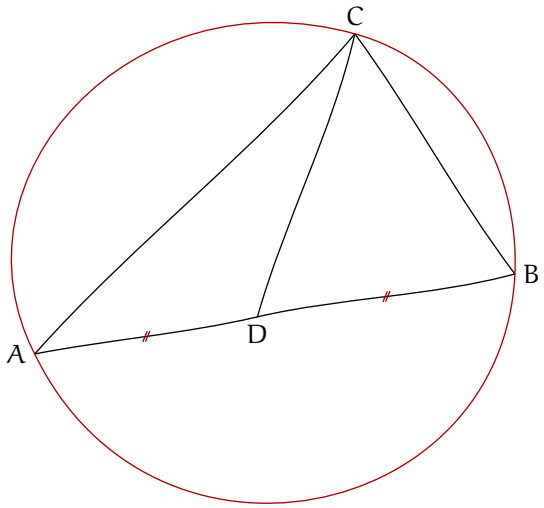
```
randomseed := uniformdeviate infinity;
```

Now METAPOST writes the (random) value used in the log for you to copy. Note that if you are using `luamplib` you need to add the `\mplibshowlog{enable}` option to get this value in the log.

8.4 Drawing freehand

This idea is shamelessly stolen from the wonderful collection of METAPOST examples available at `http://melusine.eu.org/syracuse/metapost/`. But since the examples there are all in French (including all the names of the custom macros), perhaps it would be better to say 'translated' rather than 'stolen'; moreover my implementations are easier to use with plain METAPOST.

8.4.1 Making curves and straight lines look hand drawn

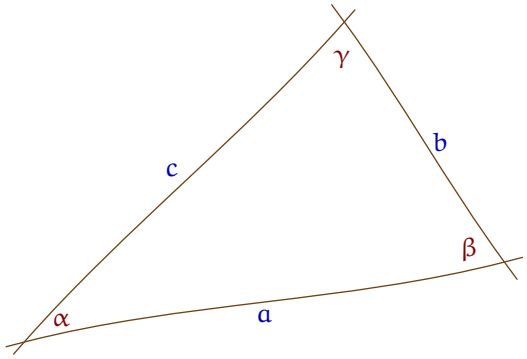


A small amount of random wiggle makes the drawing come out charmingly wonky. Notice that the `freehand_path` macro will transform a path whether it is straight or curved, and open or closed. Notice also that to find D the mid-point of a AB, you need to find the point along the freehand path; if you simply put `1/2[A,B]` there's no guarantee that the point would actually be on the free hand path between A and B. In this case a little extra randomness has been added, and the two segments AD and DB have been marked with traditional markers to show that they are equal. The `moved_along` macro combines shifted and rotating to make the markers fit the wonky lines properly. The Euler Math font complements the hand-drawn look. You might find that a little of this type of decoration goes a long way.

```
def freehand_segment(expr p) =
  point 0 of p {direction 0 of p rotated (4+normaldeviate)} ..
  point 1 of p {direction 1 of p rotated (4+normaldeviate)}
enddef;
def freehand_path(expr p) =
  freehand_segment(subpath(0,1) of p)
  for i=1 upto length(p)-1:
    & freehand_segment(subpath(i,i+1) of p)
  endfor
  if cycle p: & cycle fi
enddef;
color sepia; sepia = (0.44, 0.26, 0.08);
picture marker; marker = image(for s=-1/2, 1/2:
  draw (left--right) scaled 2 rotated 60 shifted (s,0);
endfor);
def moved_along expr x of p =
  rotated angle direction x of p
  shifted point x of p
enddef;
beginfig(1);
  pair A, B, C, D;
  A = (0,-30); B = (180,0);  C = (120,90);
  D = (1/2 + 1/40 normaldeviate)[A, B];
  path triangle, circumcircle, bisector;
  triangle = freehand_path(A--D--B--C--cycle);
  bisector = freehand_segment(C--D);
  circumcircle = freehand_path(A..B..C..cycle);
  draw triangle;
  draw bisector;
  draw circumcircle withcolor .67 red;
  draw marker moved_along 1/2 of triangle withcolor .67 red;
  draw marker moved_along 3/2 of triangle withcolor .67 red;
  label.lft("$A$", A);  % preamble loads "euler-math" package
  label.rt ("$B$", B);
  label.top("$C$", C);
  label.bot("$D$", D);
endfig;
```

8.4.2 Extending straight lines slightly

This second freehand figure uses a macro to draw a wonky line through two points with a bit of overlap at each end. The overlap size is given using the suffix syntax. The lines are drawn in sepia ink to enhance the hand-drawn look. The angle labels are positioned on invisible arcs between neighbouring wonky lines.



❖ If you prefer to use plain old `mpost`, then the AMS Euler font is still available as `eurm10`, and it is encoded as a subset of the TEX math italic layout — essentially it has all the Greek letters but none of the arrows, nor the musical notation:

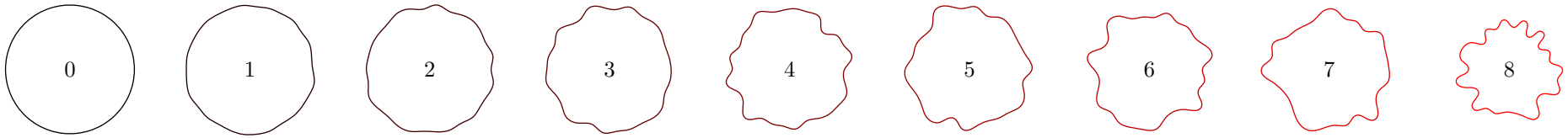
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	Φ	Ψ	Ω	α	β	γ	δ	ϵ
16	ζ	η	θ	ι	κ	λ	μ	ν	ξ	π	ρ	σ	τ	υ	ϕ	χ
32	ψ	ω	ε	ϑ	ϖ			φ								
48	0	1	2	3	4	5	6	7	8	9	.	,	<	/	>	
64	∂	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z					
96	ℓ	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	ı	ȷ	℘		

To make the example on the right work with `mpost`

```
\documentclass{standalone}
\usepackage{luamplib}
\mplibtextextlabel{enable}
\usepackage{euler-math}
\begin{document}
\begin{mplibcode}
 vardef freehand_through@#(expr a, b) =
   save t; pair t;
   t = @# * unitvector(b - a) rotated (4 + normaldeviate);
   a - t .. a {t} .. b {t} .. b + t
 enddef;
 vardef mid_arc@#(expr p, a, b) =
   save c; path c;
   c = fullcircle scaled (2*@#) shifted p cutbefore a cutafter b;
   point arctime 1/2 arclength c of c of c
 enddef;
 beginfig(1);
   pair A, B, C; A = (0,-30); B = (180,0);  C = (120,90);
   path a, b, c;
   a = freehand_through 7 (A, B);
   b = freehand_through 8 (B, C);
   c = freehand_through 6 (C, A);
   color sepia; sepia = (0.44, 0.26, 0.08);
   drawoptions(withcolor sepia);
     draw a; draw b; draw c;
   drawoptions(withcolor .67 blue);
     label.bot ("$a$", point 3/2 of a);
     label.rt  ("$b$", point 3/2 of b);
     label.ulft("$c$", point 3/2 of c);
   drawoptions(withcolor .5 red);
     label("$\alpha$", mid_arc 16 (A, a, c));
     label("$\beta$",  mid_arc 14 (B, b, a));
     label("$\gamma$", mid_arc 14 (C, c, b));
   drawoptions();
 endfig;
\end{mplibcode}
\end{document}
```

8.5 Increasingly random shapes of the same size

If you want a random-looking shape, the general approach is to find a method to make a path that allows you to inject some random noise at each point of the path.



For these shapes the objective was to make them increasingly random, but to keep them all the same length. Each time round the outer loop the *shape* is redeclared to clear it, and then redefined by an inline-loop with n steps like this:

```
shape = for i=1 upto n: (s,0) rotated (360/n*i) .. endfor cycle;
```

except that some random noise is added to the s at each step: when the noise is zero ($r = 0$) you get a circle; as the noise increases the circle is increasingly distorted.

The scaling is done using the `arclength` operator. This works like `length` but instead of telling you the number of points in a path, it returns the actual length as a dimension. Dividing the desired length by this dimension gives the required scaling factor for the random shape just defined. Notice that you have to do this in two steps, and update the shape using `:=`. This is because you need to have defined *shape* before you can refer to it.

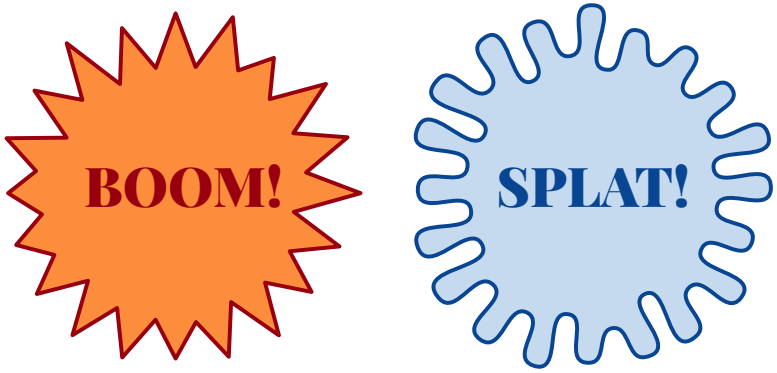
```
beginfig(1);
  numeric desired_length, n, s;
  desired_length = 180; n = 30; s = 80;
  for r=0 upto 8:
    path shape;
    shape = for i=1 upto n:
      (s + r * normaldeviate, 0) rotated (360/n*i) ..
    endfor cycle;

    shape := shape scaled (desired_length/arclength shape);

    draw shape shifted (r*s, 0) withcolor (r/8)[black,red];
    label(decimal r, (r*s, 0));

  endfor
endfig;
```

8.6 Explosions and splashes

Random numbers are also useful to make eye catching banners for posters, presentations, and infographics. Here are two simple example shapes ⟶

```
input colorbrewer-rgb
randomseed:=2128.5073;
beginfig(1);
  n = 40; r = 12 ; s = 72;
  path explosion; explosion = for i=1 upto n:
    (if odd i: - fi r + s + uniformdeviate r) * dir (360/n*i) --
  endfor cycle;
  path splash; splash = for i=1 upto n:
    (if odd i: - fi r + s + uniformdeviate r) * dir (360/n*i) ..
  endfor cycle;
  picture P[];
  P1 = image(
    fill explosion withcolor Oranges 7 4;
    draw explosion withpen pencircle scaled 2 withcolor Reds 7 7;
    label("BOOM!", center explosion) withcolor Reds 7 7;
  );
  P2 = image(
    fill splash withcolor Blues 7 2;
    draw splash withpen pencircle scaled 2 withcolor Blues 7 7;
    label("SPLAT!", center splash) withcolor Blues 7 7;
  );
  label.lft(P1, 10 left); label.rt(P2, 10 right);
```

In this figure n is the number of points in the shape, s is the radius, and r is the amount of randomness added to or removed from s. In order to get a clear zig-zag outline, the loop alternately adds or subtracts r; and then adds a random amount on top to make it look random. Notice that the only difference between the `explosion` and `splash` is that how the connecting lines are constrained to be straight or allowed to make smooth curves.

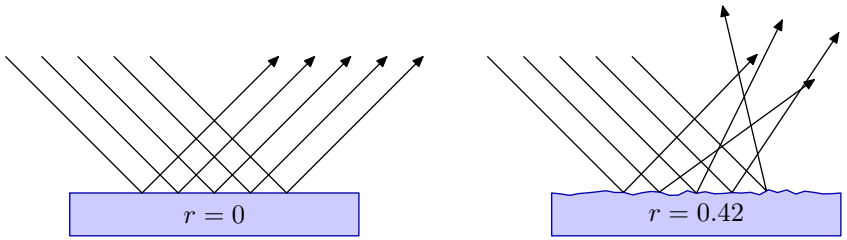


The display font used here is Playfair Display Black. If you have it installed as a system font, you can use `fontspec` and `luamplib` with `lualatex` as described in §3.2, but if you are still using plain `mpost`, then you need to hunt for it in your local `psfonts.map`. If you run METAPOST with the `-recorder` option, it will create a list of all the files used, with the current job name and an extension of `.fls`. This file will include a line which tells you exactly which version of `psfonts.map` is being used. The DVIPS documentation explains the format of the file, but for METAPOST's purposes the first word of each non-comment line defines a font name you can try. When you find it you can add something like this to the top of the example:

```
defaultfont = "PlayfairDisplay-Black-osf-t1--base";
defaultscale = 3;
```

8.7 Simulating jagged edges or rough surfaces

You can use the idea of adding a little bit of noise to simulate a rough surface.



These diagrams are supposed to represent light rays reflecting from a surface: on the left the surface is smooth ($r = 0$) and on the right it's rough ($r = 0.42$). The parameter r is used in the METAPOST program as a scaling factor for the random noise added to each point along the rough surface; the only difference in the code to produce the two figures was the value of r.First the base block is created with some noise on the upper side. Then five rays are created. Applying `ypart` to the pair of times returned by `intersectiontimes` gives us the point of the base where the incident ray hits it. This point and the perpendicular at that point are then used to get the angle for the reflected ray. The diagrams are effective because the rays are reflected at realistic looking angles.

The simple approach to adding noise along a path works well in most cases provided there's not too much noise, but it is always possible that you'll get two consecutive values at opposite extremes that will show up as an obtrusive jag in your line. To fix this you can simply run your program again to use a different random seed value; or you could try using `..` instead of `--` to connect each point, but beware that sometimes this can create unexpected loops.

```
def perpendicular expr t of p =
  direction t of p rotated 90 shifted point t of p
enddef;

vardef block(expr wd, dp, theta, r) =
  save base, ray, b;
  path base, ray[]; numeric b;

  base = origin for i=1 upto 31:
    -- (i/32 * wd, r * normaldeviate)
  endfor -- (wd,0) -- (wd,-dp) -- (0,-dp) -- cycle;

  image(
    fill base withcolor .8[blue,white];
    draw base withcolor .67 blue;

    for i = 2 upto 6:
      ray[i] = (left--right) scaled 2/3 wd rotated -theta
                             shifted (i/8 * wd,0);
      b := ypart (ray[i] intersectiontimes base);
      ray[i] := point 0 of ray[i] -- point b of base -- point 0 of ray[i]
                reflectedabout(point b of base, perpendicular b of base);
      drawarrow ray[i];
    endfor

    label("$r=" & decimal r & "$", center base);
  )
enddef;
beginfig(0);
  draw block(108, 16, 45, 0);
  draw block(108, 16, 45, 0.42) shifted 180 right;
endfig;
```

8.7.1 Walking along a torn edge

It's also possible to use a random walk approach so that each random step takes account of the previous one to avoid any big jumps. Here's one way to do that.

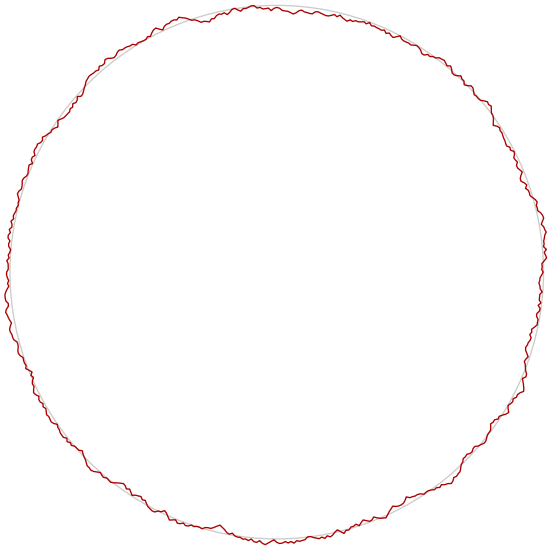
```
beginfig(1);
  path t; numeric x, y;
  x = 0; y=0;
  t = (x, -20) -- (x, y) for i=1 upto 288:
      -- (incr x, walkr y) endfor -- (x, -20) -- cycle;
  fill t withcolor (1, 1, 31/32); draw t withcolor .67 blue;
endfig;
```

The `walkr` routine works like the `incr` and `decr` commands; it updates the value of the argument. The idea is that the further away from zero you are, the more likely is that the next value will take you back towards zero.

```
vardef walkr suffix $ =
  save t; boolean t;
  t = uniformdeviate 1 < (2 ** - abs($));
  $ := $ if t: + else: - fi signr $; $
enddef;
vardef signr suffix $ =
  if $ < 0: - else: + fi uniformdeviate 1
enddef;
```

You can use this to produce more realistic torn edges. You can also apply this as a form of jitter to a curved path, by adding a suitably rotated vector to enough points along the path, as shown in the example on the right.——————————→

```
beginfig(1);
  path c; c = fullcircle scaled 200;
  draw c withcolor .8 white;
  y=0; n = 600;
  path t; t = for i=0 upto n-1:
    point i/n*length(c) of c
    + (0, walkr y) rotated angle direction i/n*length(c) of c
    --
  endfor cycle;
  draw t withcolor .67 red;
endfig;
```



9 Plane geometry

This section deals with drawing geometrical figures that involve lines, angles, polygons, and circles. Plain METAPOST provides very few tools that are explicitly designed to help draw geometric figures, but it is usually possible to find an elegant construction using these tools and the relevant primitive commands. It is tempting to build up your own library of special purpose macros, but experience suggests that it is often better to adapt a general technique to the task in hand, and to create a specific solution to your current problem. One of the main issues is catching exceptions; since it is hard to write completely general macros, I have tried simply to present each technique so that you can understand it and adapt it as required.

The classical constructions from Euclid's *Elements* are often useful sources of inspiration for macros, but they do not always point in the right direction. For example consider the first proposition: *given two points find a third point, so that the three points make an equilateral triangle.* Euclid's construction is to draw an arc, with radius equal to the length of the segment between the two points, at each point and find the intersection. This might lead us to a function like this:

```
vardef equilateral_triangle_point(expr a, b) =
  save c; path c; c = fullcircle scaled 2 abs(b-a);
  (c shifted a intersectionpoint c shifted b)
enddef;
```

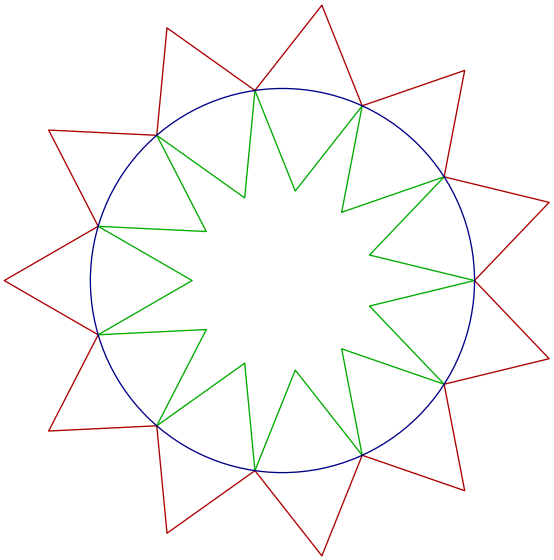
This works but has a couple of issues. First using `intersectionpoint` feels a bit like cheating; secondly, and more seriously, the point returned depends on the orientation of the points `a` and `b`. In some configurations the first intersection found will be on the left, in others on the right. We could fix this by rotating the circle `c` by `angle (b-a)`, but we can do better with a simple rotation of the second point about the first:

```
vardef equilateral_triangle_point(expr a, b) =
  b rotatedabout(a,60)
enddef;
```

And if you want to get right back to primitives you could even write that as:

```
vardef equilateral_triangle_point(expr a, b) =
  b shifted -a rotated 60 shifted a
enddef;
```

Here is the equilateral triangle point macro in action.



```
beginfig(1);
  path c; c = fullcircle scaled 144;
  numeric n; n = 11;
  for i=0 upto n-1:
    pair a,b,p,q;
    a = point 8/n * i of c;
    b = point 8/n * (i + 1) of c;
    p = equilateral_triangle_point(a,b);
    q = equilateral_triangle_point(b,a);
    draw a -- p -- b withcolor .67 green;
    draw a -- q -- b withcolor .67 red;
  endfor
  draw c withcolor .53 blue;
endfig;
```

9.1 Bisecting lines and paths

THE BEST WAY TO BISECT a line depends on how you have defined it. If you have two ⟨pair⟩s a and b, then the simplest way to find the ⟨pair⟩ that bisects them is to write `1/2[a,b]`. This mediation mechanism is entirely general, so you can write `1/3[a,b]`, `1/4[a,b]`, and so on to define other pairs that are part of the way from a to b. But the number before the left bracket does not have to be confined to the range $(0, 1)$. If you write `3/2[a,b]` you will get a pair on the extension of the line from a to b beyond b. To get a pair going the other way you can either reverse a and b, or use a negative number; but don't get caught out by the METAPOST precedence rules: `-1/2[a,b]` is interpreted as `-(1/2[a,b])` and not as `(-1/2)[a,b]`, so either put in the parentheses or swap the order of the pairs: `(3/2)[b,a]`. See ⟶

If you want to work with a ⟨path⟩ variable, rather than separate ⟨pair⟩ variables, you can use the `point t of p` notation to do mediation along the path. For a simple straight path p of length 1 then `point 1/2 of p` will give you the midpoint. More generally, `point 1/2 length p of p` will give you the midpoint of a path of any length. This works fine for simple paths, along which METAPOST's time moves evenly, but for more complicated, curved paths you have to use this rather cumbersome notation:

```
point arctime 1/2 arclength p of p of p
```

If your path makes a regular polygon, then you can bisect the shape with the line

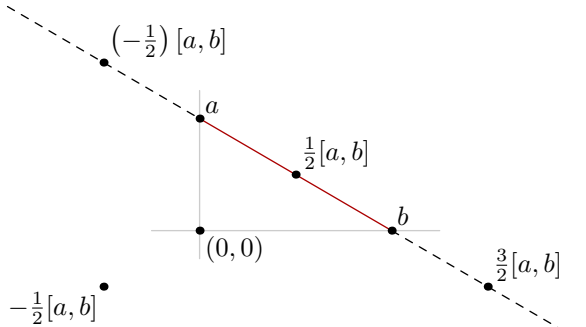
```
point t of p -- point t + 1/2 length p of p
```

☞ If the polygon has an odd number of sides, then $2t$ must be a whole number.

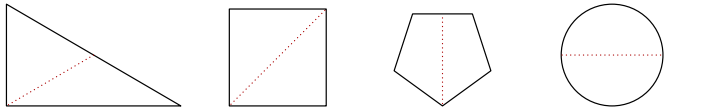
IN A TRIANGLE these bisecting lines are called medians. The three medians intersect at the centroid of the triangle. The centroid is a good place to put a label on a triangle. You could find it using the macro from 4.3, or with a construction using `whatever` on any two medians, but since we know that the centroid divides each median in the ratio $2:1$ we can find the centroid of a triangle path p most simply with:

```
z0 = 2/3[point 0 of p, point 3/2 of p];
```

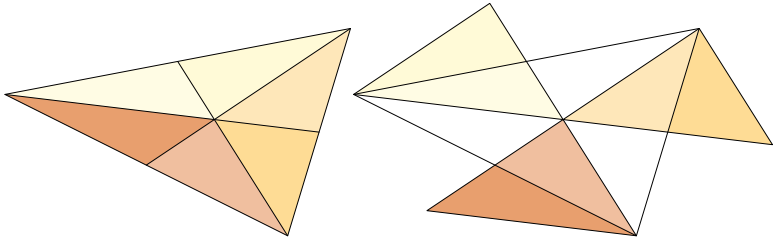
The median lines are the basis for several beautiful theorems about the geometry of the triangle. The theorem shown here was first published in 2014.



Probably not what was intended...



Dotted lines drawn with: `point 0 of p -- point 1/2 length p of p`



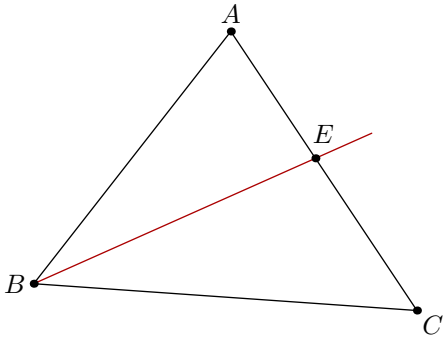
Lee Sallows' theorem of median triangles

9.2 Bisecting angles

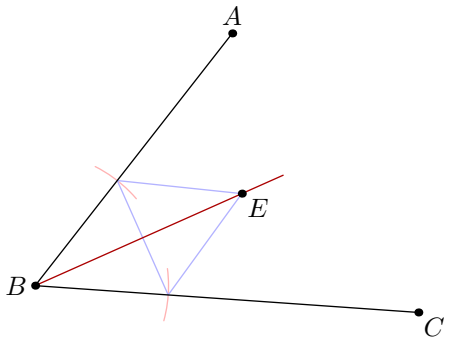
In an equilateral triangle the medians also bisect the angles at each vertex; this is the basis of Euclid's method of bisecting an angle set out in the Second Proposition. You can do the same in METAPOST, but it might not always be the best way. Whatever approach you take, an angle is defined by three points; one that defines the corner and two that define the lines extending from that corner. In this exploration I've used a, b, and c to represent the points, with b being the one in the middle, and at the corner.

Euclid's method is to draw an arc centred at the corner, and then construct an equilateral triangle on the two points where the arc crosses the lines. This is shown on the right, with a macro that re-uses the equilateral triangle point macro given above. But if your aim were to find any point on the line bisecting $\angle ABC$, then you could simplify this and make it more efficient by using $e = \frac{1}{2}[p, q]$ instead of calling the triangle macro at all. However the macro is still making two calls to `intersectionpoint`. If you wanted to eliminate this you could use the useful plain METAPOST macro `unitvector` to produce a solution based on adding two equal length vectors from the corner to the two other points. Another approach is to exploit another geometric theorem that states that the bisector of an angle in a triangle divides the opposite side in the ratio of the two other sides. So if sides AB and BC have lengths p and q then the bisector will be $\frac{p}{p+q} = \frac{1}{1+q/p}$ from A to C, and you can express this simply using METAPOST's mediation syntax:

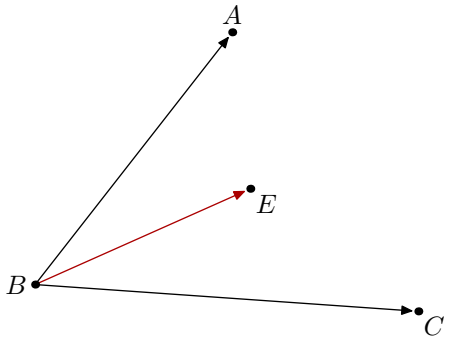
```
vardef interior_bisector(expr a,b,c) =
  (1/(1+abs(c-b)/abs(a-b)))[a,c]
enddef;
```




```
vardef euclidean_bisector(expr a,b,c,r) =
  save arc,p,q,e;
  path arc; pair p,q,e;
  arc = fullcircle scaled r shifted b;
  p = (a--b) intersectionpoint arc;
  q = (b--c) intersectionpoint arc ;
  e = equilateral_triangle_point(p,q);
  e
enddef;
```



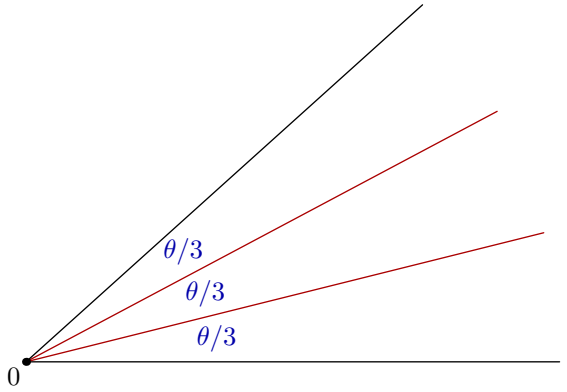

```
vardef vector_bisector(expr a,b,c,r) =
   b + unitvector (a-b) scaled r
     + unitvector (c-b) scaled r
enddef;
```



9.3 Trisections and general sections of angles

There is no classical method to trisect an arbitrary angle, so you need to resort to measuring and arithmetic in METAPOST. If the angle is a given this is trivial:

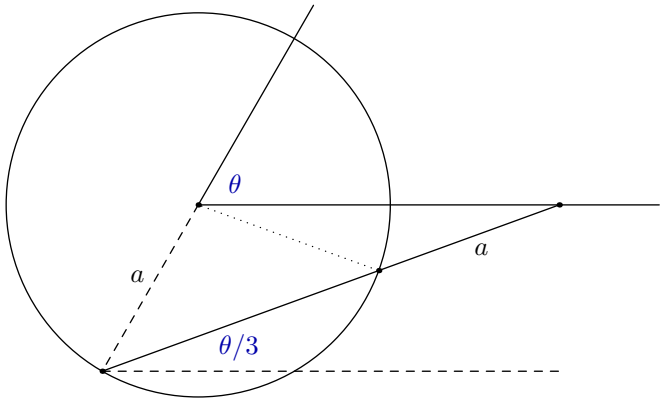
```
beginfig(1);
  path ray; ray = origin -- 200 right;
  numeric theta; theta = 42;
  draw ray;
  draw ray rotated 1/3 theta withcolor 2/3 red;
  draw ray rotated 2/3 theta withcolor 2/3 red;
  draw ray rotated theta;
  dotlabel.llft("$0$", origin);
  label("$\theta/3$", 72 right rotated 1/6 theta) withcolor 2/3 blue;
  label("$\theta/3$", 72 right rotated 3/6 theta) withcolor 2/3 blue;
  label("$\theta/3$", 72 right rotated 5/6 theta) withcolor 2/3 blue;
endfig;
```



But if you have only the coordinates of some points then you need to use the `angle` primitive to measure the angle first; `angle` takes a ⟨pair⟩ argument and returns a numeric representing the angle in degrees measured clockwise from the x-axis to a line through the origin and the point represented by the pair. This definition means that if you have three points A, B, and C, then you can measure $\angle ABC$ with `angle(C-B)-angle(A-B)`. Following the usual convention this gives you the angle at B; if you list the points in clockwise order you will get a positive result. If you don't care about the order, you can make this into a more robust macro:

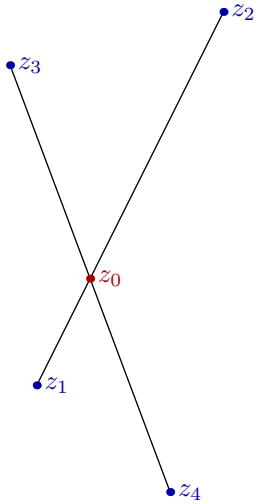
```
vardef measured_angle(expr P, Q, R) =
  (angle (P-Q) - angle (R-Q)) * turningnumber (P--Q--R--cycle) mod 360
enddef;
```

The primitive `turningnumber` is explained on p. 111 of *The METAFONTbook*. It takes a closed path and returns number of times that you would turn through 360° if you traversed the path. We use this here to negate the measured angle if necessary, so that you always get the interior angle. The `mod 360` on the end ensures that the result is in the range $0 \le \theta < 360$. Armed with a measured angle, all you then need is arithmetic. It might be possible to use the *solve* macro to simulate the Neusis construction (that allows you to measure a length) illustrated on the right, but measuring the angles is rather easier.



9.4 Intersections

If you have line segments defined by their endpoints, then the canonical way to find their intersection, is to use the mediation syntax with *whatever* twice:




```
beginfig(1);
  z1 = (10, 50);
  z2 = (80, 190);
  z3 = (0, 170);
  z4 = (60, 10);

  z0 = whatever [z1, z2] = whatever [z3, z4];

  draw z1--z2; draw z3--z4;

  forsuffixes @=0,1,2,3,4:
    dotlabel.rt(TEX("$z_" & decimal @ & "$"), z@)
      withcolor 2/3 if @=0: red else: blue fi;
  endfor
endfig;
```

The mediation syntax works even if the intersection point does not actually lie on either of the two line segments. The intersection will be the point where the two (infinite) lines through the pairs of points meet. If the two lines are parallel, you'll get an 'inconsistent equation' error. If you want to capture the calculated values, then use undefined numeric variables instead of *whatever*:

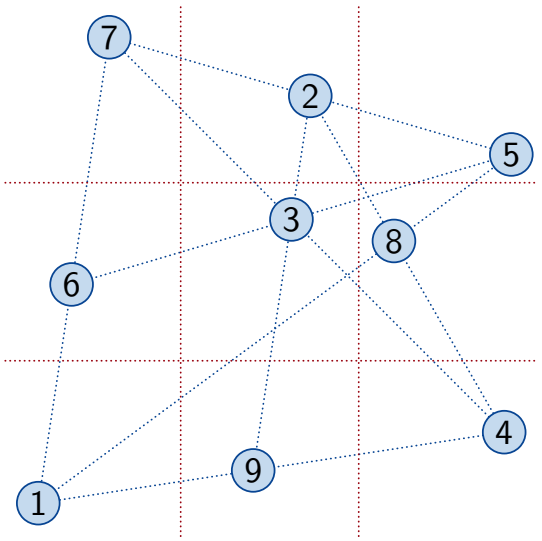
```
z0 = alpha [z1, z2] = beta [z3, z4];
```

In this example you would find $\alpha = 0.286$ and $\beta = 0.5$. If you are trying to find where the line through your points intersects a horizontal or vertical, then you only need one mediation and a simple equation for the relevant x or y coordinate:

```
z0 = alpha [z1, z2]; x0 = 0;  % for example
```

If you have defined your lines as paths, and especially if they are more complicated than straight lines, you need to use the `intersectiontimes` primitive or the `intersectionpoint` macro, as explained on pp.136–137 of *The METAFONTbook*.

A puzzle square featuring some intersections



The points were defined like this (the order was important).

```
z1 = (10,10);
z4 = 144 right rotated 12;
z5 = z4 shifted (2, 78);
z7 = z4 reflectedabout(origin, (1,1));

z2 = 1/2 [z5, z7];
z9 = whatever [z1, z4];
z2-z9 = whatever * (z7-z1);
z8 = whatever [z1, z5] = whatever [z2, z4];
z3 = whatever [z2, z9] = whatever [z4, z7];
z6 = whatever [z1, z7] = whatever [z3, z5];
```

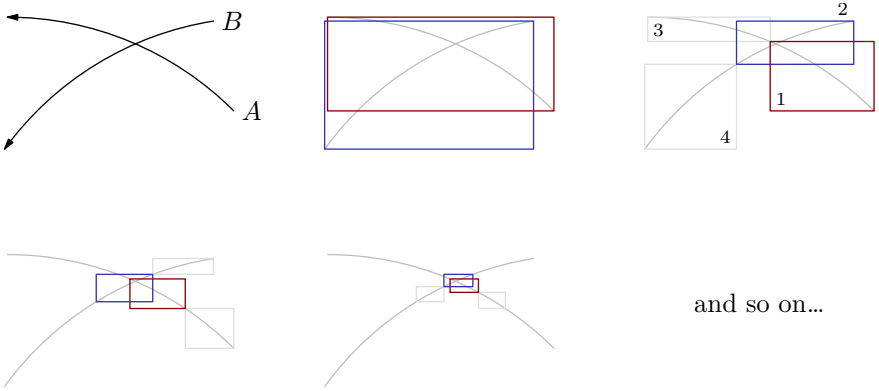
9.4.1 The intersection algorithm

METAPOST inherits a fast algorithm for finding the intersection between two paths from METAFONT. It is explained rather more succinctly than usual at the end of Chapter 14 of *The METAFONTbook*, with more detail given in the web source for METAFONT. The core algorithm works on paths of length 1. If you have longer paths, METAPOST works its way along the paths applying the core algorithm to successive pairs of unit subpaths. It does this is lexicographic order; this means that, if you have two circles A and B, and you do this:

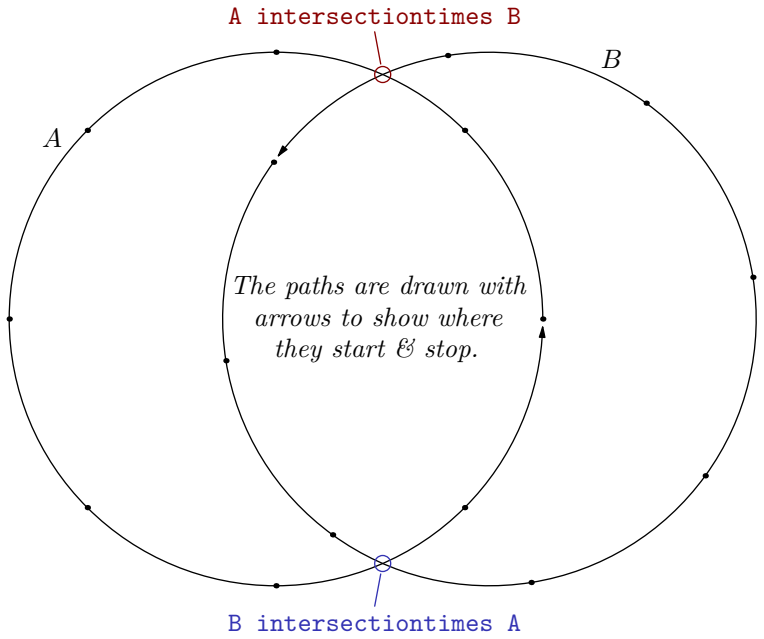
```
(t, u) = A intersectiontimes B;
```

then METAPOST will first look for an intersection between subpath $(0,1)$ of A and subpath $(0,1)$ of B, then subpath $(0,1)$ of A and subpath $(1,2)$ of B, and so on, with B varying faster, until you get to subpath $(7,8)$ of A and subpath $(7,8)$ of B. But you may never get that far, as the process stops as soon as the first intersection is found. The upshot of this is that the intersection point found will always be as early as possible on A. Note that after the call above point t of A will be very close to point u of B, as they both refer to the same intersection point. If you want the alternative point that is earlier on B, then use 'B intersectiontimes A' instead.

When we get down to paths of length 1, the algorithm works something like this:



and so on...



The two paths are represented as rectangles that enclose the end points and the control points for each path. If these rectangles don't overlap then there is certainly no intersection. Otherwise METAPOST bisects each path and considers four smaller rectangles, in the order $(1,2)$, $(1,4)$, $(3,2)$, $(3,4)$ (as shown). In this case it will pick $(1,2)$, discard 4, and push 3 onto a stack. It carries on doing this, back tracking as required, until it finds sufficiently small overlapping rectangles. The two times returned by `intersectiontimes` are the midpoints of the subpaths enclosed by these two tiny rectangles, which is why they do not always refer to exactly the same point.

9.4.2 Finding all intersection points

As noted above, the intersectiontimes algorithm will stop at the first intersection of the two paths, but it is possible that the two paths will intersect again further along. If you want to find all the intersection points then the simplest technique is just to unwrap the algorithm slightly, and loop through all the unit subpaths applying intersectiontimes to each pair. Using an array to hold the points and a counter, you can get them with something like this:

```
pair P[], times; numeric n; n = 0;
for i = 1 upto length(A):
  for j = 1 upto length(B):
    times := subpath (i-1,i) of A intersectiontimes subpath (j-1,j) of B;
    if xpart times > -1:
      P[incr n] = 1/2[point xpart times of subpath (i-1,i) of A,
                      point ypart times of subpath (j-1,j) of B];
    fi
  endfor
endfor
```

and then use them like this:

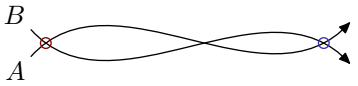
```
for i=1 upto n:
    draw fullcircle scaled 4 shifted P[i]; % or whatever
endfor
```

There are a couple of METAPOST technical points to note. The intersectiontimes operation returns a pair, which we assign to a pair variable *times* above; we have to use := to re-assign it in each loop, and we have to use an explicit pair variable because you can't assign to a literal pair; METAPOST will give you an error if you try (t, u) := A intersectiontimes B;. This may come as a surprise, because you *can* legally do (t, u) = A intersectiontimes B, but in a loop this causes an inconsistent equation error on the second iteration. If you need to avoid the repeated use of xpart and ypart, one alternative is to do this inside the loop:

```
...
numeric t, u;
(t, u) = A intersectiontimes B;
...
```

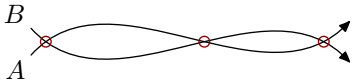
Now the numerics are reset each time and the equation is not inconsistent.

The technique discussed on the left, works well on paths where the points on one or both of the paths are close together, so that the unit subpaths are short; But it is possible to create quite long paths of unit length and these may intersect each other more than once, like so:



Here the two paths A and B are Bézier splines of with length=1, so the normal METAPOST algorithm is only ever going to give you one of the intersections. In the diagram above, the red circle marks the point given by A intersectiontimes B. We can try reversing the first path, and in this case you get the point marked in blue, but what about the one in the middle?

The most reliable approach is to take a copy of one of the paths, and snip it off at the intersection and try again until there is nothing left to snip.



The three points marked here were captured like this:

```
pair P[]; numeric n; n=0;
path R; R := A; %take a copy of A
forever:
  R := R cutbefore B;  % snip where we cross B
  exitif length cuttings = 0;  % stop if nothing was cut
  P[incr n] = point 0 of R; % capture the point
  R := subpath (epsilon, infinity) of R; % nudge along
endfor
```

This technique also works on paths with length greater than one, so you may prefer it as your general "get all the intersections" approach. Note that the cutbefore macro is defined using intersectiontimes.

9.5 Parallel and orthogonal or whatever

Given five known points — A, B, C, D, and E — METAPOST can find the point F on the line $A\,..\,B$, so that $E\,..\,F$ is parallel to $C\,..\,D$ like this:

```
F = whatever[A, B]; % F is on the line A..B
E-F = whatever * (C-D) % E..F || C..D
```

In the second line the expressions `E-F` and `C-D` return ⟨pair⟩ variables and the equation with `whatever` says that they must be scalar multiples of each other. With the first equation, this is enough for METAPOST to work out where F should go. Note that `whatever` can take any real value, positive or negative, so it does not matter whether you put `E-F` or `F-E`. Note also that for the same reason, while F will lie on the *line* through A and B, it might not lie on the *segment* from A to B. But also note that you *cannot* write the second equation as

```
E-F * whatever = (C-D); % <--- gives an error
```

because you can only apply `whatever` to known quantities.

❖ To define a line perpendicular to $C\,..\,D$ rather than parallel, then you can write:

```
G = whatever[A, B];
E-G = whatever * (C-D) rotated 90;
```

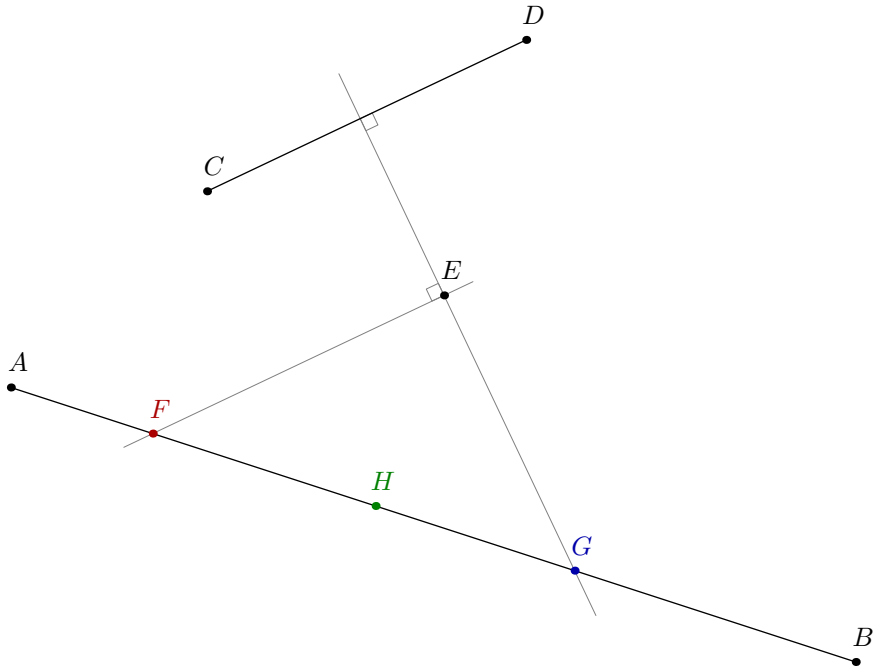
and obviously the `90` can be adjusted to whatever angle you please, if you want something between parallel and orthogonal.

❖ To define the line through E that is perpendicular to $A\,..\,B$, you should just use $(A - B)$ instead of $(C - D)$. The diagram shows H, the point on $A\,..\,B$ that is closest to E; you can (I trust) work out how to define that yourself.

❖ If you just need to compute the perpendicular distance from the point E to a line $A\,..\,B$, rather than defining the point H, then you can use Knuth's 'slick' formula:

```
abs ypart ((E-A) rotated -angle (B-A))
```

This effectively rotates E about A by the angle of the line, so that the problem is reduced to measuring the height of a point above the x-axis, which is what `ypart` does, of course.



There are some limitations to what you can do with METAPOST's linear equations; for one thing you can't generally say things like `length(C-A) = 72`. If you want to find the two points on a line that are a given distance from an external point, it's often simpler to find the intersection points of the line with a suitably scaled and shifted circle, even if you don't actually then draw the circle. You can usually find the other point by reversing the circle.

9.6 Drawing circles

The canonical way to draw a circle in plain METAPOST is to use the pre-defined path `fullcircle` with a suitable transformation. The path is defined (in `plain.mp`) using two METAPOST primitive commands:

```
path fullcircle; fullcircle = makepath pencircle;
```

A **pencircle** is the basic nib that is used to draw lines that digitize neatly; it represents a true circle of diameter 1, passing through the points $(\pm.5, 0)$ and $(0, \pm.5)$. When processed with **makepath** it turns into a closed polygonal path with eight points that closely approximates a circle with diameter 1 bp centred on the point $(0, 0)$. To use it, you can scale it and shift it. To draw a circle with radius 2 cm at the point $(34, 21)$ you would do:

```
draw fullcircle scaled 4cm shifted (34, 21);
```

Remember to scale before you shift, and that *fullcircle* has unit *diameter*, not unit *radius*. To draw a circle centred at point A that passes through point B [I] try:

```
draw fullcircle scaled 2 abs (B-A) shifted A;
```

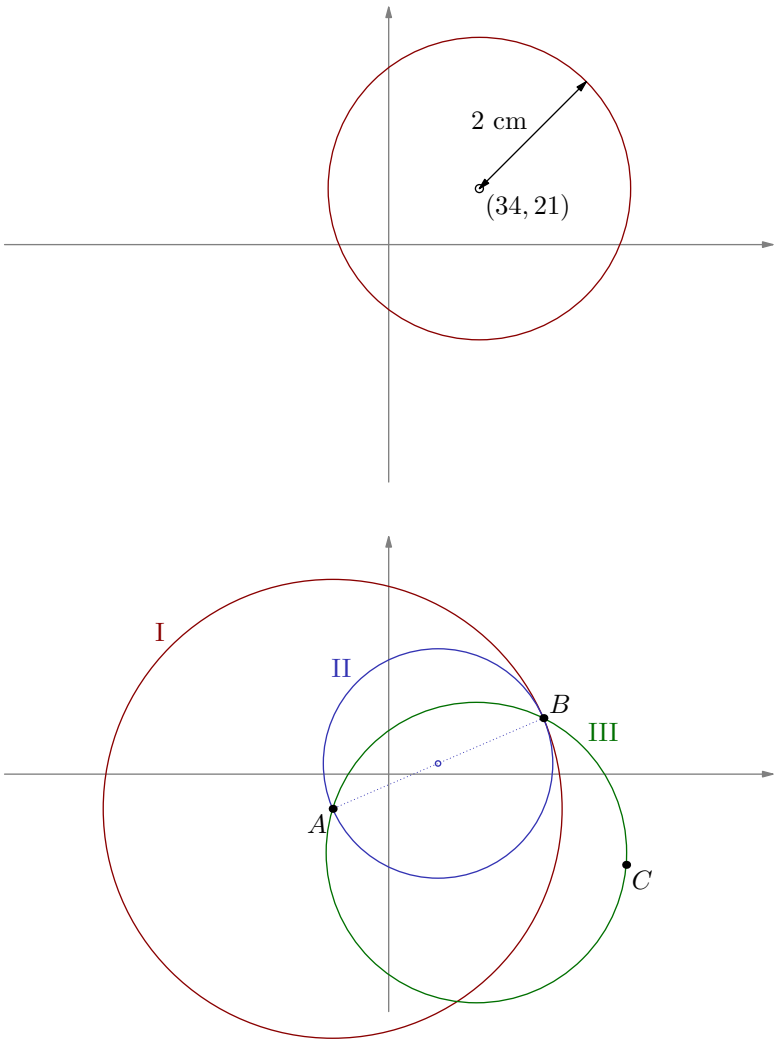
There are of course an infinite number of circles that you can draw through two points, but if the line between the two points is a diameter [II], then you can do:

```
draw fullcircle scaled abs (B-A) shifted 1/2[A,B];
```

Finally three points define a unique circle [III]:

```
vardef circle_through(expr A, B, C) =
    save o; pair o;
    o = whatever * (A-B) rotated 90 shifted 1/2[A,B]
      = whatever * (B-C) rotated 90 shifted 1/2[B,C];
    fullcircle scaled 2 abs (A-o) shifted o
enddef;
```

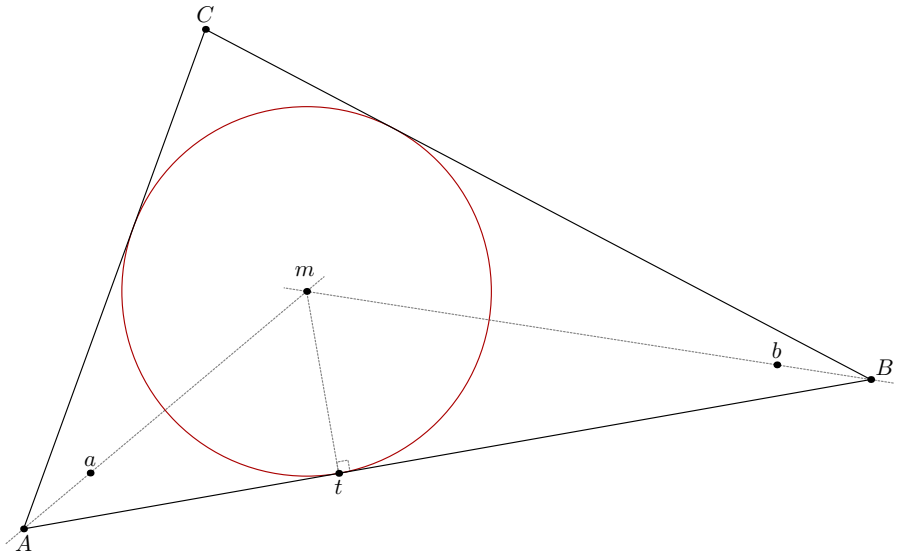
Plain METAPOST also defines *halfcircle* and *quartercircle*, as the appropriate subpaths of *fullcircle*, both starting at point 0 (3 o'clock). Curiously, this differs from METAFONT where *quartercircle* is defined first, and the other two composed from it. The reference point of these two paths is the center of the complete circle of which they would be part; so if you did "**draw** *quartercircle* **shifted** $(34, 21)$;", you would get an quarter-circle arc from $(34.5, 21)$ to $(34, 21.5)$.



9.7 Incircle and excircles of a triangle

The incircle of a triangle is the largest circle contained in the triangle. The centre of the incircle lies at the intersection of the internal angle bisectors. So we can use ideas from §9.2 and §9.4 to define a macro that returns the required path given points A, B, and C:

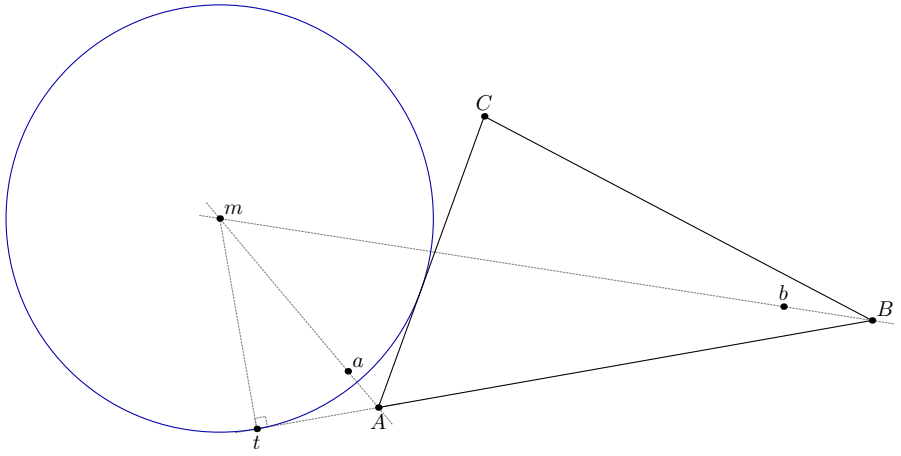
```
vardef incircle(expr A, B, C) =
  save a, b, m, t; pair a, b, m, t;
  a = A + unitvector (C-A) + unitvector (B-A);
  b = B + unitvector (A-B) + unitvector (C-B);
  m = whatever[A,a] = whatever [B,b]; t = whatever[A,B];
  t-m = whatever * (B-A) rotated 90;
  fullcircle scaled 2 abs (t-m) shifted m
enddef;
```



The excircles of a triangle are the three circles lying outside the triangle and tangent to one edge and the extensions of the other two. The centres of each excircle lie at the intersection of one internal angle bisector and the external angle bisector of one of the other corners. To get the external angle bisector, all you have to do is reverse the direction of one of the `unitvector` calls (can you see why?)

```
vardef excircle(expr A, B, C) =
  save a, b, m, t; pair a, b, m, t;
  a = A + unitvector (C-A) - unitvector (B-A);
  b = B + unitvector (A-B) + unitvector (C-B);
  m = whatever[A,a] = whatever [B,b]; t = whatever[A,B];
  t-m = whatever * (B-A) rotated 90;
  fullcircle scaled 2 abs (t-m) shifted m
enddef;
```

☞ To get the other excircles, call the macro with the points in a different order.



9.8 Circumcircle of a triangle

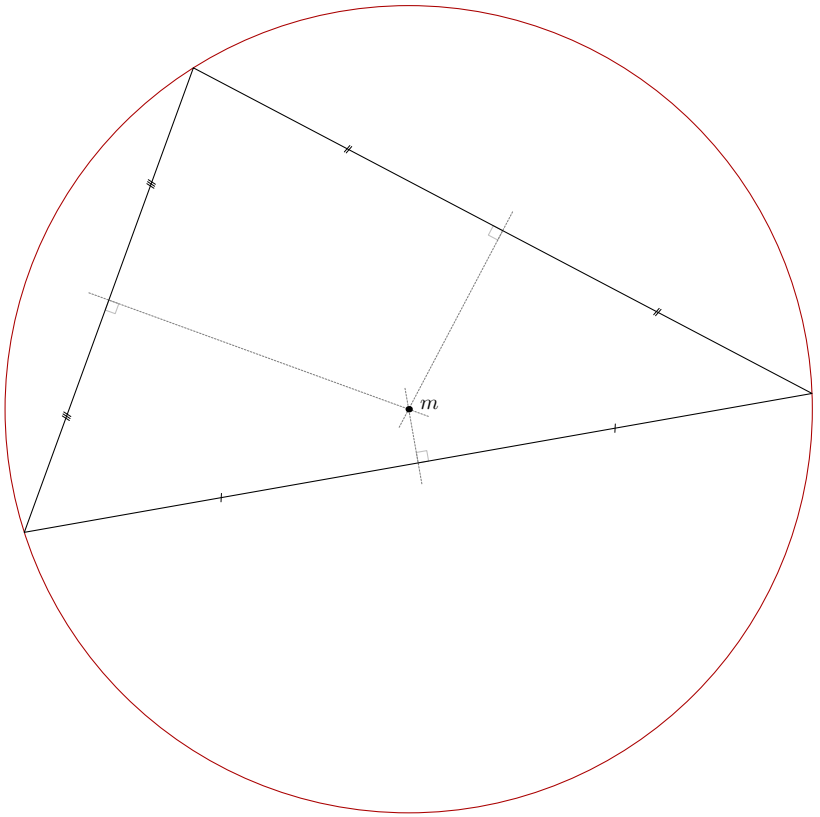
The circumcircle of a triangle is the circle through the three corners, so if you already have the corners of your triangle as separate ⟨pair⟩ variables, you can use the `circle_through` macro from §9.6. Or you can adapt the macro to take a single triangular path:

```
vardef circumcircle(expr T) =
  save m; pair m;
  m = whatever * (point 0 of T - point 1 of T) rotated 90 shifted point 1/2 of T
    = whatever * (point 1 of T - point 2 of T) rotated 90 shifted point 3/2 of T;
  fullcircle scaled 2 abs (point 0 of T - m) shifted m
enddef;
```

Note that as the diagram on the right shows, the centre of the circumcircle is the intersection of all three of the perpendicular bisectors of sides, but for the purposes of drawing in METAPOST you only need to find the intersection of two of them. You could write

```
  m = whatever * (point 0 of T - point 1 of T) rotated 90 shifted point 1/2 of T
    = whatever * (point 1 of T - point 2 of T) rotated 90 shifted point 3/2 of T
    = whatever * (point 2 of T - point 3 of T) rotated 90 shifted point 5/2 of T;
```

but this does not add any more information to the equation for m, and METAPOST will sometimes give you an "inconsistent equation" error if your triangle is long and thin.



☞ The marks that show line segments are equal were created by this macro.

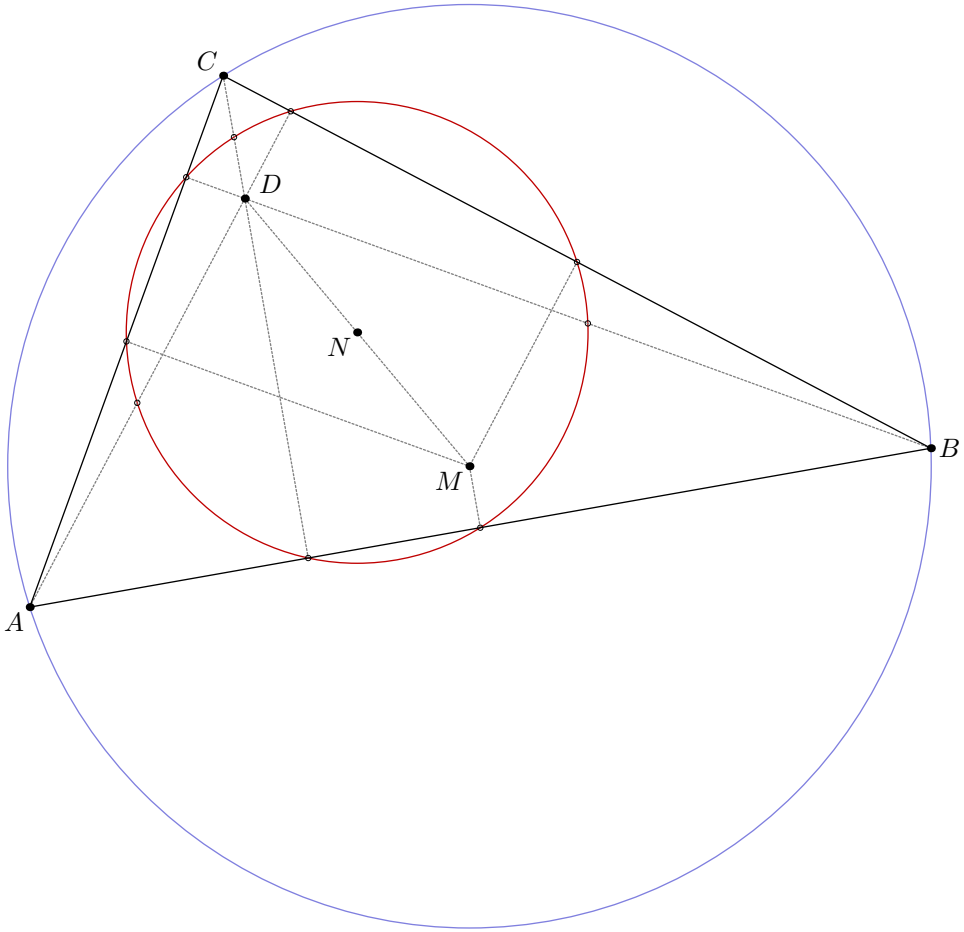
```
vardef mark_equal(expr a, b, n) =
  save m, s; picture m; m = image(
    numeric s; 2s = n - 1; for t=-s upto s:
      draw (down--up) scaled 2 rotated -13 shifted (t,0)
        withpen pencircle scaled 1/4;
    endfor
  );
  draw m rotated angle (b-a) shifted 1/4[a,b];
  draw m rotated angle (b-a) shifted 3/4[a,b];
enddef;
```

Given the triangular ⟨path⟩ T, the macro was used like this: ⟶

```
mark_equal(point 0 of T, point 1 of T, 1);
mark_equal(point 1 of T, point 2 of T, 2);
mark_equal(point 2 of T, point 0 of T, 3);
```

9.9 The nine-point circle of a triangle

The orthocentre of a triangle is the point at the intersection of the three altitudes, shown as point D below. The point N, half-way from D to the circumcentre M is the centre of the remarkable nine-point circle which passes through the bases of the three altitudes and bisects the six line segments AB, AC, AD, BC, BD, and CD.




```
beginfig(1);
  pair A, B, C, D, N, M, p, q, r;
  A = origin; B = 343 dir 10; C = 212 dir 70;
  % pedal points (not labelled)
  p = whatever[B, C]; A - p = whatever * (B-C) rotated 90;
  q = whatever[C, A]; B - q = whatever * (C-A) rotated 90;
  r = whatever[A, B]; C - r = whatever * (A-B) rotated 90;

  D = whatever[A, p] = whatever[B, q];
  N = 1/4(A + B + C + D);  % remarkably...
  M = D rotatedabout(N, 180);  % M is also the circumcentre

  path circumcircle, nine_point_circle;
  nine_point_circle = fullcircle scaled 2 abs(N - 1/2[A, B]) shifted N;
  circumcircle = fullcircle scaled 2 abs(M - A) shifted M;

  draw nine_point_circle withcolor 3/4 red;
  draw circumcircle withcolor 1/2[3/4 blue, white];

  drawoptions(dashed evenly scaled 1/4 withcolor 1/2);
  draw 1/2[A,B] -- M -- 1/2[B, C];
  draw 1/2[C,A] -- M -- D;
  draw A -- p;
  draw B -- q;
  draw C -- r;

  % mark the nine points with small circles
  drawoptions(withpen pencircle scaled 1/4);
  draw fullcircle scaled 2 shifted 1/2[A, B];
  draw fullcircle scaled 2 shifted 1/2[A, C];
  draw fullcircle scaled 2 shifted 1/2[A, D];
  draw fullcircle scaled 2 shifted 1/2[B, C];
  draw fullcircle scaled 2 shifted 1/2[B, D];
  draw fullcircle scaled 2 shifted 1/2[C, D];
  draw fullcircle scaled 2 shifted p;
  draw fullcircle scaled 2 shifted q;
  draw fullcircle scaled 2 shifted r;

  drawoptions();
  draw A--B--C--cycle;
  dotlabel.llft("$A$", A);
  dotlabel.rt("$B$", B);
  dotlabel.ulft("$C$", C);
  dotlabel.urt("\ $D$", D);
  dotlabel.llft("$M$", M);
  dotlabel.llft("$N$", N);
endfig;
```

9.10 Lines tangent to a point on a path

METAPOST represents paths internally as a sequence of nodes. Each node consists of three pairs: the pre-control point, the point itself, and the post-control point. For a given path p you can extract these points at time t with these operators:

```
precontrol t of p
point t of p
postcontrol t of p
```

Unless you explicitly set them differently, METAPOST's curve fitting will make these three points co-linear, so you can draw a tangent at point t with

```
draw precontrol t of p -- postcontrol t of p;
```

The length of the tangent line drawn like this depends on the size and shape of the curve, but it is somewhat arbitrary. So you may prefer to extract a ⟨pair⟩ representing the tangent at point t with

```
pair d; d = postcontrol t of p - precontrol t of p;
```

In fact, this is so useful that `plain.mp` provides `direction` as a shorthand:

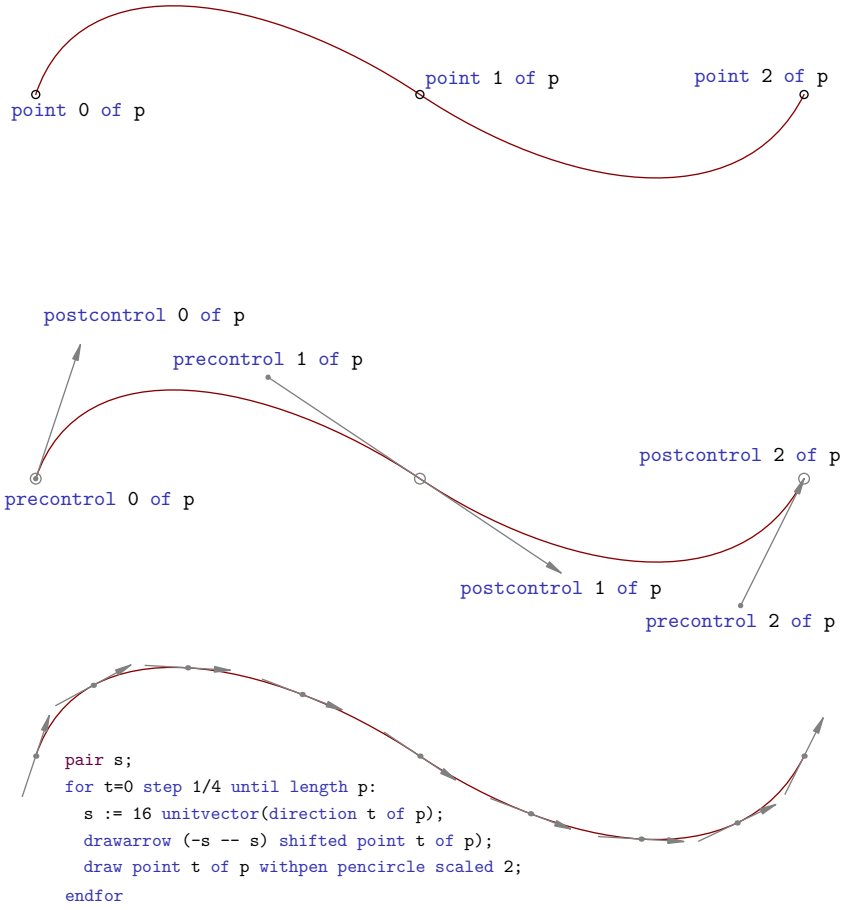
```
vardef direction expr t of p = postcontrol t of p - precontrol t of p enddef;
```

which can save you some typing. But the clever bit is that t does not have to be a whole number. If you set $t = \frac{1}{4}$ (say), METAPOST works out the corresponding fractional control points, so that you can use `direction t of p` to get a tangent at any point.

The vector pairs returned have the right direction, but still have rather arbitrary magnitudes, so the usual idiom is something like this:

```
path s; s = origin -- 36 unitvector(direction t of p);
drawarrow s shifted point t of p;
```

or the snippet shown on the right ⟶



9.11 Lines tangent to a circle

The techniques of the preceding section can be used to add a tangent line to a given point on a circular path, but not to find the tangent lines from a given point outside a circle. To do this, you need to adapt the standard geometrical construction: for a given circle C and a point p, find the midpoint of p and the center of C; draw a semicircle through p, centred on this midpoint; the tangent point is where the semicircle intersects C. Given a suitable `path C` and `pair p` you can do this:

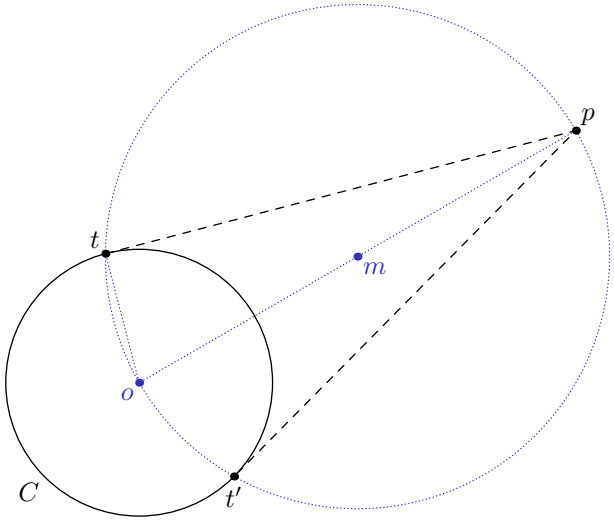
```
pair o, m, t, t'; o = center C; m = 1/2[o, p];
t  = C intersectionpoint halfcircle zscaled (p-o) shifted m;
t' = C intersectionpoint halfcircle zscaled (o-p) shifted m;
```

No parentheses are needed around the second path, because `intersectionpoint` is defined with `secondarydef`.

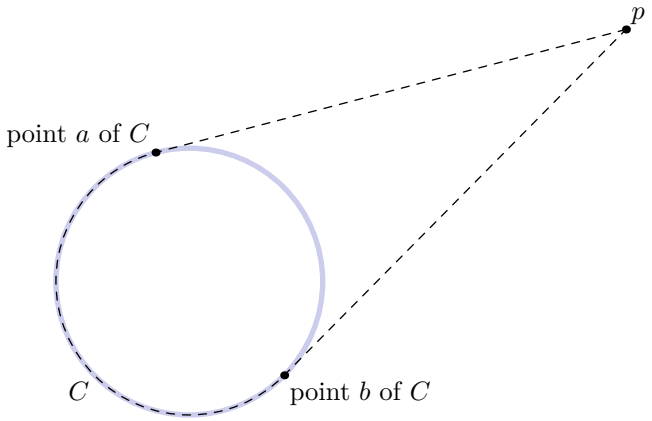
Things are a little more complicated if you want the points as times along the path C and you care about which tangent point is which. Here is a routine that returns the tangent points from p as two times a and b on C, with b adjusted so that $b > a$ in all cases regardless of the relative rotation of C and p. This means that `subpath (a, b) of C` is always the "long way round" C, on the opposite side from p, and `subpath (a, b-8) of C` is always the shorter segment.

```
% return a <pair> with the two times on C that
% correspond to the external tangents from p to C
vardef tangent_times(expr C, p) =
  save m, o, G, H, a, b;
  pair m, o; path G, H; numeric a, b;
  o = center C; m = 1/2[o, p];
  H = halfcircle zscaled (p-o) shifted m;
  G = halfcircle zscaled (o-p) shifted m;
  (a, whatever) = C intersectiontimes H;
  (b, whatever) = C intersectiontimes G;
  (a, b if b < a: + 8 fi)
enddef;
```

Note the elegant syntax here; if `z` is a ⟨pair⟩ then the operation `zscaled z` is equivalent to `scaled abs z rotated angle z`.



Here is the macro in action. Having obtained the two times a and b from the macro, the dashed line was drawn along a path that was composed with: `p -- subpath (a,b) of C -- cycle`



9.12 Lines tangent to two circles (exterior)

The same `tangent_times` macro can be reused to find the tangents that touch two circles, using an approach like this:

```
path A, B;
A = fullcircle scaled 144;
B = fullcircle scaled 60 shifted (200, 140);

numeric R, r;
R = abs (point 0 of A - center A);
r = abs (point 0 of B - center B);

path C; numeric t, u;
C = fullcircle scaled (2R-2r) shifted center A;
(t, u) = tangent_times(C, center B);

draw A withpen pencircle scaled 2 withcolor 3/4[blue, white];
draw B withpen pencircle scaled 2 withcolor 3/4[blue, white];

draw subpath (t, u) of A -- subpath (u-8, t) of B -- cycle;
```

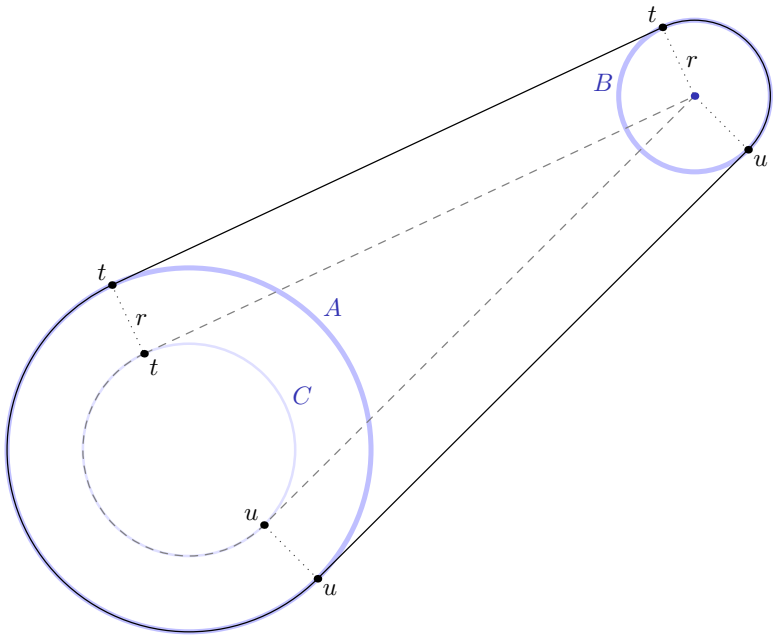
Here A and B are the two circles you want to connect, and A is larger than B. R is the radius of the larger, r of the smaller. C is an auxiliary circle centred at the same point as A and scaled so that its radius is $R - r$. If we then find the tangent points on C from the center of B, the points we want are the corresponding points on A and B.

Notice how the times are used with `subpath`; if $a > b$, then the path returned from `subpath (a, b) of P` is the same as `reverse subpath (b, a) of P`, which means that `subpath (u,t) of B` would give you the wrong side. The remedy is to subtract 8 from u (or, more generally, to subtract the length of path B). Because there are 8 points on a `fullcircle` path, point u and point $u - 8$ refer to the same place, but since $u - 8 < t$, the subpath will run clockwise as required.

☞ This all works provided that all three circles A, B, or C have the same rotation. But this may not always be the case. For example, you might have defined B as

```
B = fullcircle scaled 60 shifted 240 dir 36;
```

and then point t of B would *not* correspond to point t of the auxiliary circle C.



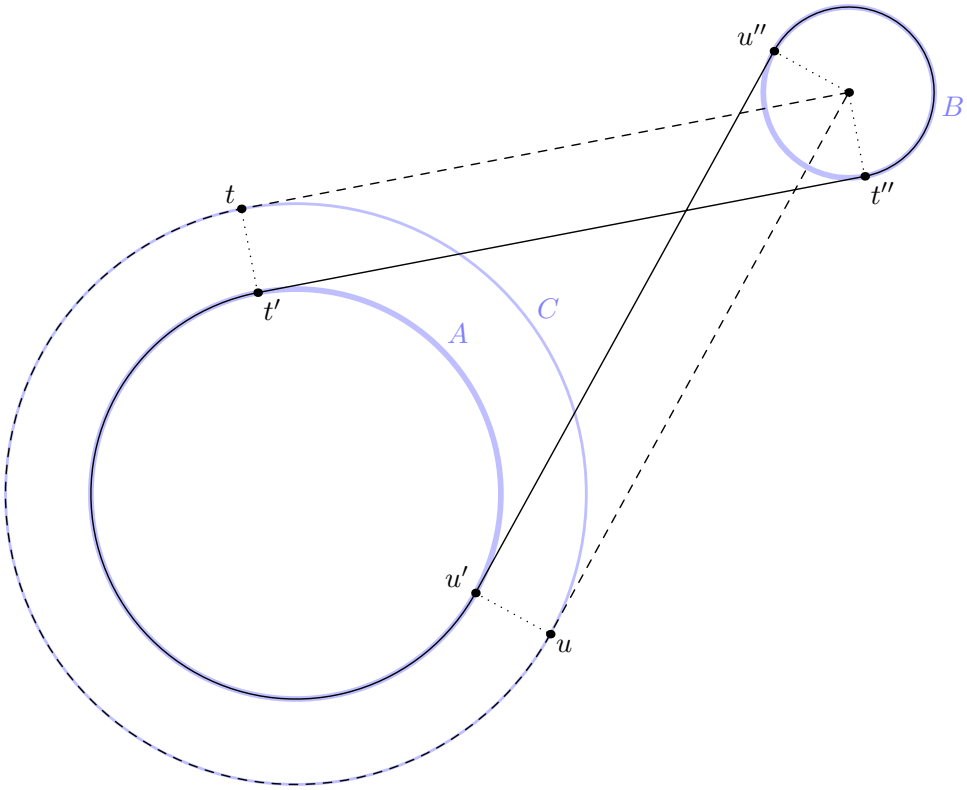
To cope with circles that might not have the same rotation, you need to adjust the tangent times to take account of the different relative rotation.

```
vardef adjust_time(expr tt, AA, BB) =
  tt + 1/45 angle (point 0 of AA - center AA)
  - 1/45 angle (point 0 of BB - center BB)
enddef;
```

This macro exploits the relationship between `angle` and points around a `fullcircle` path: 360° = 8 points. You can see it in action on the following page.

9.13 Lines tangent to two circles (interior)

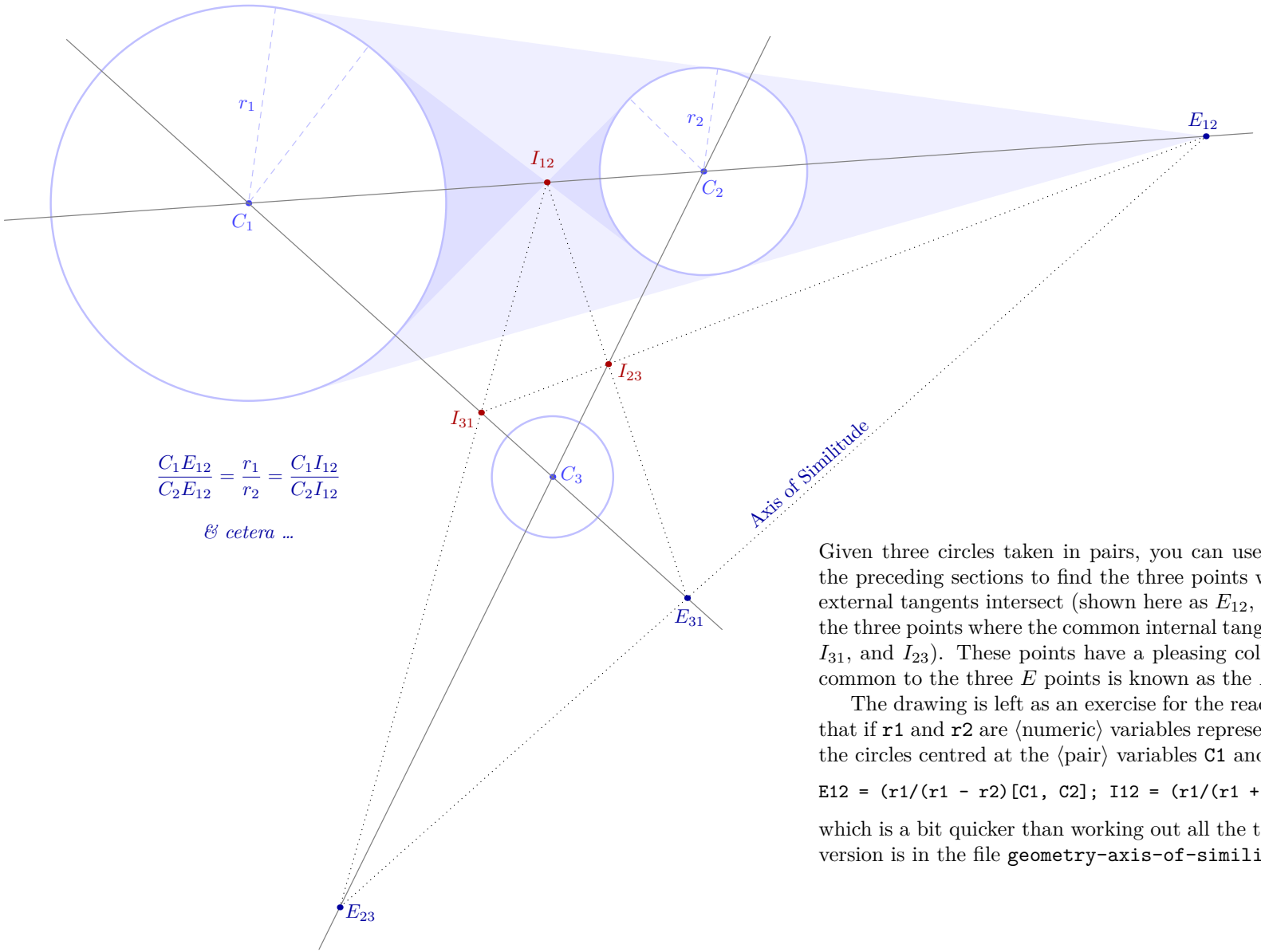
To find the interior tangents, you just need to add the smaller radius rather than subtract it, and add 4 to the times on the smaller circles, so that they are on the other side:



The complete code for this is shown on the right. It uses the same routines given above; `tangent_times` from section 9.11, and `adjust_time` from section 9.12.

```
path A, B; % randomly rotated to show off "adjust_time"
A = fullcircle scaled 144 rotated uniformdeviate 360;
B = fullcircle scaled 60 shifted 240 right rotated 36;

numeric R, r;
R = abs (point 0 of A - center A);
r = abs (point 0 of B - center B);

path C;
C = fullcircle scaled (2R+2r) shifted center A; % NB +ve

numeric t, t', t'', u, u', u'';
(t, u) = tangent_times(C, center B);
t' = adjust_time(t, C, A);
u' = adjust_time(u, C, A);
t'' = adjust_time(t + 4, C, B);  % Note the plus fours
u'' = adjust_time(u + 4, C, B);

draw A withpen pencircle scaled 2 withcolor 3/4[blue, white];
draw B withpen pencircle scaled 2 withcolor 3/4[blue, white];
draw C withpen pencircle scaled 1 withcolor 3/4[blue, white];

draw subpath (t', u') of A -- subpath (u'', t'') of B -- cycle;
draw center B -- subpath (t, u) of C -- cycle dashed evenly;

draw center B -- point t'' of B dashed withdots scaled 1/2;
draw center B -- point u'' of B dashed withdots scaled 1/2;
draw point t of C -- point t' of A dashed withdots scaled 1/2;
draw point u of C -- point u' of A dashed withdots scaled 1/2;

dotlabel.ulft(btex $t$   etex, point t   of C);
dotlabel.lrt (btex $t'$  etex, point t'  of A);
dotlabel.lrt (btex $t''$ etex, point t'' of B);
dotlabel.lrt (btex $u$   etex, point u   of C);
dotlabel.ulft(btex $u'$  etex, point u'  of A);
dotlabel.ulft(btex $u''$ etex, point u'' of B);
drawdot center B withpen pencircle scaled dotlabeldiam;

drawoptions(withcolor 1/2[blue, white]);
  label.urt(btex $A$ etex, point 1/2(t'+u'- 7.6) of A);
  label.rt (btex $B$ etex, point 1/2(t''+u''- 2) of B);
  label.urt(btex $C$ etex, point 1/2(t+u-8) of C);
drawoptions();
```

9.14 Axis of similitude



Given three circles taken in pairs, you can use the techniques of the preceding sections to find the three points where the common external tangents intersect (shown here as E_{12}, E_{31}, and E_{23}) and the three points where the common internal tangents intersect (I_{12}, I_{31}, and I_{23}). These points have a pleasing collinearity. The line common to the three E points is known as the *Axis of Similitude*.

The drawing is left as an exercise for the reader, except to note that if `r1` and `r2` are ⟨numeric⟩ variables representing the radius of the circles centred at the ⟨pair⟩ variables `C1` and `C2`, then:

```
E12 = (r1/(r1 - r2)[C1, C2]; I12 = (r1/(r1 + r2)[C1, C2];
```

which is a bit quicker than working out all the tangent points. My version is in the file `geometry-axis-of-similitude.mp`

9.15 Inversion, pole, and polar

Inversion in a circle is a generalization of reflection in a line. It is useful for certain constructions in geometry, and easy to implement as a macro METAPOST. For given circle, and a given point P lying outside the circle, the inverted point P' lies inside the circle at the intersection of the line from P to the centre of the circle, and the line between the tangent points [§9.11] from P, shown here as Q and R. ⟶

But OPQ and OQP' are similar triangles, so $r/OP = OP'/r$ and so $OP' = r^2/OP$, and since P' must lie on the line through O and P, this is enough to to find P' directly given P, O, and r:

```
P' = O + unitvector(P-O) scaled (r * r / abs (P-O));
```

But examining `plain.mp` shows that `unitvector` is a macro defined like this:

```
vardef unitvector primary z = z/abs z enddef;
```

which suggests this alternative formulation:

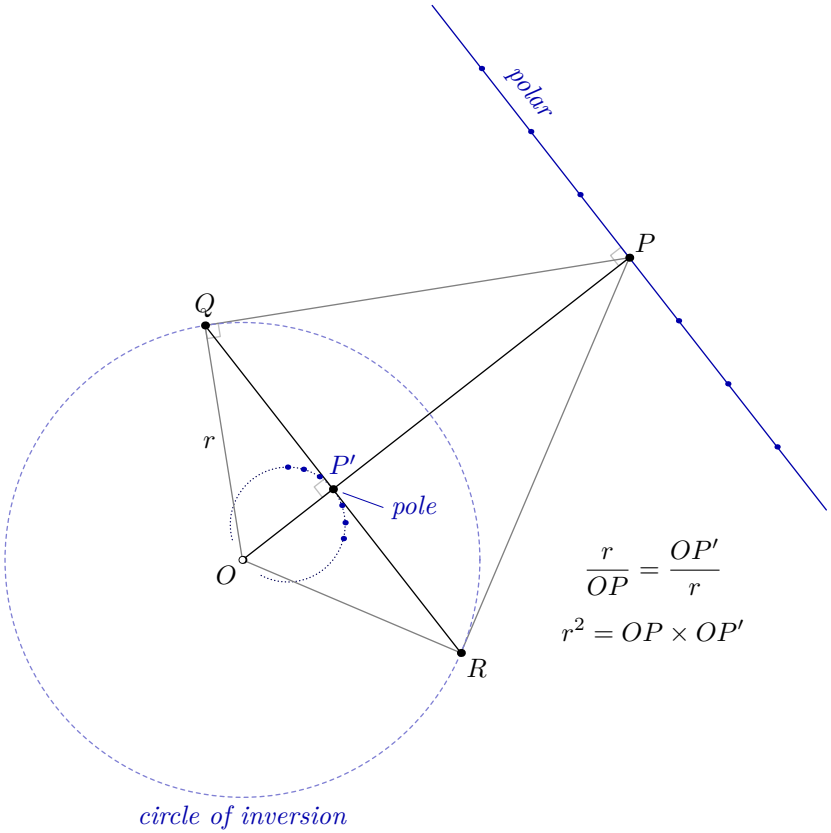
```
P' = O + (P - O) scaled (r / abs (P - O) * r / abs (P - O));
```

or as a macro, and dividing first to avoid overflow:

```
vardef invert(expr P, O, r) =
  save s; numeric s; s = abs(P - O);
  O + (P - O) / s * r / s * r
enddef;
```

This works well in most cases, but you could consider checking that s is not too small. If it is more convenient to deal with the ⟨path⟩ of the circle of inversion instead of the centre and the radius, you can get the macro to calculate them for you:

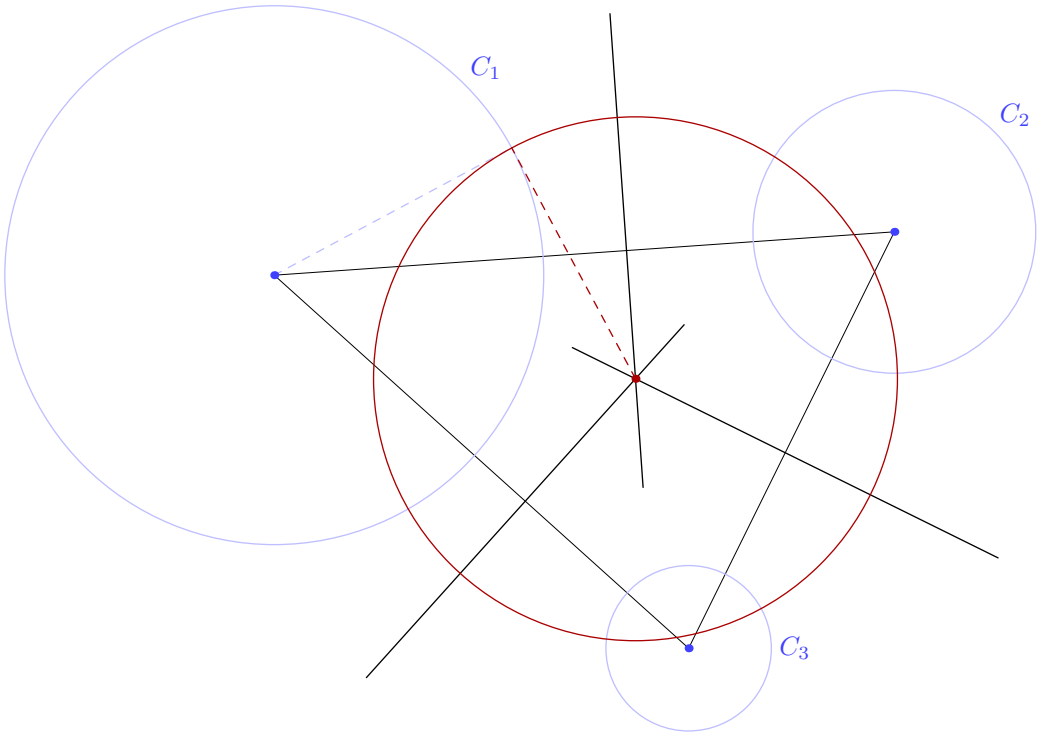
```
vardef invert(expr P, C) =
  save o; pair o; o = 1/2[point 0 of C, point 4 of C];
  save r; numeric r; r = abs (point 0 of C - o);
  save s; numeric s; s = abs (P - o);
  o + (P - o) * r / s * r / s
enddef;
```



- Inversion is reciprocal, so P is the inverse of P' above. Points on the circle of inversion invert to themselves.
- For any given line, the *pole* of the line with respect to a circle, is the inverse of the point on the line closest to the centre of the circle.
- For any given point, the *polar* of the point with respect to a circle, is the line through the inverse of the point perpendicular to the line through the point and the center of the circle of inversion.
- The inversion of points on the polar (shown as blue dots) lie on a circle through O and P'. The complete circle would be the inversion of the infinitely extended polar.

9.16 Radical axis and radical centre

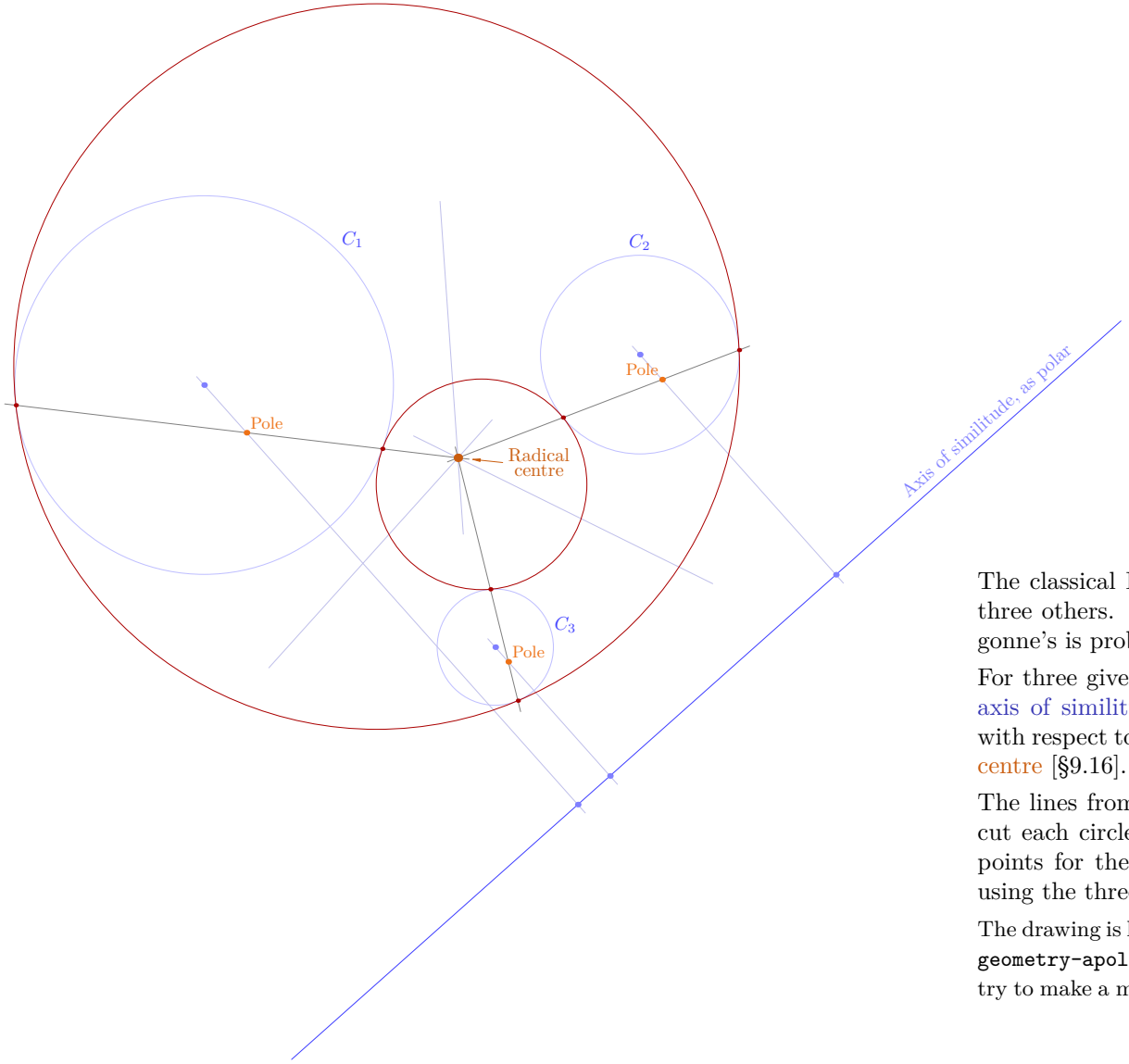
The *radical axis* of two circles is the line, orthogonal to the line between the centres of the two circles which is the locus of points which have equal power with respect to both circles; that is the points from which the tangents to each circle are of equal length. A circle centred at any point on the axis, and drawn with radius equal to the length of the tangent will cut both circles at right angles.



In a system of three circles as shown, the *radical centre* (`radix`) is the intersection of the three mutual radical axes. The tangents from this point to all three circles have the same length, so a circle with this radius (shown above in red) cuts all three circles at right angles.

```
vardef radical_axis(expr ca, cb) =
  numeric t, d, ra, rb;
  ra = abs(center ca - point 0 of ca);
  rb = abs(center cb - point 0 of cb);
  d = abs(center cb - center ca);
  2t = 1 + (ra+rb) / d * (ra-rb) / d;
  (up -- down) scaled 89
    rotated angle (center cb - center ca)
    shifted t[center ca, center cb]
enddef;
beginfig(1);
  z1 = origin; z2 = 233 dir 4; z3 = 209 dir -42;
  path c[], a[];
  c1 = fullcircle scaled 202 shifted z1;
  c2 = fullcircle scaled 106 shifted z2;
  c3 = fullcircle scaled  62 shifted z3;
  a1 = radical_axis(c1, c2);
  a2 = radical_axis(c2, c3);
  a3 = radical_axis(c3, c1);

  pair radix, t; path orthogonal_circle;
  radix = whatever[point 0 of a1, point 1 of a1]
        = whatever[point 0 of a2, point 1 of a2];
  t = c1 intersectionpoint
      halfcircle zscaled (radix-z1) shifted 1/2[radix, z1];
  orthogonal_circle = fullcircle
    scaled 2 abs (t - radix) shifted radix;

  draw z1 -- z2 -- z3 -- cycle withpen pencircle scaled 1/4;
  draw a1; draw a2; draw a3;
  drawoptions(withcolor 3/4[blue, white]);
    draw c1; draw c2; draw c3;
    draw z1 -- t dashed evenly;
  drawoptions(withcolor 2/3 red);
    draw orthogonal_circle;
    draw radix -- t dashed evenly;
    draw radix withpen pencircle scaled dotlabeldiam;
  drawoptions(withcolor 1/4[blue, white]);
    draw z1 withpen pencircle scaled dotlabeldiam;
    draw z2 withpen pencircle scaled dotlabeldiam;
    draw z3 withpen pencircle scaled dotlabeldiam;

  label.urt(btex $C_1$ etex, point 1 of c1);
  label.urt(btex $C_2$ etex, point 1 of c2);
  label.rt (btex $C_3$ etex, point 0 of c3);
endfig;
```

9.17 Circles tangent to other circles



The classical Problem of Apollonius is to find a circle tangent to three others. All of the approaches are rather involved, but Gergonne's is probably the simplest to follow in METAPOST.

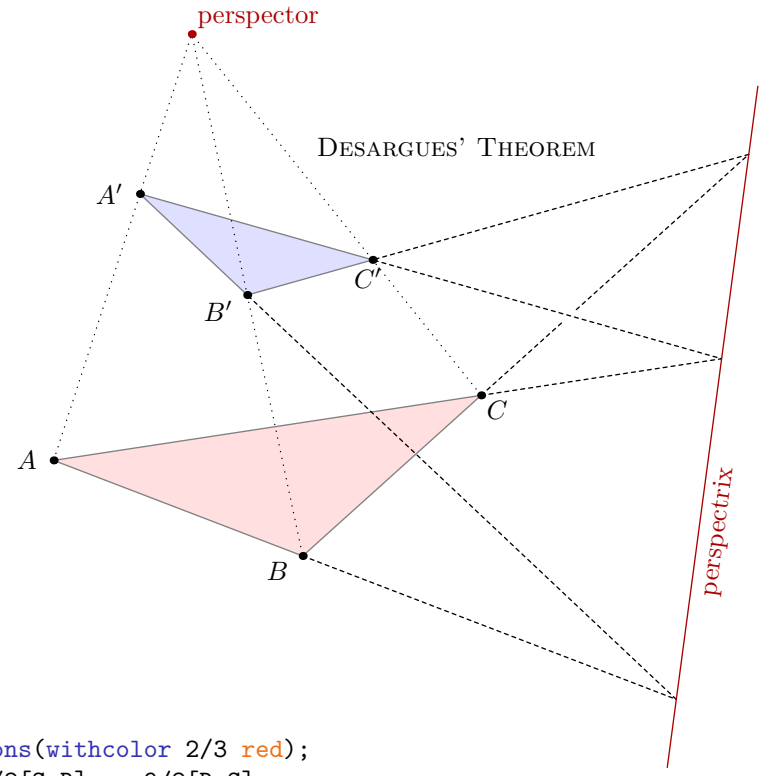
For three given circles C_1, C_2, and C_3, you first find the external axis of similitude [§9.14]; then find the poles [§9.15] of this line with respect to each of the three circles; and thirdly find the radical centre [§9.16].

The lines from the radical centre through each of the three poles cut each circle in two places. These six points show the tangent points for the two tangent circles, and you can draw the circles using the three point circle technique [§9.6].

The drawing is left as an exercise for the reader, although you can find my `geometry-apollonius.mp` in the source for this document. You might try to make a more robust version or to find all the other tangent circles.

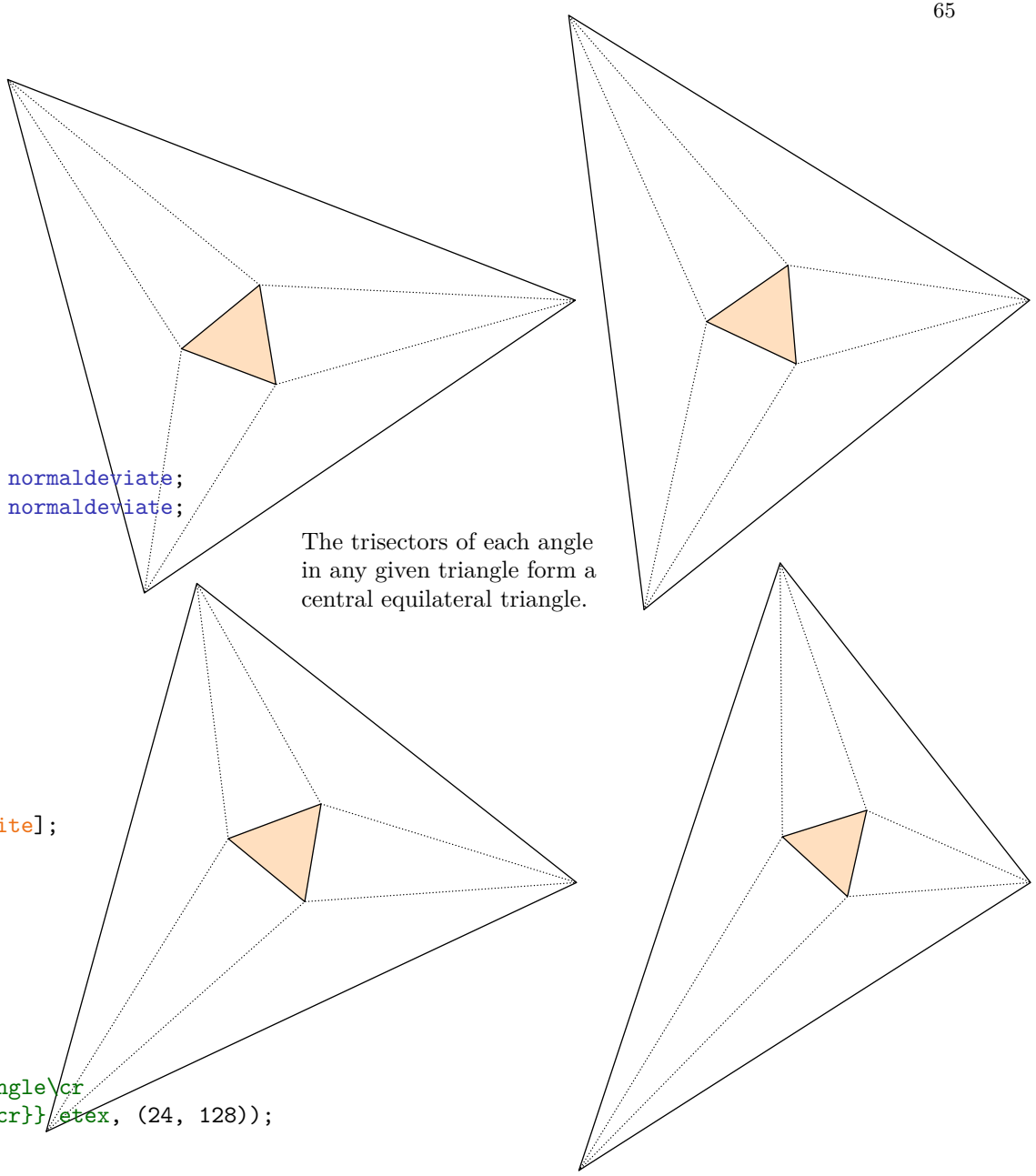
9.18 Coordinate geometry examples

```
beginfig(1);
  pair P, A, B, C, A', B', C', R, S, T;
  P = 200 dir 102;
  A = 100 dir 159; B = origin; C = 90 dir 42;

  A' = 3/8[P, A];  % the factors should not
  B' = 1/2[P, B];  % be the same!
  C' = 5/8[P, C];

  R = whatever [A, B] = whatever [A', B'];
  S = whatever [B, C] = whatever [B', C'];
  T = whatever [C, A] = whatever [C', A'];

  path t[];
  t1 = A -- B -- C -- cycle;
  t2 = A' -- B' -- C' -- cycle;
  fill t1 withcolor 7/8[red, white];
  fill t2 withcolor 7/8[blue, white];
  draw t1 withcolor 1/2 white;
  draw t2 withcolor 1/2 white;

  drawoptions(dashed withdots scaled 1/2);
  draw P--A;
  draw P--B;
  draw P--C;

  drawoptions(dashed evenly scaled 1/2);
  draw B--R--B';
  draw C--S--C';
  undraw subpath (1/4, 3/4) of (C'--T) withpen
      pencircle scaled 5;
  draw C--T--C';
```



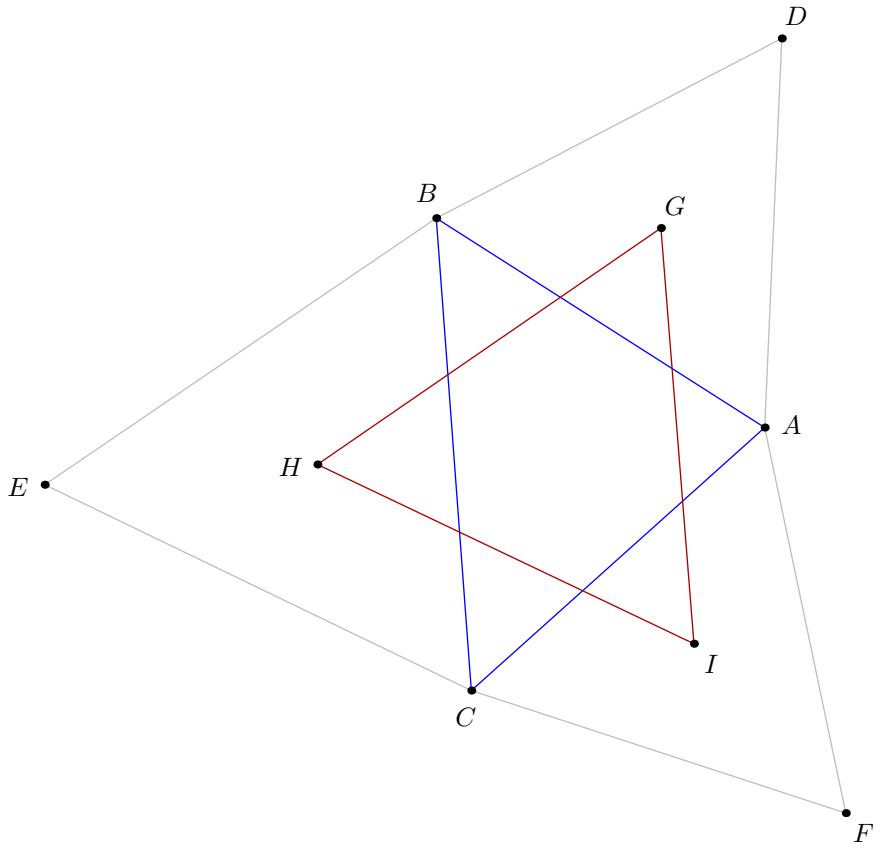

```
  drawoptions(withcolor 2/3 red);
    draw 9/8[S,R] -- 9/8[R,S];
    draw thelabel.bot("perspectrix", origin)
      rotated angle (T-R) shifted 1/2[T, R];
    dotlabel.urt("perspector", P);
  drawoptions();

  dotlabel.lft ("$A$ ", A);
  dotlabel.llft("$B$ ", B);
  dotlabel.lrt ("$C$ ", C);
  dotlabel.lft ("$A'$ ", A');
  dotlabel.llft("$B'$ ", B');
  dotlabel.bot ("$C'$ ", C');
  label.rt("\textsc{Desargues' Theorem}", 1/2[P, C'] shifted 10 right);
endfig;
```

```
randomseed := 2485.81543;
vardef measured_angle(expr p, o, q) =
   (angle (p-o) - angle (q-o)) mod 360
enddef;
beginfig(1);
picture T;
for i=0 upto 1:
  for j=0 upto 1:
    clearxy;
    T := image(
      z1 = (120 + uniformdeviate 21, 0);
      z2 = (120 + uniformdeviate 21, 0) rotated 120 rotated 21 normaldeviate;
      z3 = (120 + uniformdeviate 21, 0) rotated 240 rotated 21 normaldeviate;
      numeric a, b, c;
      a = measured_angle(z3, z1, z2);
      b = measured_angle(z1, z2, z3);
      c = measured_angle(z2, z3, z1);
      z4 = whatever [z1, z2 rotatedabout(z1, 1/3 a)]
         = whatever [z2, z3 rotatedabout(z2, 2/3 b)];
      z5 = whatever [z2, z3 rotatedabout(z2, 1/3 b)]
         = whatever [z3, z1 rotatedabout(z3, 2/3 c)];
      z6 = whatever [z3, z1 rotatedabout(z3, 1/3 c)]
         = whatever [z1, z2 rotatedabout(z1, 2/3 a)];
      fill z4--z5--z6--cycle withcolor 3/4[red + 1/2 green, white];
      draw z4--z5--z6--cycle;
      draw z1 -- z4 -- z2 -- z5 -- z3 -- z6 -- cycle
           dashed withdots scaled 1/4;
      draw z1 -- z2 -- z3 -- cycle;
    );
    draw T shifted (200i, 240j);
  endfor
endfor
label.rt(btex \vbox{\halign{#\hfil\cr The trisectors of each angle\cr
in any given triangle form a\cr central equilateral triangle.\cr}} etex, (24, 128));
endfig;
```



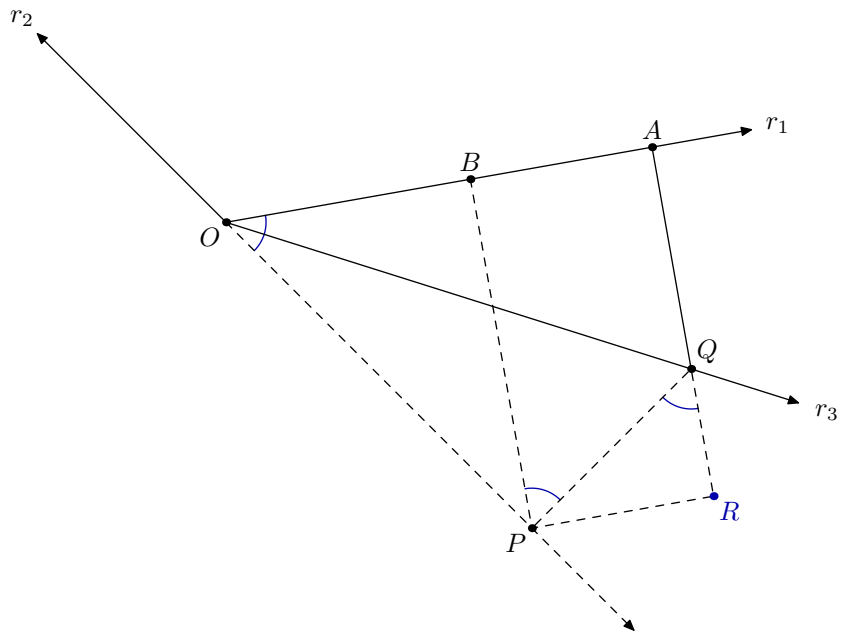
The trisectors of each angle in any given triangle form a central equilateral triangle.

```
def equilateral_triangle_point(expr a, b) =
  a rotatedabout(b, 60)
enddef;
primarydef o beyond z = z scaled (1+o/abs(z)) enddef;
beginfig(1);
  randomseed := 543.01811;

  pair A, B, C, D, E, F, G, H, I;
  A = 91 dir (  0 + 18 normaldeviate);
  B = 92 dir (120 + 18 normaldeviate);
  C = 93 dir (240 + 18 normaldeviate);

  D = equilateral_triangle_point(A, B);
  E = equilateral_triangle_point(B, C);
  F = equilateral_triangle_point(C, A);

  G = 1/3(A + B + D);
  H = 1/3(B + C + E);
  I = 1/3(C + A + F);

  draw A -- B -- C -- cycle withcolor blue;
  draw A -- D -- B -- E -- C -- F -- cycle withcolor 3/4;
  draw G -- H -- I -- cycle withcolor 2/3 red;

  forsuffixes @ = A, B, C, D, E, F, G, H, I:
    draw @ withpen pencircle scaled dotlabeldiam;
    label("$" & str @ & "$", 10 beyond @);
  endfor

  label.bot("\textsc{Napoleon's Theorem}",
    point 1/2 of bbox currentpicture);
endfig;
```



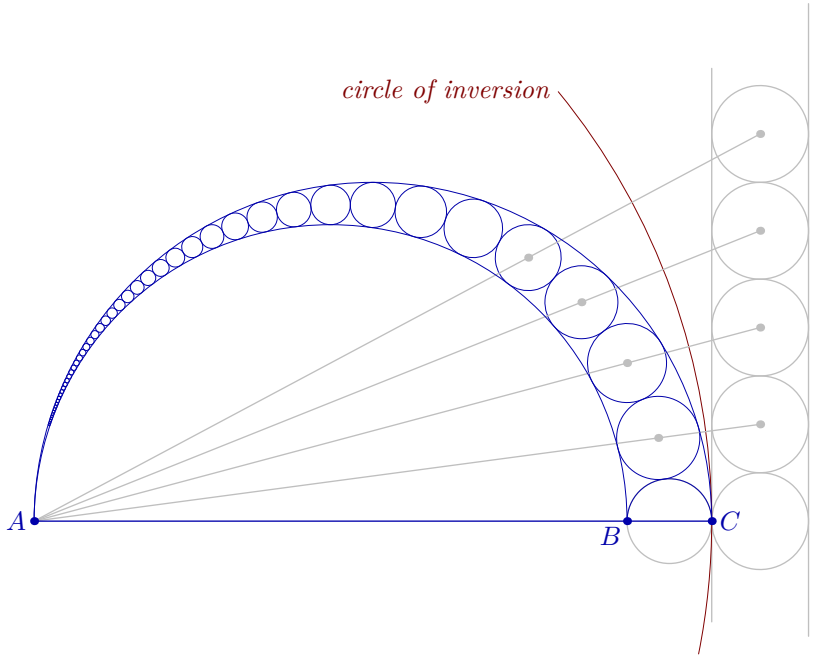
Napoleon's Theorem

```
beginfig(1);
% define the end points of the three rays
z1 = right scaled 200 rotated 10;
z2 = right scaled 100 rotated 135;
z3 = right scaled 225 rotated -17.5;
% define the other points, relative to Q
pair A, B, P, Q, R;
Q = 0.8125 z3;
A = whatever[origin, z1]; A-Q = whatever * z1 rotated 90;
P = whatever[origin, z2]; P-Q = whatever * z2 rotated 90;
B = whatever[origin, z1]; B-P = whatever * z1 rotated 90;
R = whatever[A,Q];        R-P = whatever * (B-P) rotated 90;
% mark the angles
drawoptions(withcolor .67 blue);
path c; c = fullcircle scaled 30;
draw c rotated angle (Q-P) shifted P cutafter (P--B);
draw c rotated angle (P-Q) shifted Q cutafter (Q--R);
draw c rotated angle P               cutafter (origin--z1);
drawoptions();
% draw the rays and A--Q
drawarrow origin -- z1; label(btex $r_1$ etex, z1 scaled 1.05);
drawarrow origin -- z2; label(btex $r_2$ etex, z2 scaled 1.08);
drawarrow origin -- z3; label(btex $r_3$ etex, z3 scaled 1.05);
draw A--Q;
% draw the dashed lines
drawoptions(dashed evenly);
draw B--P--R--Q--P; drawarrow origin -- P scaled 4/3;
drawoptions();
% label the points
dotlabel.urt(btex $Q$ etex, Q);
dotlabel.top(btex $A$ etex, A);
dotlabel.lrt(btex $R$ etex, R) withcolor .67 blue;
dotlabel.top (btex $B$ etex, B);
dotlabel.llft(btex $P$ etex, P);
dotlabel.llft(btex $O$ etex, origin);
endfig;
```



```
beginfig(1);
  pair A,B,C; A = origin; C = 254 right;  B = 7/8[A, C];
  path c[];
  c1 = fullcircle scaled 2 abs(A-C); % large circle for the inversions
  c2 = fullcircle scaled abs(A-C) shifted 1/2[A,C];
  c3 = fullcircle scaled abs(A-B) shifted 1/2[A,B];
  c4 = fullcircle scaled abs(B-C) shifted 1/2[B,C];
  c5 = invert(c4,c1);

  drawoptions(withcolor 3/4 white);
    draw c4; draw c5;
    draw invert(subpath(-3/8, 3/2) of c2, c1);  % vertical lines
    draw invert(subpath(-3/8, 3/2) of c3, c1);

  drawoptions(withcolor 1/2 red);
    draw subpath(-1/4,7/8) of c1 withpen pencircle scaled 1/4;
    label.lft("\textit{circle of inversion}", point 7/8 of c1);

  numeric d; d = abs(point 0 of c5 - point 4 of c5); % diameter of c5
  for i=1 upto 48:
    path c, c'; c = c5 shifted (0, i*d); c' = invert(c, c1);
    draw c' withpen pencircle scaled 1/4 withcolor 2/3 blue;
    if i<5:
      drawoptions(withcolor 3/4 white);
        draw c; draw origin -- center c;
        draw center c withpen pencircle scaled dotlabeldiam;
        draw center c' withpen pencircle scaled dotlabeldiam;
      drawoptions();
    fi
  endfor

  drawoptions(withcolor 2/3 blue);
    draw A--C;
    draw subpath (0,4) of c2 withpen pencircle scaled 1/4;
    draw subpath (0,4) of c3 withpen pencircle scaled 1/4;
    draw subpath (0,4) of c4 withpen pencircle scaled 1/4;
    dotlabel.lft("$A$", A); dotlabel.llft("$B$", B); dotlabel.rt("$C$", C);
endfig;
```

```
vardef invert(expr P, C) =    % invert path P or pair P in circle C
  save I, r, s, T; pair I; I = center C; r = abs(point 0 of C-I);
  if pair P: s = abs(P-I); I if s > eps: + (P-I) / s * r / s * r fi
  elseif path P: T = length P; for t=0 upto T-1:
    invert(point t of P, C) .. endfor
    if cycle P: cycle else: invert(point T of P, C) fi
  fi
enddef;
```



One must also recognize that any attempt to illustrate geometry involves a basic fallacy. For example, a straight line is unbounded and infinitely thin and smooth, while any illustration is unavoidably of finite length, of positive thickness, and rough edged.

— Benoit Mandelbrot, *The Fractal Geometry of Nature*

9.19 Drawing angle marks

Observant readers will have noticed that the occasional angle marks in the preceding examples are generally drawn using plain METAPOST commands rather than a macro. This is partly in order to make the examples self-contained and partly to show what can be done with the default plain METAPOST format.

To mark a right angle at point a on the line $a\,..\,b$ you can do something like this:

```
draw unitsquare scaled 5 rotated angle (b-a) shifted a;
```

with a suitable pen and a suitable colour. You *could* write a macro to do this, but it hardly seems worth the effort.

On the other hand, you might want to make a macro for a curved angle mark since, it is a bit more cumbersome. It is probably simplest make the macro create only the required path, so you can use it with `draw` or `fill` as required. The idea here is that P, O, and Q are ⟨pair⟩ variables and r is a ⟨numeric⟩ representing the desired radius.

```
vardef angle_mark(expr P, O, Q, r) =
  fullcircle scaled 2r rotated angle (P - O)
    shifted O cutafter (O -- Q)
enddef;
```

Note that `(P-O)` returns a ⟨pair⟩, but `(O--Q)` make a ⟨path⟩. Less is usually more in coordinate geometry diagrams, but you could go on to make it much fancier if you wanted:

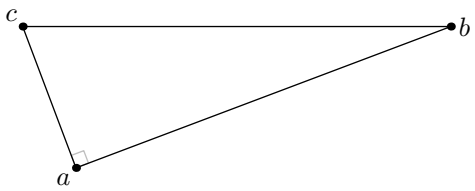
```
vardef fancier_angle_label(expr P, O, Q, r) = image(
  save a, t; path a; string t;
  a = fullcircle scaled 2r rotated angle (P - O) shifted O cutafter (O -- Q);
  fill O -- a -- cycle withcolor 7/8[red, white]; draw a withcolor 2/3 red;
  t = decimal (round(angle (Q-O) - angle (P-O)) mod 360) & "°";
  label(t, O + r * (unitvector(P-O) + unitvector(Q-O)));
) enddef;
```

Note that the angle label is calculated automatically. But it is tediously time-consuming to make this sort of macro completely general and fool-proof, so this example might not work for all angles.

For example, this is one way to annotate a right-angle triangle:

```
pair a, b, c; a = 10 dir 10; b = 160 dir 20;
c - a = whatever * (b - a rotated 90); ypart c = ypart b;
draw unitsquare scaled 5 rotated angle (b-a) shifted a
    withcolor 3/4;
draw a--b--c--cycle;
```

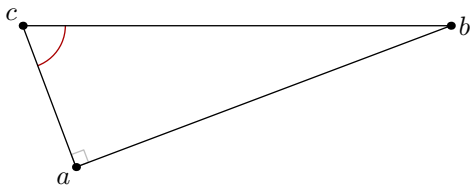
which produces this:



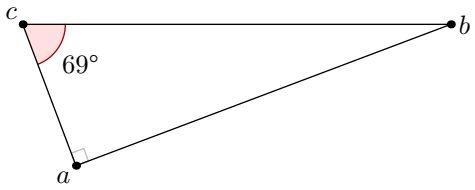
Equipped with the macro shown on the left, you could add this:

```
draw angle_mark(a, c, b, 16) withcolor 2/3 red;
```

to get this:



Or, using the fancier macro:



10 The missing trigonometry functions

METAPOST provides only two basic trigonometry functions, `sind` and `cosd`. This lack appears to be a deliberate design; in general it's much easier to use the `rotated` and `angle` functions than to work out all the sine, cosines and arc-tangents involved in rotating parts of your picture. But if you really want the 'missing' functions they are not hard to implement.

First you might want versions that accept arguments in radians instead of degrees. For this you need to know the value of π, but this is not built into plain METAPOST. If you are using the default number system then it's enough to define it to five decimal digits, but if you are using one of the new number systems you might want more digits of precision. In fact there's no harm in always defining these extra digits; even when you are using the default `scaled` number system, METAPOST will happily read as many extra digits of π as you supply, before it rounds the value to the nearest multiple of $\frac{1}{65536}$ (which turns out to be 3.14159). The same applies to the `double` number system, but the `binary` and `decimal` number systems will give you an error if you supply more digits that the default precision. So in general it's best to use no more than 32 digits. It's also possible, but not really worth the trouble, to define a routine to calculate π to the current precision.

However you define it, once you are armed with a value for π you can then define functions to convert between degrees and radians, and some more 'normal' versions of sine and cosine.

There's no built-in arccos or arcsin function but each is very easy to implement using a combination of the `angle` function and the Pythagorean difference operator.

METAPOST does have built-in functions for tangents; but they are called `angle` and `dir` and they are designed for pairs. So **angle** $(x, y) = \arctan(y/x)$ while **dir** 30 gives you the point (x, y) on the unit circle such that $\tan 30° = y/x$. You can use these ideas to define tangent and arctan functions if you really need them, but often `angle` and `dir` are more directly useful for drawing. You should also be aware that the `tand` function shown here does not check whether x is close to zero; if this is an issue, then add something like this at the appropriate point:

if *abs*$(x) <$ *eps*: *infinity* **else**: y/x **fi**

```
numeric pi;
% approximate value
pi := 3.14159;
% measure round a circular arc...
pi := 1/4 arclength (quartercircle scaled 16);
% up to 32 digits of precision
pi := 3.1415926535897932384626433832795;
% as many digits as are needed...
vardef getpi =
  save lasts, t, s, n, na, d, da;
  lasts=0; s=t=3; n=1; na=0; d=0; da=24;
  forever:
    exitif lasts=s;
    lasts := s;
    n := n+na; na := na+8;
    d := d+da; da := da+32;
    t := t*n/d;
    s := s+t;
  endfor
  s
enddef;
pi := getpi;

% conversions
def degrees(expr theta) = theta / pi * 180 enddef;
def radians(expr theta) = theta / 180 * pi enddef;
% trig functions that expect radians
def sin(expr theta) = sind(degrees(theta)) enddef;
def cos(expr theta) = cosd(degrees(theta)) enddef;
% inverse trig functions
def acosd(expr a) = angle (a,1+-+a) enddef;
def asind(expr a) = angle (1+-+a,a) enddef;
def acos(expr a) = radians(acosd(a)) enddef;
def asin(expr a) = radians(asind(a)) enddef;
% tangents
vardef tand(expr theta) = save x,y; (x,y)=dir theta; y/x enddef;
def atand(expr a) = angle (1,a) enddef;
```

11 Traditional labels and annotations

This section describes labels and annotations in what can be called the traditional METAPOST environment, where your figures are compiled with `mpost`*. The section after this describes labels & annotations in the newfangled (but better) world of* `lualatex` *and the* `luamplib` *package.*

METAPOST does not draw text directly; but it provides two different mechanisms to turn some text into a ⟨picture⟩, which can then be treated like any other; saved as a variable, drawn directly, or transformed in some way with a scaling, a reflection, or a rotation. The first mechanism is described below, the second in §11.2.

11.1 Simple strings in PostScript fonts with `infont`

The first mechanism is the primitive binary operation `infont`. As explained in section 8.3 of the METAPOST manual, it takes two strings as arguments: the left hand argument is the string of text to be printed; the right hand argument is the name of the font to use; and the result is a `picture` primary.

To make a suitable string you can enclose your text in double quotes to make a string token, or to refer to a ⟨string⟩ variable, or do one of these:

- Concatenate two other strings with `&`.
- Use `substring (a,b) of s` to get a substring of string `s`.
- Use `min(a,b,...)` or `max(a,b,..)` to find the lexicographically smallest (or largest) string in the list `a,b,...`. The list must have at least two entries, and they must all be strings.
- Use `char` to convert a numeric expression to the corresponding ASCII code; the numeric expression is rounded to the nearest integer modulo 256.
- Use `decimal` to get a string representing the value of a numeric expression.
- Apply `str` to any suffix (and hence to any variable). You get back a string representation of the suffix or variable name.
- Use `readfrom` to read one line from a file as a string.
- Use `fontpart` to extract the name of the font used in a picture created with `infont` — the string will be empty if there's no text element in the picture.
- Use `textpart` to get the text used in a picture created by `infont` — the string will be empty if there's no text element in the picture.

To find the name of a suitable font, you have to consult your local `psfonts.map` file, and probably the PSNFSS documentation. Here are a few of the many fonts available on my local TEX installation; the name to use with `infont` is in the first column.

`pagk8r`	Avant Garde	Hand in glove 42
`pbkl8r`	Bookman	Hand in glove 42
`pcrr8r`	Courier	Hand in glove 42
`phvr8r`	Helvetica	Hand in glove 42
`pncr8r`	New Century Schoolbook	Hand in glove 42
`pplr8r`	Palatino	Hand in glove 42
`ptmr8r`	Times	Hand in glove 42
`pzcmi8r`	Zapf Chancery	*Hand in glove 42*
`pzdr`	Zapf Dingbats	❁✪✻❄❄❄
`psyr`	Symbol	αβχδεϕ
`eurm10`	Euler	𝔞𝔟𝔠𝔡𝔢𝔣

The text example in the first line of this table was produced with

```
draw "Hand in glove 42" infont "pagk8r" shifted (124,144);
```

Note that in PostScript terms each of these font names refers to a combination of three files: an encoding that maps the characters you type to the glyphs in the font; a font metrics file that defines the sizes of the virtual boxes surrounding these glyphs; and a set of PostScript routines that actually draw them. In a TEX installation these combinations are defined in a font map file, usually called `psfonts.map`. If you run `mpost` with the `-recorder` switch it will write an extra log file (with a `.fls` extension) that lists the names of all the files used in a job. The actual font map file in use will be one of these. You can then browse it to find a definitive list of the font names you can use with your local METAPOST.

11.1.1 Character sets used by `infont` to set text

Standard METAPOST is configured to accept as input only space and the usual 94 visible ASCII characters (that is the characters numbered 32 to 126 in the tables at the right), but you can use any 8-bit characters as the payload of a string. However, plain METAPOST is set by default to use `cmr10`, the familiar Computer Modern typeface developed by Knuth for TEX, and unfortunately, this is encoded using the TEX text font encoding (also known as 'OT1', and as shown in the first table in Appendix F of the *TEXbook*). From the point of view of using `infont` to make simple labels, this means that the characters for space and seven other characters (`< > \ _ { | }`) are in the wrong place. You are likely to notice this first if you try to set a label with two words; the space will come out as a small diagonal stroke accent that is used in plain TEX to make the characters Ł and ł, used in Polish and other Slavic languages.

To fix this you should change the default font at the start of your program:

```
defaultfont := "texnansi-lmr10";  % for Computer Modern Roman
```

If you want `cmss10`, use `texnansi-lmss10` and so on. The encoding is shown on the right. The characters printed in black correspond to the widely used ISO Latin 1 encoding. If you want to use one of the standard PostScript fonts listed on the previous page, then the encoding to use is either `8y` to get the same `texnansi` arrangement or `8r` to get the arrangement shown in the lower table.

Choosing a font with one of these encodings means that if you use Windows code page 1252 or ISO Latin 1 as the encoding for your text editor, you can create labels with accented characters using `infont` and without resorting to `btex ... etex`. But if you are using UTF-8 characters (as many of us are now), then you have to do some extra work to get them printed correctly with `infont`. A solution is shown on the next page.

When labelling a drawing, it is always possible to use `btex ... etex` to produce your accented characters as discussed in section 11.2 below; but it may be that you are using METAPOST to represent data and labels supplied from some other program or a website. In this case it can be useful to be able to work with at least a subset of UTF-8 input. This is discussed in the next section.

Font: texnansi-lmr10

0	▫	€			/	·	″	¸	fl		▫	ff	fi	▫	ffi	ffl
16	ı	ȷ	`	´	ˇ	˘	¯	°	¸	ß	æ	œ	ø	Æ	Œ	Ø
32		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	¨
128	Ł	'	‚	f	„	…	†	‡	ˆ	‰	Š	‹	Œ	Ž	ˆ	–
144	ł	‘	’	“	”	•	–	—	˜	TM	š	›	œ	ž	~	Ÿ
160	▫	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Font: pplr8r

0	▫	˙	fi	fl	/	″	Ł	ł	¸	°	▫	˘	–	▫	Ž	ž
16	ˇ	ı													`	'
32		!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
128			‚	f	„	…	†	‡	ˆ	‰	Š	‹	Œ			
144			“	”	”	•	–	—	˜	TM	š	›	œ			Ÿ
160	▫	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
176	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	Đ	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

11.1.2 Mapping a subset of UTF-8 for `infont`

UTF-8 is a way of representing 16-bit Unicode characters with sequences of 8-bit characters. So your UTF-8 aware editor may show you an é but METAPOST, knowing nothing about UTF-8, will see this as Ã©. But you can write a fairly simple routine to decode a commonly-used subset of UTF-8.

```
vardef decode(expr given) =
  save a,b,i,s,out; string s, out; numeric a, b, i;
  out = ""; i=0;
  forever:
    i := i+1; s := substring(i-1,i) of given; a := ASCII s;
    if a < 128:
    elseif a = 194:
      i := i+1; s := substring (i-1,i) of given;
    elseif a = 195:
      i := i+1; s := char (64 + ASCII substring (i-1,i) of given);
    else:
      s := "?";
    fi
    out := out & s;
    exitif i >= length given;
  endfor
  out
enddef;
```

Use it with `infont` like this: '`decode("café") infont "ptmr8r"`' to produce a normal ⟨picture⟩ that can be passed to `draw` or saved as usual.

```
draw "café noir £2.50" infont "ptmr8r";
draw decode("café noir £2.50") infont "ptmr8r" shifted 12 down;
defaultfont := "pncr8r";
label.rt("café noir £2.50", 24 down);
label.rt(decode("café noir £2.50"), 36 down);
```

Note that you can't just use `draw` with a string variable; you have to use `infont` to turn the string into a picture. On the other hand, `label` calls `infont` automatically, but you must explicitly set the default font, preferably to one with an encoding that is compatible with ISO Latin 1.

- You can extend this idea to cope with other UTF-8 characters, including those that use three bytes. The UTF-8 page on Wikipedia shows you how it works. Essentially you look at the values of the next 2 or 3 characters and then pick the appropriate character in your encoding with `char`. But your output is still, of course, limited to the 256 characters in your encoded font.
- If you get tired of typing `decode`, you could define a short cut with a shorter name. You could even write it as a primary without parentheses like this:

  ```
  def U primary s = if string s: decode(s) fi enddef;
  ```

  which would let you write:

  ```
  label.rt(U"café à la crème", (x,y));
  ```

- However, there's no point in making any of this too elaborate. If you really want proper Unicode support you should use METAPOST with LuaTEX. (See below in §12).

The fragment on the left produces:

cafÃ© noir Â£2.50
café noir £2.50

cafÃ© noir Â£2.50
café noir £2.50

The `label` macro automatically calls `infont` with the current value of `defaultfont`; notice how it also adds `labeloffset` space.

11.1.3 Typographical minus signs with `infont`

If you are producing labels for a numeric reference scale, like the axis of a chart, it is convenient to be able to write a loop like this:

```
for x=1 upto 3: label.bot(decimal x, (x*cm, 0)); endfor
```

to produce your labels, however if x is negative this does not come out so well, because the first character of the string produced by `decimal -1` is an `ASCII 45`, which is the hyphen character. What we need is the mathematical minus sign instead; this is what you get with `btex $-1$ etex` of course, but that's harder to put in a loop with traditional METAPOST. Instead you can do this:

```
string minus_sign;
minus_sign := char 143; % if you are using the texnansi encoding
minus_sign := char 12;  % if you are using the 8r encoding
for x = -3 upto 3:
  label.bot(if x<0: minus_sign & fi decimal abs(x), (x*cm, 0));
endfor
```

Note that this does not work with the default encoding used in `cmr10` because there is no minus sign available in that font. Plain TeX uses `char 0` from `cmsy10`.

11.1.4 Bounding boxes and clipping with `infont`

Once the encoding is fixed, the other two parts of a PostScript font are the font metrics and the programs that draw the actual glyphs. The font metrics define the width of each character and provide a kerning table to adjust the space between particular pairs. This means that certain characters will overlap each other or stick out beyond the bounding box of the picture produced by `infont`. This is not normally a problem unless the picture happens to be at the edge of your figure. In the first example observe how the last letter sticks out to the right; in the second a wider baseline has been added to prevent this. If you want this effect, but you don't want to see the baseline, then draw it using the colour `background`.

11.1.5 But what about the `label` command?

As a convenience, the plain METAPOST format provides a `label` macro that automatically turns strings into pictures for you using whatever font name is the current value of `defaultfont` and scaled to the current value of `defaultscale`.

Plain METAPOST defines a `label` macro (approximately) like this:

```
def label(expr s, z) =
  draw s if string s: infont defaultfont
                      scaled defaultscale fi shifted z
enddef;
```

plus some clever code to align the label for you.

11.1.6 Bounding boxes and alignment with `infont`

To allow you to align a text label on a specific point, METAPOST provides five unary operators to measure the bounding box of a picture; they are shown in red in the diagram, and you can use them to measure the width, depth, and height of a textual picture. You can also work out the location of the baseline of the text or the x-height, provided you know how much your picture has been shifted. The easiest way to do this is to measure the picture *before* you shift it.

```
picture pp; pp = "proof" infont "pplri8r";
```

Here the picture *pp* is created with the origin of the text sitting at coordinates $(0, 0)$; then you can get the dimensions like this

```
wd = xpart urcorner pp;
ht = ypart urcorner pp;
dp = ypart lrcorner pp;
```

In this particular case you will find that you have $wd = 20.47292$, $ht = 7.19798$, and $dp = -2.60017$. The depth is negative because the descenders on the *p* and the *f* in the chosen font stick down below the base line. The height is greater than the x-height, because the *f* also sticks up, so you need to make another measurement:

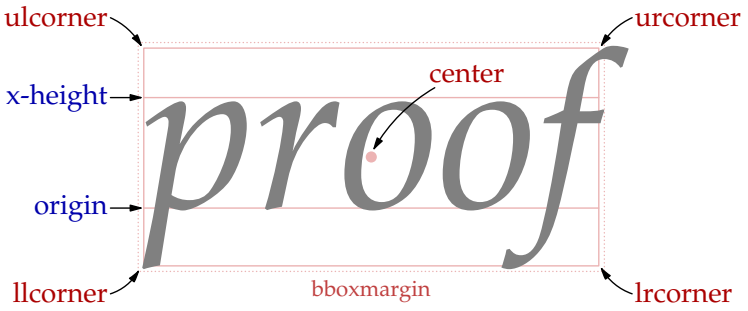
```
numeric xheight; xheight = ypart urcorner ("x" infont "pplri8r");
```

Armed with these measurements you can align your text labels neatly so that they are all positioned on the base line or vertically centred on the lower case letters regardless of any ascenders or descenders. To draw your label left-aligned with its origin at position (x, y) you just need to use: **draw** *pp* **shifted** (x, y). To draw it right-aligned, you subtract *wd* from the x-coordinate: **draw** *pp* **shifted** $(x - wd, y)$. Or to centre it, subtract $1/2\,wd$. To center it vertically on the lowercase letters, subtract $1/2\,xheight$ from the y-coordinate. You might of course like to wrap these adjustments up in your own convenient macro to help you maintain consistency in a diagram with many labels.

Alternatively you can adjust the bounding box of your textual picture and then use it with `label` as normal. Assuming *wd* is set to the width of your picture and *xheight* is set correctly for the current font, then

```
setbounds pp to unitsquare xscaled wd yscaled xheight;
```

will make the `label` alignment routines ignore any ascenders or descenders.



❖ Beware that if the resulting label is right at the edge of your drawing then any parts of the text that stick out of the adjusted bounding box will be clipped.
See also §13.3 for more on what happens if you rotate the text.

11.1.7 Setting Greek letters with `infont`

μῆνιν ἄειδε θεὰ Πηληϊάδειω Ἀχιλῆος

While it's technically possible to set the whole of Homer's *Iliad* using the Greek fonts available to `infont`, it's probably not a great use of time; on the other hand you might want to label parts of a diagram with Greek letters, and for single Greek letters `infont` is more than adequate.

The Greek letters for Computer Modern are in the maths-italic font `cmmi10`, which uses the encoding shown on p. 430 of *The TEXbook*. For historical reasons there's no omicron available, so you are supposed to use the *o* character instead. Fortunately you are unlikely to need more than the first few, and it's quite easy to remember that `char 11` = α, `char 12` = β, and so on. Producing the upper case letters is a bit more of a fiddle with this encoding as you need to know which ones use a Roman letter form; for details examine the program on the right, or check the table in §8.4.2 that shows Herman Zapf's elegant Euler font, available as `eurm10`. This makes a refreshing change for some diagrams.

If you have fonts installed from the Greek Font Society, then you get a wider choice, and a slightly more modern encoding. All of the plain letters are available in the normal ASCII positions, so you do not have to muck about with `char xx` so much. However in recent versions there is no character you can use as a word space, so if you want to set Greek text rather than individual letters, see §12.2.

32	̓	!		̔	%		′	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	·	̓		᾽	;
64	̓	A	B	̔	Δ	E	Φ	Γ	H	I	Θ	K	Λ	M	N	O
80	Π	X	P	Σ	T	Υ	″	Ω	Ξ	Ψ	Z	[	̓	]	̆	̔
96	̀	α	β	ς	δ	ϵ	ϕ	γ	η	ι	θ	κ	λ	μ	ν	ο
112	π	χ	ρ	σ	τ	υ		ω	ξ	ψ	ζ	«	̣	»	̂	—
128	ὰ	ά	ὰ	ᾶ	ὰ	ά	ὰ	ᾶ	ά	ά	ᾰ	ὰ	ά	ά	ᾰ	ᾶ
144	ᾶ	ᾶ	ᾶ		ᾶ	ᾶ	ᾶ		ὴ	ή	ὴ		ὴ	ή	ὴ	
160	ή	ή	ῆ	ῆ	ή	ή	ῆ	ῆ	ῆ	ῆ	ῆ	ῆ	ῆ	ῆ	ῆ	ῆ
176	ὼ	ώ	ὼ	ῶ	ὼ	ώ	ὼ	ῶ	ώ	ώ	ῶ	ῶ	ώ	ώ	ὼ	ῶ
192	ῶ	ῶ	ῶ		ῶ	ῶ	ῶ		ὶ	ί	ὶ	ῖ	ὺ	ύ	ὺ	ῦ
208	ί	ῖ	ῖ	ῖ	ύ	ῦ	ῦ	ῦ	ῖ	ῖ	ῖ	Ϊ	ῦ	ῦ	ῦ	Ÿ
224	ὲ	έ	ὲ	ῆ	ὸ	ό	ὸ	ῶ	έ	έ	ῆ	ῆ	ό	ῶ	ῶ	ῶ
240	ϊ	ῖ	ῖ	ῖ	ϋ	ῦ	ῦ	ῦ	ᾳ	ῃ	ῳ	ῥ	ῥ			

```
beginfig(1);
  string ab, AB;
  ab = ("" for i=11 upto 23: & char i endfor
     & "o" for i=24 upto 33: & char i endfor);
  AB = ("AB" & char 0 & char 1 & "EZH" & char 2 & "IK"
        & char 3 & "MNO" & char 4 & char 5 & "P"
        & char 6 & "T"   & char 7 & char 8 & "X"
        & char 9 & char 10);

  draw ab infont "cmmi10";
  draw AB infont "cmmi10" shifted 12 down;
  draw ab infont "eurm10" shifted 32 down;
  draw AB infont "eurm10" shifted 44 down;
endfig;
```

αβγδεζηθικλμνοξπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΟΞΠΡΣΤΥΦΧΨΩ

αβγδεζηθικλμνοξπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΟΞΠΡΣΤΥΦΧΨΩ

```
string ab, AB;
ab = "abgdezhjiklmnoxprstufqyw";
AB = "ABGDEZHJIKLMNOXPRSTUFQYW";
```

`draw ab infont "grmn1000"`
αβγδεζηθικλμνοξπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΟΞΠΡΣΤΥΦΧΨΩ

`draw ab infont "gporsonrg6r"`
αβγδεζηθικλμνοξπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΟΞΠΡΣΤΥΦΧΨΩ

`draw ab infont "gneohellenicrg6r"`
αβγδεζηθικλμνοξπρστυφχψω
ΑΒΓΔΕΖΗΘΙΚΛΜΝΟΞΠΡΣΤΥΦΧΨΩ

11.2 Setting text with `btex ... etex`

As soon as you need anything complicated in a label, like multiple fonts, multiple lines, or mathematics, you will find it easier to switch from `infont` to the `btex ... etex` mechanism that calls TEX to create your textual picture. In fact you might prefer to use TEX for all your labels, even simple strings, for the sake of consistency. The only downside is that this mechanism is a little bit slower.

The `btex` mechanism produces a textual picture just as `infont` does with a height, width, and depth that you can measure, and adjust, as discussed in section 11.1.6. And again, just like `infont` you can either use `draw` to place the resulting picture directly, or pass it to the `label` macro.

What you need to be aware of is that METAPOST places everything you put between the `btex` and `etex` into an `\hbox{...}` and processes it with plain TEX. This has several implications: on the positive side you have easy access to italics and bold letters, mathematical formulae, symbols like `$\alpha$`, and anything else you can normally put in an `\hbox{}`; but there are some restrictions especially if you want to do anything more than produce a simple single-line label in the default Computer Modern type face. The next few sections deal with some of the things you might want to do.

11.2.1 Producing display maths

One of the obvious restrictions that TEX imposes in restricted horizontal mode is that you can't use `$$ ... $$` to produce display maths. This means that the various mode-sensitive constructs like $\sum$ and $\int$ will come out in their smaller forms. And your fractions will look like they are $\frac{3}{4}$ size. If you want them big, then the solution is simple: just add `\displaystyle` at the beginning of your formula ⟶

```
...
label(btex $\displaystyle \int_0^t 3x^2\, dx$ etex, (x,y));
...
```

11.2.2 Getting consistent baselines for your labels

As already discussed §11.1.6, you can fiddle with the bounding box of a text picture to make the `label` macro line things up on a common baseline, but there is a much easier way with `btex` and `etex`. Plain TEX provides a `\strut` command that inserts an invisible rule that sticks up 8.5pt above the baseline and 3.5pt below. If you put one of these in each of the your labels, then they will all have the same vertical size and will all line up neatly: `label(btex \strut a etex, origin);`

11.2.3 Multi-line text labels

Another consequence of the `\hbox` feature is that there is no automatic text wrapping done for you, but again you can work round this easily because TEX lets you nest a `\vbox` inside an `\hbox`. This gives you proper paragraph-like wrapping but you will almost certainly need to adjust the line length, justification, and indentation in order to get a satisfactory result →

You may only need the full power of TEX's paragraph making system occasionally though: more usually you will just have one or two lines in each label, and you might be quite happy to control the line breaks manually. In this case it's helpful to wrap a little tabular structure around your text. Here's how to define something suitable in plain TEX. First you need to define a suitable macro at the start of your figure

```
verbatimtex
\def\s#1{\let\\\cr\vbox{\halign{\hfil\strut ##\hfil\cr#1\crcr}}}
etex
```

then you can write labels like this:

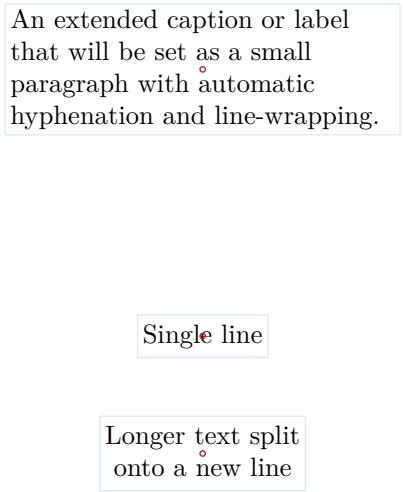
```
...
label(btex \s{Single line} etex, z1);
label(btex \s{Longer text split\\onto a new line} etex, z2);
...
```

Notice how you can still use the macro with single lines, you just get a one-line table as it were. Note also that the definition of `\s` as given will centre each line of the text under the one above. If you want them left aligned or right aligned, omit one of the `\hfil` commands. The three examples given above are typeset over here ⟶ The small red circle show the reference points and the pale blue lines the bounding boxes of the pictures that METAPOST gets back from TEX.

You can of course achieve the same effects using LATEX tabular structures, but then you have to use the `-tex=latex` option to run METAPOST.

Note: In case it's not obvious, if you want text wrapping or tabular arrangements as discussed here, you need to use `btex ... etex` to set your labels. There's no text wrapping with `infont`. On the other hand if all of your labels are done with `infont`, but you just want one extra that has two lines, you can split the text into two separate labels and position them independently.

```
...
label(btex \vbox{\hsize 2in\parindent 0pt\raggedright
    An extended caption or label that will be set as a
    small paragraph with automatic hyphenation and
    line-wrapping.
} etex, z0);
...
```



11.2.4 Pins and braces

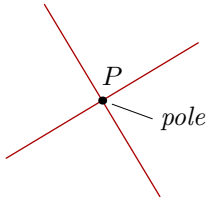
In some awkward corners, you may find that you just can't get your label in the right place with dotlabel even if you adjust labeloffset. In these cases there are two simple techniques you can use. First, you could separate drawing the dot from placing the label; given a point P you can try:

```
draw P withpen pencircle scaled dotlabeldiam;
label(btex $P$ etex, P shifted 10 dir 68);
```

Using dotlabeldiam ensures that your dots match any others done with dotlabel. Secondly, if that's not enough, use a temporary pair to create a call out line:

```
z0 = P + 20 dir -20;
draw z0 -- P
  cutafter fullcircle scaled 8 shifted P
  withpen pencircle scaled 1/4;
label.rt(btex \textit{pole} etex, z0);
```

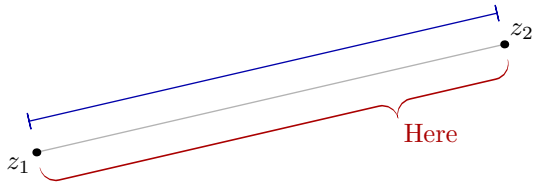
If you want to do this sort of thing often, then it might be worth making a macro; it is hard to write anything completely general, but see §12.6 for an example.



You might also want to mark a straight line between two points. The simplest way to do this is just to use drawdblarrow on a copy of your straight path shifted to one side, like so:

```
drawdblarrow (z1--z2) shifted (12 up rotated angle (z2-z1));
```

If you combine this with temporarily setting ahangle:=180, you get the simple dimension line shown in blue.



The red braces are a more complex variation on this theme ⟶

```
vardef do_brace(expr a, b, offset, r) =
  save d, e, m, n, brace, hook; pair e; path brace, hook[];
  d = angle(b-a);
  e = up scaled offset rotated d shifted r[a,b];
  n = 1/2 offset; m = abs(n);
  hook1 =                          origin {0, n} .. (m, n) {1,0};
  hook2 = (-m,-n) {1,0} .. {0, n} origin {0,-n} .. (m,-n) {1,0};
  hook3 = (-m, n) {1,0} .. {0,-n} origin;
  brace = (hook1 rotated d shifted a --
           hook2 rotated d shifted e --
           hook3 rotated d shifted b) shifted (up scaled n rotated d);
  draw brace withpen pencircle yscaled .6 xscaled .1666 rotated d;
  point 3 of brace
enddef;

label.lrt("Here", do_brace(z1, z2, -12, 3/4));
```

Note that, as well as drawing the braces, the macro uses the grouping provided by vardef to return the mid point so that you can put a label next to it.

11.2.5 Dynamic labels

If you are a maven of programming language syntax you may have noticed that `btex ... etex` fits into the type system that METAPOST inherits from METAFONT as a `picture` and not as a `string`. Effectively, `btex` and `etex` act as a special pair of quotation marks that create a picture; however the contents are used verbatim, so that the whole construction is a syntactical atom. This means that you **cannot** write this sort of thing:

```
for i=0 upto 4: % this won't work
  label(btex "$p_" & decimal i & "$" etex, (10i,0));
endfor
```

Given this input METAPOST would attempt to get TEX to typeset

```
\hbox{"$p_" & decimal i & "$"}
```

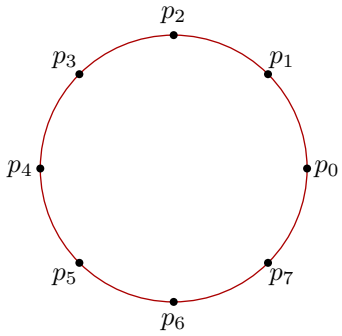
which would probably result in a 'Misplaced alignment tab character' error. To get round this problem, METAPOST provides a general mechanism to write out a string to a file, and then read the file back in. This is the mechanism used by the `TEX()` macro that is provided alongside `plain.mp`. This allows you to write:

```
input TEX
...
for i=0 upto 4:
  label(TEX("$p_" & decimal i & "$"), (10i,0));
endfor
```

This works because the TEX macro is expecting a `string` so the normal string concatenation rules are applied. The macro wraps the result with `btex` and `etex`, writes them out to a file, and then reads the file in again so that METAPOST gets the correct contents to pass to TEX.

The only trouble with this is that it makes METAPOST open a file, write to it, close it, and then read it in again for each label one at a time; this means that it's very slow. The example on the right shows how to speed things up, by using the same file for all the labels and only writing it once. The `write` command is a METAPOST primitive, and `EOF` is defined in `plain.mp`.

```
path c; c = fullcircle scaled 100; draw c withcolor .67 red;
for i=0 upto 7:
  fill fullcircle scaled 3 shifted point i of c;
  z[i] = point i of c scaled 1.15;
  write "label(btex $p_" & decimal i & "$ etex,("
        & decimal x[i] & "," & decimal y[i]
        & "));" to ".mplabels";
endfor
write EOF to ".mplabels";
input ".mplabels";
```



Note that you can't use `decimal` on a ⟨pair⟩ variable, but you can save the pair as a `z`-variable and then use the `x` and `y` syntax. The scaling trick used here only works because `c` is centred on the origin. If `c` were drawn elsewhere, you would have to write:

```
... point i of c shifted -center c
                scaled 1.15
                shifted center c ...
```

11.2.6 Matching fonts

Despite the apparent restriction of using plain TEX it is almost always possible to match the font and format of an enclosing LATEX document. The simplest approach is to use the plain TEX font mechanism with the names from `psfonts.map`.

```
verbatimtex
\font\rm=ptmr8r\rm
etex
```

Adding this at the top of your METAPOST program will set your text in Times New Roman, although any maths will still be set using Computer Modern. To fix this, all you have to do is to redefine all the maths fonts in all sizes you need; this is not really that hard but it is a fiddle to get all the details right. Fortunately the wonderful `font-change` package has done it all for you for a large range of fonts; with this package installed you can use

```
verbatimtex
\input font_times
etex
```

instead, and all of your TEX labels, including bold letters, italics, small caps, and mathematics will be set in Times New Roman.

If you still can't get your labels to match, you can force METAPOST to use LATEX instead of plain TEX. You need to use the `-tex` command line switch:

```
mpost -tex=latex
```

and also load the packages you need in a `verbatimtex` block at the top of your file ⟶

Plain `mpost` needs an old-fashioned `.dvi` file to work with, so you can only use an engine that still produces one, like `latex` or `elatex`, and not any of the more modern engines, like `pdflatex`. Generating the labels takes a little bit longer because you have to load rather more 'infrastructure' for LATEX, and you are limited to whatever font packages you have that work with the traditional LATEX engine. For a more modern approach, read on into section 12.

Here are some samples of the fonts available in the `font-change` package. For full details, and especially details about using AMS symbols, see the package documentation.

`font_times`	**NB**. Learn $v = u + at$ right *now*!
`font_palatino`	**NB**. Learn $v = u + at$ right *now*!
`font_charter`	**NB**. Learn $v = u + at$ right *now*!
`font_utopia`	**NB**. Learn $v = u + at$ right *now*!
`font_cmbright`	**NB**. Learn $v = u + at$ right *now*!
`font_century`	**NB**. Learn $v = u + at$ right *now*!
`font_concrete`	**NB**. Learn $v = u + at$ right *now*!
`font_bookman`	**NB**. Learn $v = u + at$ right *now*!
`font_arev`	**NB**. Learn $v = u + at$ right *now*!

```
verbatimtex
 \documentclass{article}
 \usepackage{mathpazo}
 \usepackage{xcolor}
 \begin{document}
etex
```

Note that the `\documentclass` and the `\begin{document}` lines are required, but METAPOST is smart enough to add an `\end{document}` for you.

11.2.7 Setting verbatim listings

There is a good chance that you will never need to set a verbatim listing in a METAPOST drawing, but if you do there are a couple of things to think about. The issue about setting text verbatim with TEX is that turning off the control characters can be tricky, so if you have text for a label with characters that are special in TEX like the backslash or the underscore, then the simplest thing to do is to avoid TEX completely and use `infont` instead.

```
string s; s = "\TeX\ sets maths like this $e=mc^2$";
draw ("1. " & s) infont defaultfont;
draw ("2. " & s) infont "texnansi-lmr10"  shifted 20 down;
draw ("3. " & s) infont "cmtt10"          shifted 40 down;
draw ("4. " & s) infont "texnansi-lmtt10" shifted 60 down;
```

But as you can see, (1) this is a bit of a disaster with the default font `cmr10` because it does not have all the glyphs in the usual ASCII positions (as noted above §11.1.1). The solution is to use the version of the font with the `texnansi` encoding (2), but you probably want it in the monofont (3) and as you can see `cmtt10` has the "visible space" character instead of a regular space. If this is not what you want then use the alternative encoding (4).

If you want more than this, then you really need to use LATEX to process the label, as discussed in §11.2.6, and load the appropriate preamble. ⟶

```
% special operators
vardef incr suffix $ = $:=$+1; $ enddef;
vardef decr suffix $ = $:=$-1; $ enddef;
def reflectedabout(expr w,z) = % reflects about the line w..z
 transformed begingroup transform T_;
   w transformed T_ = w;
   z transformed T_ = z;
   xxpart T_ = -yypart T_;
   xypart T_ = yxpart T_; % T_ is a reflection
T_ endgroup enddef;
```

```
1.␣"TeX␣"␣sets␣maths␣like␣this␣$e=mc^2$
2. \TeX \ sets maths like this $e=mc^2$
3.␣\TeX␣\␣sets␣maths␣like␣this␣$e=mc^2$
4. \TeX \ sets maths like this $e=mc^2$
```

```
prologues := 3; outputtemplate := "%j.eps";
verbatimtex
\documentclass{article}
\usepackage{listings}
\usepackage{xcolor}
\newcommand\mpstyle{\lstset{language=Metapost,
basicstyle=\ttfamily,columns=fullflexible,commentstyle=\color{red},
frame=none,keepspaces=true,showstringspaces=false}}
\lstnewenvironment{code}[1][]{\mpstyle\lstset{#1}}{}
\begin{document}
etex
beginfig(1);
picture P; P = thelabel(btex \vbox{\begin{code}
  % special operators
  vardef incr suffix $ = $:=$+1; $ enddef;
  vardef decr suffix $ = $:=$-1; $ enddef;
  def reflectedabout(expr w,z) = % reflects about the line w..z
   transformed begingroup transform T_;
     w transformed T_ = w;
     z transformed T_ = z;
     xxpart T_ = -yypart T_;
     xypart T_ = yxpart T_; % T_ is a reflection
  T_ endgroup enddef;
\end{code}} etex, origin);
fill bbox P withcolor (1,1,31/32); draw P; draw bbox P;
endfig; end.
```

☞ Compile this with `mpost -tex=latex` and use `epstopdf` to make a PDF.

12 Modern labels, annotations, and other goodies

This section is a re-working of the previous section, that attempts to show how much nicer it is to work in the new-fangled world of `luamplib`. If this is all new to you, you probably should start by doing `texdoc luamplib` on your system and reading the documentation provided with the package. In order to use these newfangled facilities you need to create your METAPOST diagrams inside a TEX-wrapper as explained above in §3.2.

The first thing to say is that everything in the preceding section will continue to work more or less the same when you use `luamplib` with LuaLATEX. It is designed to be backwards-compatible, so that existing METAPOST programs using `infont` and `btex ... etex` will continue to work without change. The only differences are: that the `TEX()` macro is re-implemented with internal library functions so that it no longer uses temporary files, and is therefore very much faster; and it is easier to integrate your drawings into LATEX because you no longer need to muck about with `verbatimtex` blocks. So the example code shown on the right, will produce this:

Note that `defaultfont` is still `cmr10` with the encoding that has the small stroke (that plain TEX uses for the Ł character) instead of a space, and that you can still use PostScript fonts like `phvr8r`. But also notice that the `TEX()` macro and the `btex ... etex` construction have picked up the font set by the LATEX wrapper. As you can see they produce exactly the same output; `TEX()` is generally more useful because you can pass a primary string variable as an argument, which makes it easier to construct dynamic labels. `TEX()` also has the synonym `textext()` for compatibility with ConTexT. You can use either name, as you prefer.
But this isn't the clever bit…

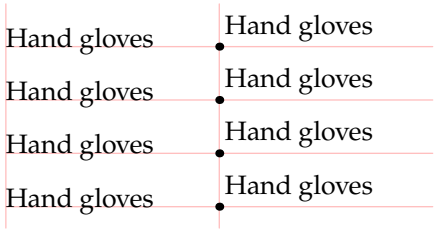
```
\documentclass[border=5mm]{standalone}
\usepackage{fontspec}
\setmainfont{TeX Gyre Pagella} % <-- note chosen font
\usepackage{luamplib}
\begin{document}
\begin{mplibcode}
beginfig(1);
  for x = 0 upto 1:
    draw (80x,16) -- (80x, -68) withcolor 3/4[red, white];
  endfor
  for y = 0 upto 3:
    draw (0, -20y) -- (160, -20y) withcolor 3/4[red, white];
  endfor

  string s; s = "Hand gloves";
  draw s infont defaultfont  shifted (0, 0);
  draw s infont "phvr8r"     shifted (0, -20);
  draw TEX(s)                shifted (0, -40);
  draw btex Hand gloves etex shifted (0, -60);

  dotlabel.urt(s,                     (80, 0));
  dotlabel.urt(s infont "phvr8r",     (80, -20));
  dotlabel.urt(TEX(s),                (80, -40));
  dotlabel.urt(btex Hand gloves etex, (80, -60));
endfig;
\end{mplibcode}
\end{document}
```

12.1 The magic of the `textextlabel` option

The clever bit is that `luamplib` allows us to turn on the `TEX()` behaviour by default, so that you can just use plain strings with the `label()` macro, and have them automatically processed through LaTeX. All you have to do is add this to the preamble:

```
\mplibtextextlabel{enable}
```

If you add this line to the example from the previous page, you get this output:



As you can see, they all come out the same. When the magic option is enabled, `luamplib` redefines the primitive binary operator `infont`. Ordinarily, this command takes two strings (the text you want to show, and the name of the font to use) and produces a picture object consisting of the text typeset in the given font.

⟨string⟩ `infont` ⟨string⟩ ⟶ ⟨picture⟩

With the option enabled, the right hand ⟨string⟩ argument (which names the font) is completely ignored, and the left hand ⟨string⟩ argument (the text to show) is passed to the `TEX()` macro. The result is still a ⟨picture⟩ of course, but instead of a simple rendering in a single font, the string will have been passed through LaTeX, so it can include maths, bold text, or any arbitrary typesetting constructions.

Note that even with the option enabled, METAPOST will not let you pass a ⟨string⟩ to `draw`. You have to put `infont "somefont"` after the string to get the magic to work; the nice thing is that the `label()` macros do this for you.

If you experiment a bit, you will find that even though the font name argument is completely ignored, you can't leave it out; you have to give at least an empty string: `draw "my text" infont ""`. However if you find yourself writing this, you probably should try `draw TEX("my text")` instead.

```
    \documentclass[border=5mm]{standalone}
    \usepackage{fontspec}
    \setmainfont{TeX Gyre Pagella}
    \usepackage{luamplib}
    \mplibtextextlabel{enable}  % <--- added option
    \begin{document}
    \begin{mplibcode}
beginfig(1);
  for x = 0 upto 1:
    draw (80x,16) -- (80x, -68) withcolor 3/4[red, white];
  endfor
  for y = 0 upto 3:
    draw (0, -20y) -- (160, -20y) withcolor 3/4[red, white];
  endfor

  string s; s = "Hand gloves";
  draw s infont defaultfont  shifted (0, 0);
  draw s infont "phvr8r"     shifted (0, -20);
  draw TEX(s)                shifted (0, -40);
  draw btex Hand gloves etex shifted (0, -60);

  dotlabel.urt(s,                      (80, 0));
  dotlabel.urt(s infont "phvr8r",      (80, -20));
  dotlabel.urt(TEX(s),                 (80, -40));
  dotlabel.urt(btex Hand gloves etex, (80, -60));
endfig;
    \end{mplibcode}
    \end{document}
```

⋆ All the examples in the rest of this section ⋆
assume that you have set `\mplibtextextlabel`

12.2 Using Unicode and matching style with OTF fonts

If you read `texdoc luamplib` carefully, you will see that you *can* use all these new facilities with plain LuaTEX, but this chapter is about using them with LuaLATEX, and in particular it assumes some familiarity with the packages `fontspec` and `unicode-math` that provide complete support for Unicode and OTF fonts; you need this familiarity in order to use `luamplib` properly.

You also need an editor that will handle Unicode. METAPOST still restricts you to using printable ASCII in your source code, but you can put whatever you want inside a string literal or a `btex ... etex` picture literal. So it becomes very easy to produce this sort of label:

```
label("café noir £2.50", origin);
```

café noir £2.50

or even whole paragraphs that use Unicode:

```
label(btex \vbox{\hsize 4in
 Nous étions à l'Étude, quand le Proviseur entra, suivi d'un
 \textit{nouveau} habillé en bourgeois et d'un garçon de classe
 qui portait un grand pupitre. Ceux qui dormaient se réveillèrent,
 et chacun se leva comme surpris dans son travail.
\par} etex, 60 down);
```

Nous étions à l'Étude, quand le Proviseur entra, suivi d'un *nouveau* habillé en bourgeois et d'un garçon de classe qui portait un grand pupitre. Ceux qui dormaient se réveillèrent, et chacun se leva comme surpris dans son travail.

But you also need a font that actually supports the Unicode characters you use. The default Latin Modern font used by LuaLATEX has a good range for English and most European languages, but is a bit lacking in (say) polytonic Greek. So you will need to define a suitable font in your preamble, and turn it on in your labels.

```
\usepackage{fontspec} \newfontface\polytonic{GFS Porson}
```

then a box like this with proper polytonic Homeric Greek source

```
label(btex \vbox{\polytonic\halign{#\hfil\cr
   ...

    (polytonic greek source in UTF8 that won't show up in
    the Latin Modern Typewriter font being used here)

   ... \cr}} etex, 120 down);
```

μῆνιν ἄειδε θεὰ Πηληϊάδεω Ἀχιλῆος
οὐλομένην, ἣ μυρί' Ἀχαιοῖς ἄλγε' ἔθηκε,
πολλὰς δ' ἰφθίμους ψυχὰς Ἄϊδι προΐαψεν
ἡρώων, αὐτοὺς δὲ ἑλώρια τεῦχε κύνεσσιν
οἰωνοῖσί τε πᾶσι, Διὸς δ' ἐτελείετο βουλή,
ἐξ οὗ δὴ τὰ πρῶτα διαστήτην ἐρίσαντε
Ἀτρεΐδης τε ἄναξ ἀνδρῶν καὶ δῖος Ἀχιλλεύς.

will produce the first few lines of the Iliad (just in case you wanted them). Essentially if you can produce something in LATEX, you can produce exactly the same in METAPOST using `luamplib`.

12.3 Multi-line labels

It is a rule of syntax in METAPOST that a string token has to be given all on one line. So if you have very long labels, or paragraphs of text, then you have to split them up into separate shorter string tokens:

```
label("\vbox{\hsize 4in It is a truth universally acknowledged,"
     & " that a single man in possession of a good fortune,"
     & " must be in want of a wife.\par}", origin);
```

taking care to include the necessary spaces, which can get fiddly.

But this is where the `btex ... etex` construction comes into play, even with `luamplib`. As we saw in the preceding section the construction fits into the METAPOST syntax scheme as a special pair of quotation marks that produces a ⟨picture⟩. Unlike regular string token, a `btex ... etex` picture token can span several lines of source code, so you can (more easily) write long TEX labels like this:

```
label(btex \vbox{\hsize 4in\noindent
     \textsc{It is a truth} universally acknowledged,
     that a single man in possession of a good fortune,
     must be in want of a wife.\par} etex, 128 down);
```

Thanks to the backward compatibility of the implementation, this works very well even when you have `mplibtextextlabel` enabled.

It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.

You also have comprehensive access to your LATEX environment, so with `luamplib` you can get tables in in METAPOST using environments like `tabular`:

```
label(btex \begin{tabular}{c}
     A way to get simple\\
     two line labels
     \end{tabular} etex, 270 down);
```

But keep in mind that whatever you ask the `TEX()` macro to typeset is going into a restricted horizontal mode box; so don't try to use floating environments like `table` or `figure`. And if you want automatic paragraph wrapping, you will have to wrap your text in a suitable `\vbox`, as shown above.

A way to get simple
two line labels

12.4 Display maths

Because the `TEX()` macro typesets everything in restricted horizontal mode, you cannot use `$$ .. $$` to create display maths directly. This is not a TEX *v* LATEX issue, it is just that for compatibility with plain METAPOST (and common sense), the designers of `luamplib` chose to typeset labels into horizontal-mode boxes This is usually what you want. If you prefer large integral operators (etc) in your labels, then you should either add `\displaystyle` at the beginning of your formula ⟶ or wrap the formula in a `\vbox` with a suitable `\hsize`. Using `\displaystyle` is probably simpler.

```
...
label("$\displaystyle \int_0^t 3x^2\, dx$", z0);
...
label("\vbox{\hsize 2in $$\int_0^t 3x^2\, dx$$}", z1);
...
```

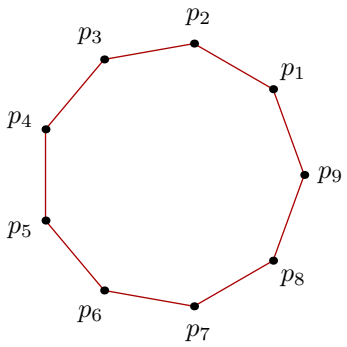
12.5 Typographical minus signs and other dynamic labels

This is really easy with `mplibtextextlabel` enabled, because we can assemble a string on the fly using standard METAPOST syntax:

```
draw (left--right) scaled 2in withcolor 2/3 red;
for i=-4 upto 4:
  dotlabel.bot("$" & decimal i & "$", (32i, 0));
endfor
```

The normal operator precedence rules ensure that the string argument to `dotlabel` is assembled before it is passed to the `TEX()` macro. The individual parts of the string you assemble do not have to be valid bits of TEX in themselves; they only have to make sense once they are actually passed to the macro. With `luamplib` there are no slow external files being used, so the complexities used above [§11.2.5] to label points around a circle can be simplified without sacrificing speed:

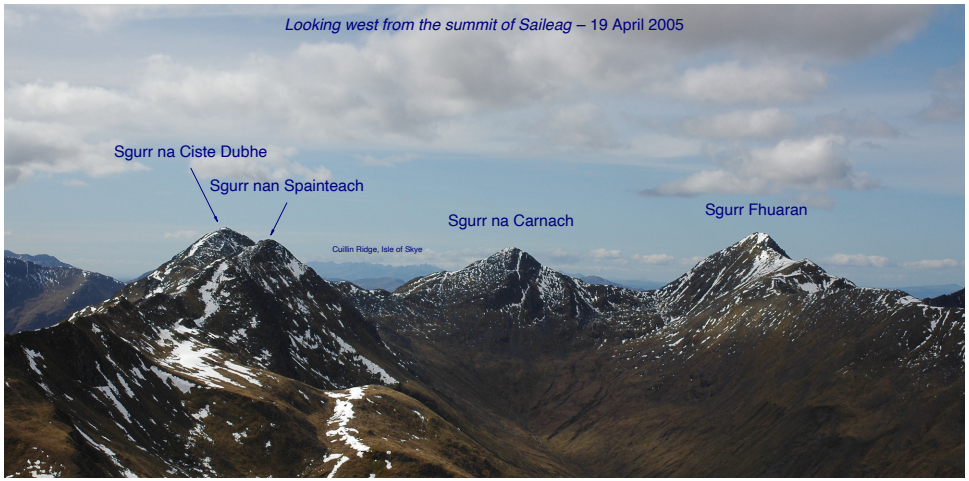
```
path P; P = for i=0 upto 8: 50 dir 40i -- endfor cycle;
draw P withcolor 2/3 red;
for i=1 upto length P:
    draw point i of P withpen pencircle scaled dotlabeldiam;
    label("$p_{" & decimal i & "}$", point i of P scaled 1.2);
endfor
```



❖ Note that you can't do this string concatenation with `btex ... etex`; although these operators might appear to be special quotation marks, they produce ⟨picture⟩ values, and in this context `&` only works with ⟨string⟩ values.

12.6 Drawing on an external image

A limitation of plain `mpost`'s use of TEX is that `\special` commands are removed from the `.dvi` file that is made into a ⟨picture⟩ variable. So, in particular, you can't use `\includegraphics` in a TEX label. But `luamplib` removes this limitation, so you can annotate images using the full array of METAPOST tools, as shown here ⟶

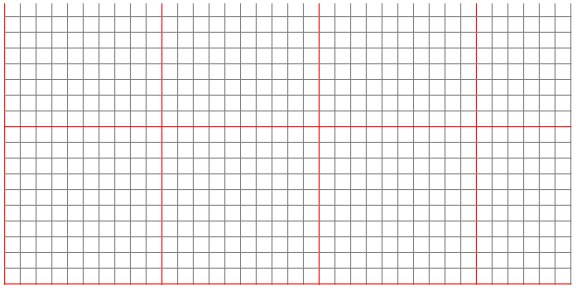


Here is a general purpose reference grid routine:

```
begingroup;
 save llx, lly, urx, ury, u; u = 10;
 (llx, lly) = llcorner currentpicture;
 (urx, ury) = urcorner currentpicture;
 drawoptions(withpen pencircle scaled 1/4);
  for x = ceiling (llx / u) upto floor (urx / u):
   draw (x*u, lly) -- (x*u, ury) withcolor if x mod u = 0: red else: 1/2 fi;
  endfor
  for y = ceiling (lly / u) upto floor (ury / u):
   draw (llx, y*u) -- (urx, y*u) withcolor if y mod u = 0: red else: 1/2 fi;
  endfor
 drawoptions();
endgroup;
```

```
\documentclass[border=1mm]{standalone}
\usepackage{luamplib}
\usepackage{graphicx}
\usepackage{fontspec}\setmainfont[Scale=0.6]{Helvetica}
\mplibtextextlabel{enable}
\begin{document}
\begin{mplibcode}
beginfig(1);
draw btex \includegraphics[width=5in]{glenshiel.jpg} etex;
% input neo-reference-grid
vardef callout@#(expr t, p, o) =
    save T; picture T; T = thelabel.@#(t, p+o);
    draw T; drawarrow p+o -- p cutbefore bbox T;
enddef;
ahangle := 20; ahlength := 2;
drawoptions(withpen pencircle scaled 1/4 withcolor 1/2 blue);
callout.top("Sgurr na Ciste Dubhe", (80, 96), (-10, 20));
callout.top("Sgurr nan Spainteach", (100, 91), (6, 12));
label.top("\tiny Cuillin Ridge, Isle of Skye", (140, 81));
label.top("Sgurr na Carnach", (190, 90));
label.top("Sgurr Fhuaran", (282, 94));
label.bot("\textit{Looking west from the summit of Saileag} - 19 April 2005",
  point 5/2 of bbox currentpicture shifted 4 down);
endfig;
```

If you uncomment the `% input neo-reference-grid` line, so that the code on ⟵ the left is included, you get this automatically-sized grid superimposed:



The grid makes it easier to find the coordinates for your annotations.

12.7 Using PDF transparency

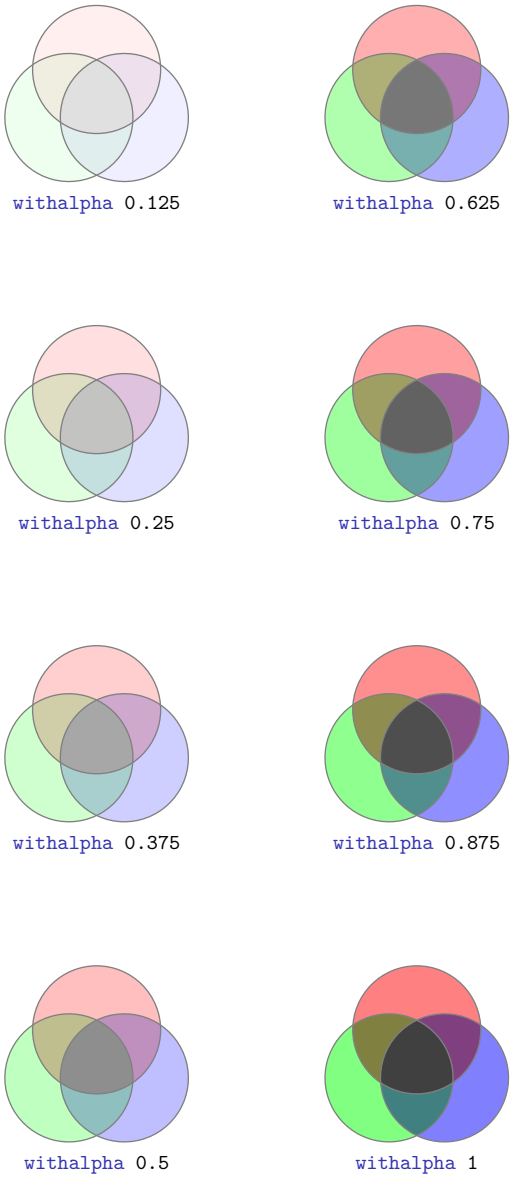
Another limitation of the plain `mpost` compiler is that it does not support any transparent colours. However, if you use the `luamplib` package with `lualatex` you get access to the PDF 1.4 transparency functions. Currently (2024) there is little documentation for this support and no 'official' macro for it, but you can add your own like this:

```
def withalpha expr a =
    withprescript "tr_alternative=2"
    withprescript "tr_transparency=" & decimal a
enddef;
```

and then use it like this, adding `withalpha` after the colour specification:

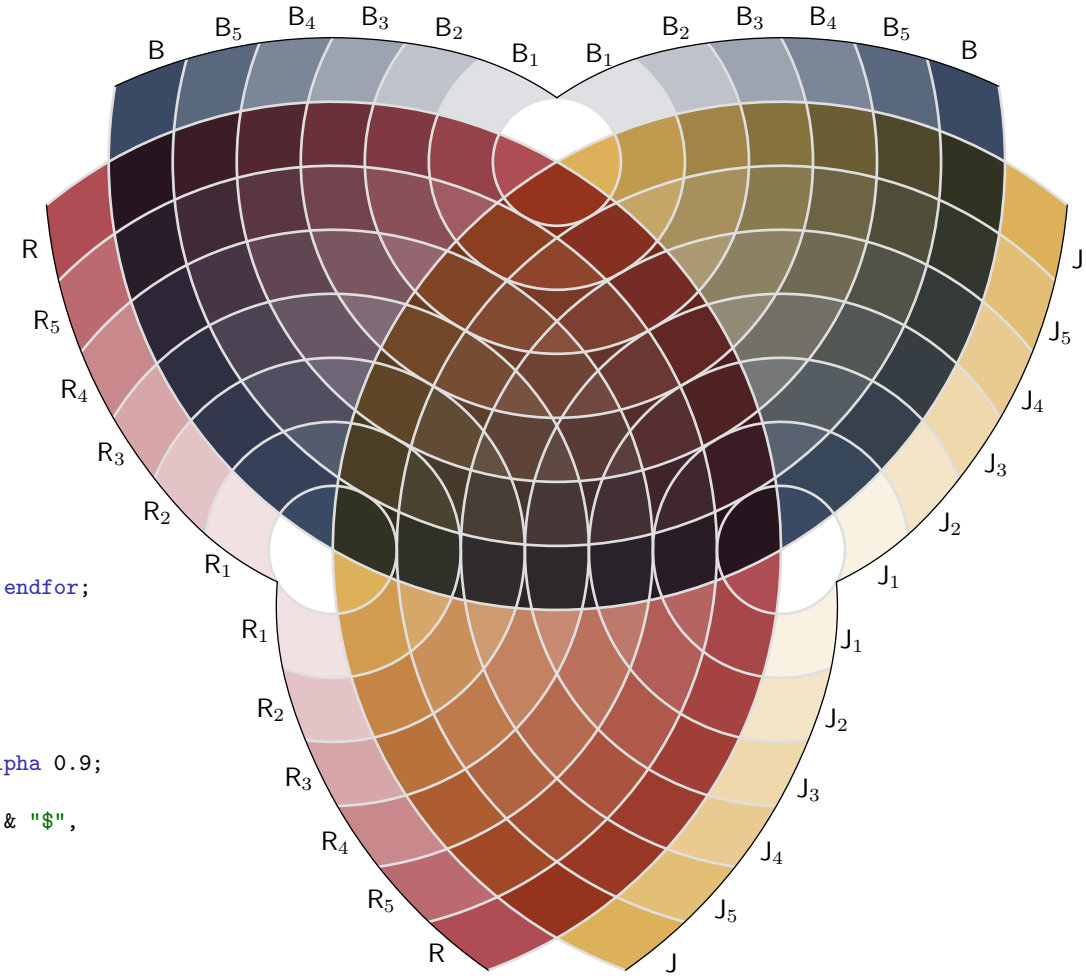
```
path r, g, b;
r = fullcircle scaled 40 shifted 10 up;
g = r rotated 120; b = g rotated 120;

numeric a; a = 0.5;
fill r withcolor 1/2[white, red] withalpha a;
fill g withcolor 1/2[white, green] withalpha a;
fill b withcolor 1/2[white, blue] withalpha a;
draw r withcolor 1/2;
draw g withcolor 1/2;
draw b withcolor 1/2;
```

The effect of changing the alpha value is shown on the right. The magic variable names `tr_transparency` and `tr_alternative` are only understood by the `luamplib` code, plain `mpost` simply ignores them. *Note that since these are not documented anywhere except in the source code, they might change in future.* You can see from the figure that `tr_transparency` controls the alpha value, but the other variable is slightly more mysterious — `tr_alternative` appears to control the PDF blending mode. A value of 1 seems to apply PDF 'normal' mode, which makes colours completely opaque with when alpha is 1; a value of 2, as used here, seems to apply PDF 'multiply' mode which blends all colours evenly, so that the order that you fill overlaps does not matter. This mode works well with slightly lighter colours.



Here is a slightly more ambitious example, using the same withalpha macro.

```
def withalpha expr a =
  withprescript "tr_alternative=2"
  withprescript "tr_transparency=" & decimal a
enddef;
beginfig(1);
  numeric u; u = 24; color shade[]; string name[];
  shade0 = 1/256(166, 57, 65);   name0 = "R"; % rouge
  shade1 = 1/256(219, 170, 72);  name1 = "J"; % jaune
  shade2 = 1/256(36, 54, 84);    name2 = "B"; % bleu
  path edge[], arc[], ring[];
  for r=1 upto 8:
    edge[r] = fullcircle rotated 90 scaled (2r * u)
    shifted (0, 7u / sqrt 3) rotated 120;
  endfor
  for i=2 upto 8:
    edge[i] := edge[i] cutbefore edge8 rotated 120
                       cutafter edge8 rotated -120;
  endfor
  arc0 = point 0 of edge1 for i=2 upto 7: .. point 0 of edge[i] endfor;
  arc1 = point 0 of edge1 for i=2 upto 7: .. point infinity of edge[i] endfor;
  for i=1 upto 7:
    ring[i] = edge[i] -- subpath (i-1, i) of arc1 --
    reverse edge[i+1] -- subpath (i, i-1) of arc0 -- cycle;
  endfor
  for r=0, 1, 2: for i=1 upto 6:  % fills first
      fill ring[i] rotated 120r withcolor (i/6)[white, shade[r]] withalpha 0.9;
      for k=0, 1:
        label("$\textsf{" & name[r] & "}" if i<6: & "_" & decimal i fi & "$",
          point i-1/2 of arc[k] shifted (-8,-2) rotated 120r);
       endfor
  endfor endfor
  for r=0, 1, 2: % now lines on top
    for i=1 upto 7:
      draw edge[i] rotated 120r withpen pencircle scaled 1 withcolor 7/8;
    endfor
    draw arc0 rotated 120r; draw arc1 rotated 120r;
  endfor
endfig;
```



13 Working with pictures

METAPOST inherits the mechanism of ⟨picture⟩ variables directly from METAFONT, except that the contents of these variables are a bit more complex. The system keeps track of the active picture in a variable called `currentpicture`, which can be copied to your own variables, or manipulated in various useful ways. In METAFONT the contents of the variable is a pattern of pixels for a font, in METAPOST the contents are vector graphics commands. This section reviews some of the things you can do with a ⟨picture⟩ variable — including putting one in a frame (see §13.11) like so ⟶

Plain METAPOST provides two built-in ⟨picture⟩ variables: `nullpicture`, which is empty, and `currentpicture`, which accumulates the results of drawing commands. When you compile your program, the `beginfig` macro will set `currentpicture` to blank, and the `endfig` macro will make METAPOST write the accumulated contents of the picture to an output file, usually as PostScript. You can also do these things yourself at any point in a program, using these macros from `plain.mp`:

```
def clearit = currentpicture := nullpicture enddef;
def shipit = shipout currentpicture enddef;
```

When you are creating a diagram with several independent elements, it is often helpful to save the `currentpicture` in a ⟨picture⟩ variable and start again. In fact it is so useful that plain METAPOST includes an `image` macro that uses the magic of macro grouping to make the process a bit easier.

```
vardef image(text t) =
  save currentpicture;
  picture currentpicture;
  currentpicture := nullpicture;
  t; currentpicture
enddef;
```

The general idea is that you declare a variable and then save a drawing into it:

```
picture P; P = image(...);
```

and then you have a picture element that can be manipulated or copied as needed. The rows of decorative beads in the frame on the right were created like this.

13.1 Creating and transforming pictures

After you have declared a variable with `picture P;` you can give it some contents in a number of ways:

- `P = nullpicture;` — this makes P empty.
- `P = currentpicture;` — save a copy of your current picture (if any).
- `P = image(... MP tokens ...);` — capture some drawing commands.
- `P = "string" infont "font-name";` — capture an image of `string` set in the given font.
- `P = btex ... TeX tokens ... etex;` — capture the result of passing some arbitrary tokens through TEX.
- `P = TEX("string");` — capture the result of passing some arbitrary string of tokens through TEX, using the `TEX` macro.

You can read more about the details of type setting in §11 and §12, but the point here is that the results are normal ⟨picture⟩ variables that you can manipulate and use like any other. You can apply any of the normal METAPOST transformations to a picture, so it can be slanted, scaled, rotated, or shifted like any ⟨pair⟩ or ⟨path⟩. Each picture has a reference point that is the position of the origin for pictures created with `image` or by saving `currentpicture` directly, and is usually the bottom left-hand corner of a typeset picture created by TEX. So to add three copies of P to your current picture, you could do:

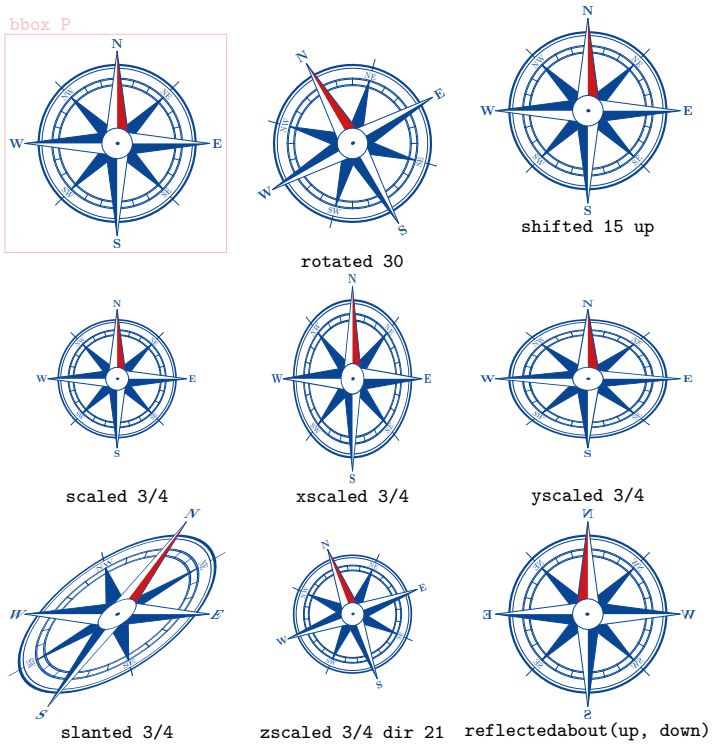
```
for i=1 upto 3: draw P shifted (20i, 0); endfor
```

and METAPOST will add copies of P with the reference points shifted to $(20, 0)$, $(40, 0)$, and $(60, 0)$. A selection of other transformations is shown on the right ⟶

If you need to measure the size of your picture, you can get the coordinates of the corners with the built-in corner commands, and do some arithmetic like this:

```
(wd, ht) = urcorner P - llcorner P;
```

You also get `ulcorner`, `lrcorner`, and `center`; plus `bbox` which returns the rectangular path round the four corners, expanded by the current value of `bboxmargin`. (See also §13.3).



The reference point for each compass is the small dot in the middle.

13.2 Clipping and bounding boxes

Once you have got your ⟨picture⟩ variable, and possibly transformed it, the main thing you can do with it is to use `draw` to add it to the current picture. But there are two other commands that are sometimes helpful that allow you to alter the apparent size of the picture.

- **setbounds** ⟨picture⟩ **to** ⟨path expression⟩
- **clip** ⟨picture⟩ **to** ⟨path expression⟩

Both commands set the boundary of your picture to the arbitrary path expression, and then the `clip` command also erases all of the picture that lies outside the boundary. (Note that this is not the same as setting the bounding box. The arbitrary path does not have to be a rectangle; after either of these commands the bounding box will be the rectangle that fits around the arbitrary path).

METAPOST inherits the `clip` command from PostScript; there is no equivalent in METAFONT. It can be useful as an alternative to `buildcycle`, but it is most commonly used for trimming a repeating pattern to a particular shape. The usual approach is to define a particular shape, *s*, then draw your pattern over a large area that covers the shape, and finally call `clip currentpicture to s` to trim the pattern to the shape. This is best done after you define all your shapes, but before you draw any others or add any labels. The example on the right shows this approach in action.

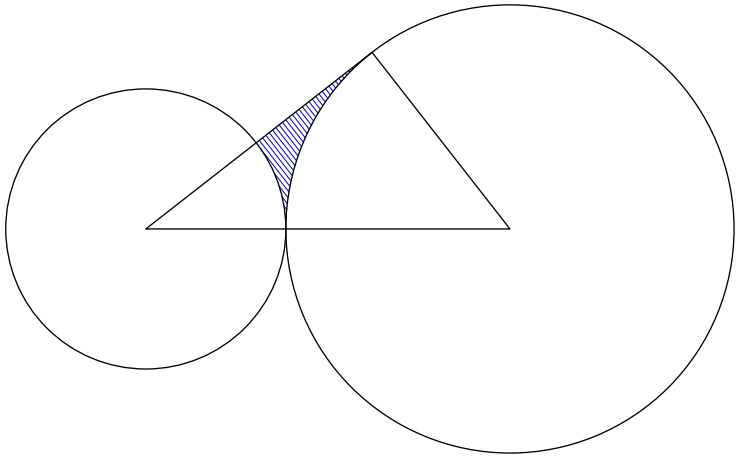
You can apply `clip` to any picture, so you might prefer to capture your pattern in ⟨picture⟩ variable with `image`, apply `clip` to that, and then `draw`. This works nicely if you want to repeat the clipped image.

You can also use this technique to fill with a gradient: just reduce the gap between each line and use the index variable to blend between two colours. Something like `withcolor (i/r)[blue, white]` in the example shown.

☞ Why would you ever want to use `setbounds`? Mainly to help with aligning type set labels, as discussed in §11.1.6, or if you want to make boundaries of different picture elements consistent in order to line them up more easily. Or perhaps to set a margin for the whole image by using something like this just before the `endfig`.

```
setbounds currentpicture to bbox currentpicture;
```

This has the effect of adding a `bboxmargin` wide strip all round.



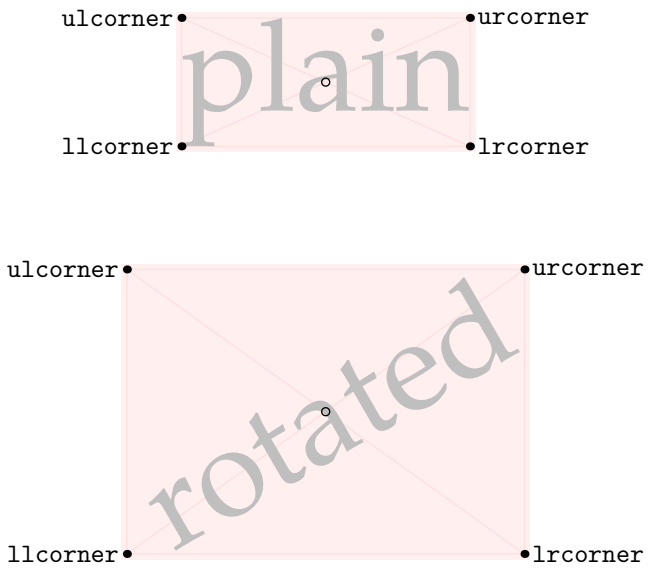
```
beginfig(1);
  numeric r; r = 42; z1 = 5/4 r * left; z2 = 2r * right;
  path c[];
  c1 = fullcircle scaled 2 abs z1 shifted z1;
  c2 = fullcircle scaled 2 abs z2 shifted z2;
  c3 = fullcircle scaled abs(z2-z1) shifted 1/2[z1,z2];
  numeric t, u;
  (t, whatever) = c2 intersectiontimes c3;
  (u, whatever) = c1 intersectiontimes (point t of c2 -- z1);

  path s;
  s = subpath (0, u) of c1 -- subpath (t, 4) of c2 -- cycle;
  numeric gap; gap = 2;
  for i=0 upto 2r / gap:
    draw (origin--right) scaled 2r rotated 45t shifted (gap*i,0)
      withpen pencircle scaled 1/4 withcolor 2/3 blue;
  endfor
  clip currentpicture to s;

  draw c1; draw c2;
  draw z1 -- z2 -- point t of c2 -- cycle;
endfig;
```

13.3 Bounding boxes of transformed pictures

When you rotate a text label, or otherwise transform a picture, the corner-points also change, but not quite in the way you might think. It turns out that the bbox is always a rectangle aligned to the edges of your page. Effectively, the corners are determined *after* any transformation, and the center is strictly the intersection of the lines between opposite corners.

You will notice this if you use the technique given on p. 29 of the METAPOST manual to draw a label on a coloured (or erased) background; if you have rotated the label, the bbox may be larger than you want. One solution is to define your label untransformed and then apply the transformation twice: first when you fill the bounding box and again when you draw the label, for example:

```
picture p; p = thelabel.top("Correctly", origin);
unfill bbox p rotated 30 shifted z0;
draw p rotated 30 shifted z0;
```



13.4 Using pictures to assemble a complex diagram

If you have a diagram with several independent parts, like the comparison above then there is a useful general technique: declare a subscripted ⟨picture⟩ variable, and then use image to draw each part separately. The advantage of this is that you do not have to worry about where the origin is, which often makes a drawing simpler (for example because you can use rotated rather than rotatedaround). Once you have created all the parts you can then add them to the final image using draw as shown on the right ⟶

Sometimes it is more convenient to use label to place the pictures, taking advantage of the automatic alignment provided. Note also that, unless you have explicitly or implicitly filled them with the background colour, the blank parts of each picture are really transparent so you can overlap them when appropriate.

The illustration above was drawn using this general sub-picture technique, approximately like this:

```
picture P[];
P1 = image(
    % first drawing...
);
P2 = image(
    % second drawing...
);
draw P1 shifted 100 up; draw P2 shifted 100 down;
```

13.5 Adding a caption to the current picture

When you have finished a complicated picture, you may want to add a caption or some other label which would look neat if it were exactly centred at the bottom of the everything else. You could keep track of exactly how wide and deep you have made the picture to do this, but there is an easier way, that will adjust itself automatically if you change the contents of the picture later.

```
beginfig(1);
% ... complete drawing that needs a caption ...
label.bot("This picture needs a label at the bottom",
                            point 1/2 of bbox currentpicture);
endfig;
```

Through the automatic alignment routines in `label`, this will produce a label neatly centred at the bottom. If it is too close you can either set a larger *bboxmargin* or use something like:

```
label.bot("This picture needs a label at the bottom",
              point 1/2 of bbox currentpicture shifted 42 down);
```

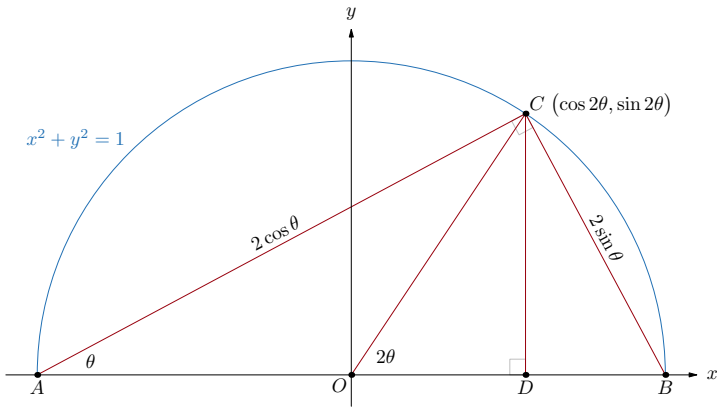
Note that in both cases the addition of the label will move the corner points of the picture, so that the bounding box will have been expanded to include the new label. You can use this feature to add a series of centered labels. But if this is not what you want, perhaps because you want to add two labels side by side at bottom, then you can "freeze" the current bounding box like this:

```
picture bb; bb = bbox currentpicture;
label.bot("Left label", point 1/4 of bb);
label.bot("Right label", point 3/4 of bb);
```

The path returned by `bbox` has four points starting at the lower left and proceeding clockwise like a `unitsquare`. So `point 1/2 of bbox currentpicture` is half way between lower left and lower right, while `point 5/2 of bbox currentpicture` is half-way from upper right to upper left.

Note that the path is defined even if the current picture is empty. If you call `bbox currentpicture` at the start of a picture you will get a square path centered on the origin and scaled to 2 *bboxmargin*.

Here is an example.



$$\triangle ACD \sim \triangle ABC$$

$$CD\big/AC = BC\big/AB \qquad AD\big/AC = AC\big/AB$$

$$\sin 2\theta\big/2\cos\theta = 2\sin\theta\big/2 \qquad \big(1+\cos 2\theta\big)\big/2\cos\theta = 2\cos\theta\big/2$$

$$\sin 2\theta = 2\sin\theta\cos\theta \qquad \cos 2\theta = 2\cos^2\theta - 1$$

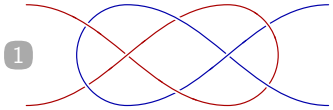
The labels at the bottom were added like this:

```
label.bot("$\triangle ACD \sim \triangle ABC$",
                point 1/2 of bbox currentpicture shifted 24 down);

path bb; bb = bbox currentpicture shifted 12 down;
label.bot(btex \vbox{....} etex, point 1/4 of bb);
label.bot(btex \vbox{....} etex, point 3/4 of bb);
```

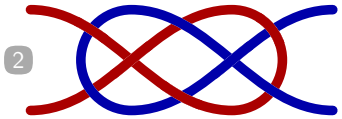
The second call to `bbox currentpicture` gets the bounding box that includes the first centered label.

13.6 Drawing pictures with various colours and pens

Consider the ⟨picture⟩ with different colours and pens in the example here ⟶ With this captured in a ⟨picture⟩ variable, you can `draw` it with different colours and pens to obtain a variety of effects:

1 

```
draw cb;
```

2 

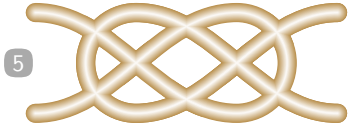
```
draw cb withpen pencircle scaled 4;
```

3 

```
draw cb withcolor 1/2;
```

4 

```
draw cb withpen pencircle scaled 4 withcolor 1/2;
```

5 

```
for i=8 step -1/2 until 1/2:
  draw cb withpen pencircle scaled i
    withcolor sqrt(i/8)[white, pretzel];
endfor
```

6 

```
for e within cb:
  draw pathpart e withpen penpart e scaled 4
    withcolor 7/8[colorpart e, background];
endfor
```

The picture is supposed to represent a fancy knot (a “Carrick bend”), and to show the red and blue strands crossing each other. The `overdraw` macro tries to do this by using `undraw` with a thick pen, then drawing the upper strand on top.

```
numeric s; s = 21;
path alpha;
alpha = ((-2s, s) {right}
.. halfcircle rotated -90 scaled 2s shifted (2s, 0)
.. {left} (-2s, -s)) shifted (s*left);

vardef overdraw(expr a, b, r, P, shade) =
  linecap := butt;
  undraw subpath (a+r, b-r) of P withpen pencircle scaled 2;
  draw subpath (a, b) of P withcolor shade;
enddef;

picture cb; cb = image(
  draw alpha withcolor 2/3 red;
  undraw alpha rotated 180 withpen pencircle scaled 2;
  draw alpha rotated 180 withcolor 2/3 blue;
  overdraw(0.21, 0.36, 0.02, alpha, 2/3 red);
  overdraw(0.67, 0.86, 0.02, alpha, 2/3 red);
  overdraw(3.4,  4.3,  0.1,  alpha, 2/3 red);
  overdraw(5.4,  5.6,  0.02, alpha, 2/3 red);
  overdraw(5.4,  5.6,  0.02, alpha rotated 180, 2/3 blue);
);
```

- Example 1 shows that by default `draw` uses the colours and pens defined in the picture
- Examples 2, 3, and 4 show what happens if you change the pen, or the colour, or both.
- Example 5 shows you how to make a pretzel in METAPOST.
- Example 6 shows you the slightly tricky syntax to extract the paths, pens, and colours from the ⟨picture⟩ and adjust them as needed.

13.7 Simulating transparency with pictures

Filling with transparent colour can sometimes be a very effective graphic technique, but plain METAPOST provides no colour model that directly supports transparency for any output format. If you are using `mpost`, you will have to resort to layering and managing the colour blending yourself. This section presents an example of the basic technique, that could be adapted to more general purpose macros as required.

☞ But, the technique is quite laborious so you might prefer to switch to `luamplib` which provides support for the PDF transparency model, as discussed in §12.7.

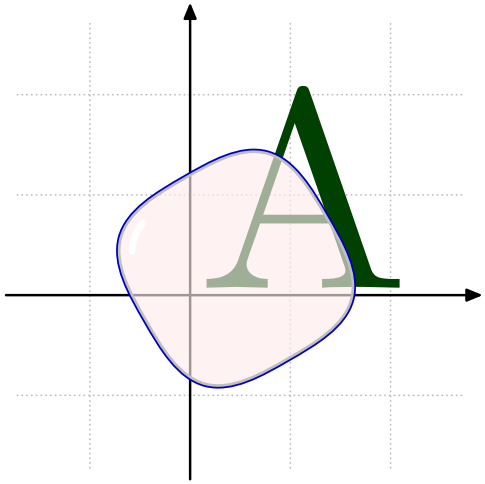
The two useful tools in the plain METAPOST kit bag are:

- The ability to loop through all the elements of a picture
- The ability to blend colours using the mediation syntax

The example drawing on the right consists of a regular grid and a text picture, with a bubble drawn over the top. The bubble can be made to look transparent like this:

1. Define the shape that you want to be transparent, decide on how opaque you want it, and the colour to use.
2. Capture the current drawing in a ⟨picture⟩ variable.
3. Loop over all the elements in that picture, redrawing each one with a blended color, and capture all this in another ⟨picture⟩.
4. Add some decoration; here there is an internal margin, and a hint of a reflection line to make it look shiny.
5. Clip the new blended-colour picture to the shape.
6. Fill the shape with the filler colour.
7. Draw the blended-colour parts on top.
8. Finally, add a neat edge (if needed).

As you may appreciate, with this approach, you need to do the transparent parts after you have drawn everything else in your drawing.



Omitting the simple grid, this drawing was produced like this:

```
% Large A
label.urt("A" infont defaultfont scaled 8, origin) withcolor 1/4 green;
% the "transparent" box
path shape; shape = (superellipse(right, up, left, down, 0.81))
  shifted 1/2 right scaled 30 rotated 30;
alpha = 5/8;  % alpha: 0=invisible, 1=opaque
color filler; filler = .95[red,white];
picture bg; bg = currentpicture;  % capture the current drawing
picture fg; fg = image(
  for e within bg:  % redraw everything with blended color
    draw e withcolor alpha[colorpart e, filler];
  endfor
  draw shape withpen pencircle scaled 2 withcolor 3/4;
  draw subpath (2.718, 3.1415) of shape  % add decoration
      shifted - center shape scaled 7/8 shifted + center shape
      withpen pencircle scaled 2 withcolor white;
);
clip fg to shape; % finally clip the fg drawing
fill shape withcolor filler;   % fill the shape
draw fg;           % and put the fg back on top
draw shape withcolor 3/4 blue;
```

13.8 Adding a background and other post-processing

The ⟨picture⟩ capture technique provides a simple way to add a background or do other post-processing on your drawing. The advantage is that you do not have to work out the size of your drawing before you start.

You could add a subtle off-white background fill like this:

```
picture P; P = currentpicture; fill bbox P withcolor (1,1,31/32); draw P;
```

Or you can be more ambitious, as shown in the example on the right ⟶
In general, you draw any background you want, like this:

```
picture P; P = currentpicture; clearit;
% do complex background drawing...
clip currentpicture to bbox P; draw P;
```

Or you can do things like make automatic adjustments to the scale. If you wanted to be sure that your drawing was not more than 5 inches wide, you could try this just before the `endfig`:

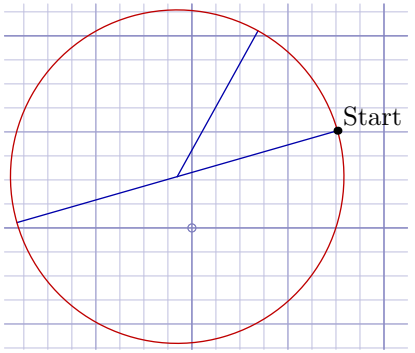
```
numeric wd; wd = xpart (urcorner currentpicture
                                        - llcorner currentpicture);
if wd > 360: currentpicture := currentpicture scaled (360/wd); fi
```

If you wanted to apply one of these changes to all the figures in your `mpost` input file then you can use the hook provided by plain METAPOST:

```
extra_endfig := "picture P; P = currentpicture; clearit;" &
                "fill bbox P withcolor (1,1,31/32); draw P;";
```

The definition of `endfig`, includes the line `scantokens extra_endfig`; so that any contents of the string variable `extra_endfig` are automatically processed before the figure is produced. If you are using `luamplib` then you can use the alternative hook that it provides so that you do not even have to type `endfig`:

```
\everyendmplib{picture P; P = currentpicture;
fill bbox P withcolor (1,1,31/32); draw P; endfig;}
```



Here is an example that adds graph paper behind a drawing. The first three lines make the example drawing, the final `input` adds the graph paper.

```
path C; C = fullcircle scaled 125 shifted 20 up rotated 16;
for t=0, 1, 4: draw center C -- point t of C withcolor 2/3 blue; endfor
draw C withcolor 3/4 red; dotlabel.urt("Start", point 0 of C);
input pics-graph-paper-inch
```

This is `pics-graph-paper-inch.mp`:

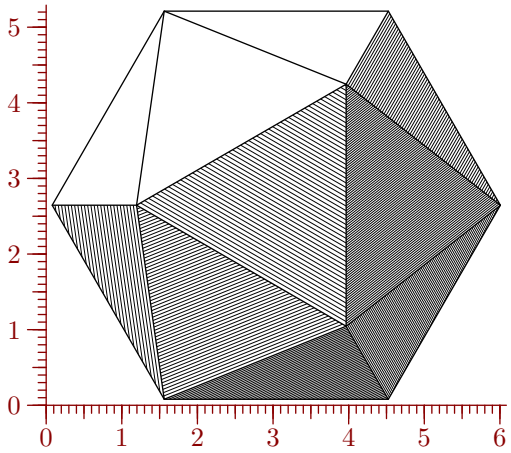
```
input automatic-grid
begingroup; save P; picture P; P = currentpicture; clearit;
draw grid(P, 9) withpen pencircle scaled 4/16 withcolor 1/16(12,12,14);
draw grid(P,36) withpen pencircle scaled 5/16 withcolor 1/16(10,10,13);
draw grid(P,72) withpen pencircle scaled 6/16 withcolor 1/16( 8, 8,12);
draw P; endgroup;
```

And this is `automatic-grid.mp`

```
vardef grid(expr p, grid_unit) =
  save llx, lly, urx, ury;
  (llx, lly) = llcorner p - (bboxmargin, bboxmargin);
  (urx, ury) = urcorner p + (bboxmargin, bboxmargin);
  image(
    for x = 1 + floor(llx / grid_unit) upto floor(urx / grid_unit):
      draw (x * grid_unit, lly) -- (x * grid_unit, ury);
    endfor
    for y = 1 + floor(lly / grid_unit) upto floor(ury / grid_unit):
      draw (llx, y * grid_unit) -- (urx, y * grid_unit);
    endfor
    if (llx < 0) and (lly < 0) and (urx > 0) and (ury > 0):
      draw fullcircle scaled 3; % show origin if in range
    fi
  )
enddef;
```

13.9 Adding a ruler

If you wish to check the dimensions of your drawing, it can be useful to add a temporary ruler that shows you the dimensions of the bounding box like this:



The red rulers were added by putting `input ruler-cm` at the end of the figure. They are drawn round the bounding box, set here with the default margin of 2 bp. The `ruler-cm.mp` file looks like this:

```
numeric u[]; u0 = 1 cm; u1 = 5 mm; u2 = 1 mm;
drawoptions(withcolor 0.54 red);
input ruler
```

and there is a companion `ruler-inch.mp` file that looks like this:

```
numeric u[]; u0 = 72; u1 = 18; u2 = 6;
drawoptions(withcolor 0.67 blue);
input ruler
```

The idea is that you set a subscripted variable `u[]` to a number of unit sizes where you want markers and then call `input ruler`.

Here is the implementation of `ruler.mp`:

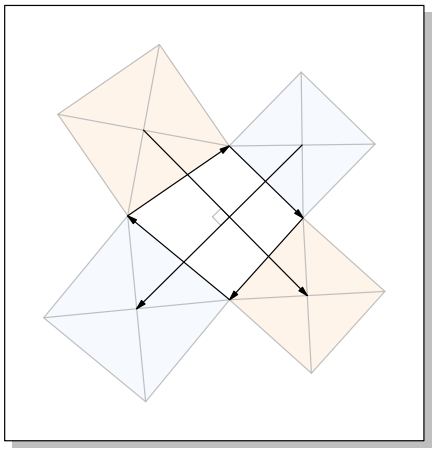
```
% add a ruler along the left hand and lower edges
% of the bounding box of the currentpicture
path B, p; pair o; B = bbox currentpicture;
for s=-1, 1:
  p := subpath (0, s) of B;
  a := arclength p;
  o := if s < 0: left else: down fi;
  for i=0 upto 3:
    exitif not known u[i];
    for j=0 upto floor(a/u[i]):
      pair t; t = point arctime j*u[i] of p of p;
      draw (origin -- (7 - 2i) * o) shifted t;
      if i=0: label(decimal j, t shifted 12 o); fi
    endfor
  endfor
  draw p;
endfor
```

The inner loop draws successively shorter lines at each of the minor units, and numbers at the major units.

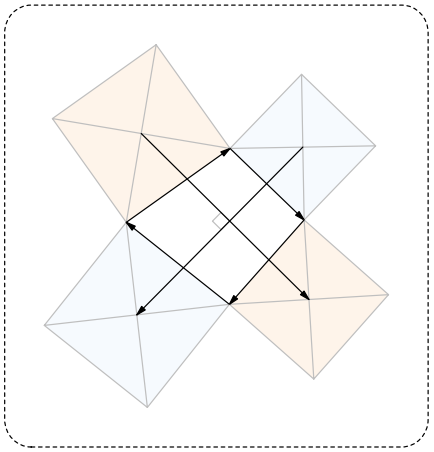
Note that this macro designed to be used a temporary input added at the bottom of a drawing to see how big it is. You would not usually leave it in place in a final drawing. This is why none of the variable names is protected. To make the macros more robust you could enclose them with `begingroup` and `endgroup`, and `save` the names, and clear `drawoptions`.

13.10 Adding a border

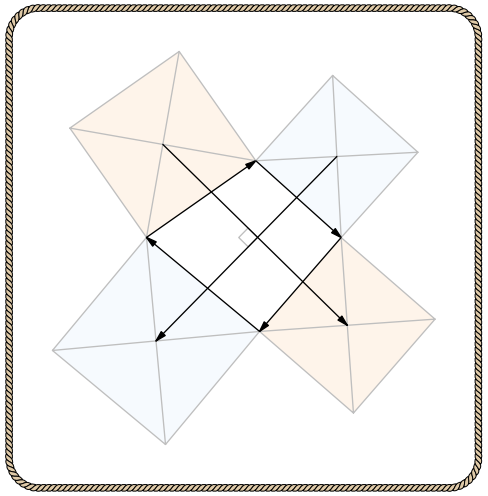
In most documents the drawings look just fine without any decoration, but sometimes you might want to add emphasis or pick out part of a drawing. The examples here can be applied to currentpicture or any other ⟨picture⟩ variable.



```
bboxmargin := 16;
picture P; P = currentpicture;
fill bbox P shifted (3,-3) withcolor 3/4;
unfill bbox P; draw bbox P; draw P;
```



```
vardef with_rounded_corners expr p =
  for i=1 upto length p:
    subpath (i-15/16, i-1/16) of p ..
  endfor cycle
enddef;

bboxmargin := 16;
draw with_rounded_corners bbox currentpicture
    dashed evenly scaled 1/2;
```



Don't take this one too seriously…

```
def twisted expr t of p =
  rotated angle direction arctime t of p of p
            shifted point arctime t of p of p
enddef;
vardef rope expr c =
  save s, w, hemp, n, a, b, A;
  color hemp; hemp = 1/256 (192, 149, 82);
  w = -1; n = -1; A = arclength c; s = A/floor(A/2);
  path a[];
  for t=0 step s until A + 1: a[incr n] =
    (0,+w) twisted t-3/2s of c .. (0,+w) twisted t-1/2s of c ..
    (0,-w) twisted t+1/2s of c .. (0,-w) twisted t+3/2s of c;
  endfor
  image(for i=1 upto n:
    path b; b = buildcycle(a[i-1], reverse a[i]);
    fill b withcolor 1/2[white, hemp];
    draw b withpen pencircle scaled 1/8;
  endfor)
enddef;

bboxmargin := 16;
draw rope with_rounded_corners bbox currentpicture;
```

13.11 Adding a frame

As promised at the start of this section, here is the code for the picture frame drawn round Raphael's young man.

```
input picture_frame
beginfig(1);
  picture F;
  F = thelabel(TEX("\includegraphics[width=200pt]{youth.jpg}"), origin);
  draw F; draw frame F;
endfig;
```

All the heavy lifting is done by `frame` macro defined in `picture_frame.mp` ⟶ This macro also needs some colours:

```
color gold, dark, grey;
gold = 1/256(243, 197, 127);
dark = 1/256(144, 87, 50);
grey = 1/256(156, 147, 138);
```

a picture of a small silvery-gold ball:

```
picture ball; ball = image(for i=0 upto 16:
  fill interpath(i/16,
    fullcircle scaled 10,
    fullcircle scaled 3 shifted (-2, 2)
  ) withcolor (i/16)[gold, 15/16 white];
endfor) scaled 1/4;
```

and an internal variable that defines the width of the frame:

```
newinternal pf_width; pf_width := 21;
```

The macro takes a path or picture P as an argument, and makes a thin rectangle f that is scaled to the desired width and the longer of the two sides of the bounding box. This thin rectangle is then decorated with background colour, strips of colour to suggest depth, a random spatter-pattern, and a row of little balls. The macro then makes two trapezium shaped copies of the decorated rectangle, pieces them together around P, and returns the result as a ⟨picture⟩.

```
vardef frame expr P =
  save base, side, f, t, u, xx;
  picture base, side; path f; numeric t, u, xx;
  t = arclength subpath (0,1) of bbox P;
  u = arclength subpath (1,2) of bbox P;
  xx = max(t, u) + 2 pf_width;
  f = unitsquare xscaled xx yscaled pf_width;
  % convenience / nonce function
  vardef paint_strip(expr y, wd, shade) =
    draw subpath (0, 1) of f
      shifted (0, if y < 0: pf_width + fi y)
      withpen pencircle scaled wd
      withcolor shade
  enddef;
  base = image(
    fill f withcolor gold; % background colour
    paint_strip(2,    3,   5/4 grey);  % grey strips
    paint_strip(3.5,  1/4, grey);
    paint_strip(5,    1/4, 1/2[gold, dark]);
    paint_strip(-6.5, 1/4, 1/2[gold, dark]);
    paint_strip(-6,   1/4, 1/2[gold, dark]);
    paint_strip(-2,   2,   5/4 grey);
    % spatter with random spots
    for i=0 upto 4 * arclength(subpath (0,1) of f):
      fill fullcircle scaled uniformdeviate 3/4
        shifted (uniformdeviate xx, uniformdeviate pf_width)
        withcolor dark;
    endfor
    % decorative balls
    for x = 2 step 3 until xx:
      draw ball shifted (x, 2);
    endfor
  );
  % make two trapezium shapes
  side = base;
  clip side to (pf_width, 0) -- (pf_width + u, 0)
    -- (2 pf_width + u, pf_width) -- (0, pf_width) -- cycle;
  clip base to (pf_width, 0) -- (pf_width + t, 0)
    -- (2 pf_width + t, pf_width) -- (0, pf_width) -- cycle;
  % arrange the pieces into a square
  image(
    draw base rotated 180 shifted point 1 of bbox P shifted (+pf_width, 0);
    draw base rotated   0 shifted point 3 of bbox P shifted (-pf_width, 0);
    draw side rotated  90 shifted point 0 of bbox P shifted (0, -pf_width);
    draw side rotated 270 shifted point 2 of bbox P shifted (0, +pf_width);
  )
enddef;
```

14 Drawing and decorating lines

This section is all about making marks along a ⟨path⟩ using `draw` and a number of other more sophisticated techniques.

14.1 Choosing a pen

METAPOST inherits a rather complicated system of pens from METAFONT. As explained in *The METAFONTbook*, the original intention of this system was to improve the digitisation of font characters so that low-resolution raster versions would look aesthetically pleasing. This is not really a problem for graphics that you might produce with METAPOST, and so for most purposes you can stick to the default circular pen, and not worry about using `pensquare`, `penrazor`, and `penspeck`, nor creating your own ⟨pen⟩ variables. On the other hand, you probably will need to change the size and shape of the pen occasionally, and it is good to understand the mechanisms available.

The default, built-in, pen is `pencircle` – at the start of every job this pen is scaled to 0.5 bp and saved as `currentpen` and `defaultpen`. Every time you use `draw`, METAPOST automatically adds `withpen currentpen` unless you have added your own `withpen`.

```
draw origin -- 20 right;  % uses "currentpen"
draw (0,10) -- (20,10) withpen pencircle scaled 2; % thick pen
```

If you get tired of typing `withpen`, you can change pens with the `pickup` macro; essentially this just updates the value of `currentpen`.

```
draw origin -- 42 right;  % uses "currentpen"
pickup pencircle scaled 2; % use a thick pen
draw (0,-24) -- (42,-24);
draw (0,-36) -- (42,-36);
pickup defaultpen;  % change back to default
```

If you are using a fat pen and you don't like the big rounded ends, then you can use `cutdraw` or set `linecap := butt;` — see §14.8 for more.

Here is an example that makes use of a circular pen transformed to a thin ellipse.

```
   \documentclass[border=5mm]{standalone}
   \usepackage{luamplib}
   \newcommand\fleuron{\begin{mplibcode}
beginfig(1);
  -z1 = z4 = 7 dir 8;
  y2 - y1 = y4 - y3 = 3(y4 - y1);
  z2 - z1 = z4 - z3 = whatever * dir 50;
  draw z1 .. controls z2 and z3 .. z4
    withpen pencircle xscaled 1.2 yscaled 0.2 rotated 50;
  currentpicture := currentpicture rotated - angle z4;
endfig;
   \end{mplibcode}}
   \begin{document}
   Here is a \fleuron\ mark.
   \end{document}
```

This example shows how to use an elliptical pen to draw a little twiddle mark — and incidentally how to define a LaTeX command that draws a METAPOST figure — that comes out looking like this: ~. The idea is that you could use it as a decoration.

~ *A typographical ornament* ~

With a little work, you could also use it as a fancy rule between sections.

The important parts are that `pencircle` is scaled to be wider than it is tall, and then rotated so that it is at the correct angle at the start and at the end of the path. Think of it as a calligraphic nib. For another example, look at §11.2.4, which uses a similar pen to draw the braces for the dimension label.

14.2 Multiple lines

THE PENS AVAILABLE in METAPOST are all simple convex polygons without holes, so if you want to draw double or triple lines, you have to draw the individual parts of the marks you want along the path.

The simplest approach is to draw the path with a thicker pen, then draw over it with a thinner pen using the background colour, like this:

```
path a; a = origin -- (48, 3) -- (96, -3) -- 144 right;
draw a withpen pencircle scaled 3/2;
draw a withpen pencircle scaled 1/2 withcolor background;
```

which produces this: Notice that the lines have been drawn with the default rounded ends. If you don't want this then use `cutdraw` for the background lines (or set `linecap := butt`), so that you get ends that look open, like this: For triple lines, you will need to draw three times: first broad, then medium with the background colour, then thin in the middle; and so on.

Beware that you will need to be careful if the line needs to have an arrowhead or to touch another object. Beware also that drawing with the background colour does not erase anything, it just draws over the path with an opaque line that happens to be the same colour as the background, so you will not see anything that happens to lie under the line even if you want to.

If you want to draw real separate parallel lines, then the simplest approach is just to shift the path sideways using something like this:

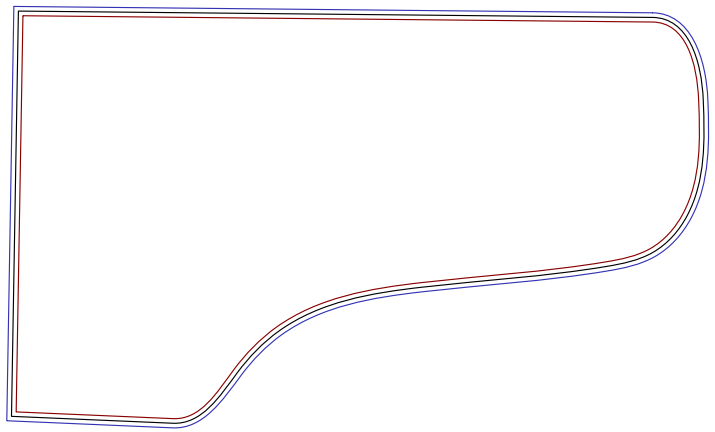
```
path p; p = origin -- 100 dir 30; draw p;
draw p shifted 4 unitvector(direction 1/2 of p rotated 90);
```

If your path is a regular cycle, like a circle or a polygon, then another approach is to draw a scaled copy.

```
path p; p = fullcircle scaled 100 shifted 42(normaldeviate, normaldeviate);
draw p; draw p shifted -center p scaled 0.98 shifted center p;
```

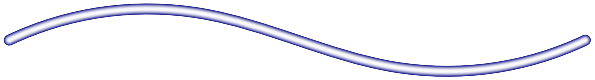
but if your path is more complex, then you need something like the macro shown at the right. The piano-shaped path p is drawn in black; the outer blue path was drawn with `draw beside(p, 2)` and the red one with `draw beside(p, -2)`.

```
vardef extended(expr p) =
  -42 unitvector(direction 0 of p) shifted point 0 of p .. p ..
  +42 unitvector(direction 1 of p) shifted point 1 of p
enddef;

vardef chopper expr t of p =
  (up--down) scaled 42
  rotated 1/2(angle direction t-eps of p + angle direction t+eps of p)
  shifted point t of p
enddef;

vardef beside(expr p, d) =
  save n, a, b, aa, bb; numeric n; n = length p;
  pair a, b, aa, bb; path _part[];
  for i = 1 upto n:
    aa := postcontrol i-1 of p - point i-1 of p;
    bb := point i of p - precontrol i of p;
    a := unitvector(aa) rotated -90 scaled d;
    b := unitvector(bb) rotated -90 scaled d;
    _part[i] = extended(
      point i-1 of p shifted a {aa} .. point i of p shifted b {bb}
    ) cutbefore chopper i-1 of p cutafter chopper i of p;
  endfor
  _part[1] for i=2 upto n: .. _part[i] endfor if cycle p: .. cycle fi
enddef;
```



Making tubes

Drawing with multiple lines can also produce a three-dimensional effect for knot diagrams (provided that you don't think of toothpaste).



This tube-like effect can be created with a loop like this:

```
path a; a = origin .. (72, 10) .. (144, -10) .. (216, 0);
for i=4 step -1/8 until 1/2:
  draw a withpen pencircle scaled i withcolor (i*i/16)[white, 1/2 blue];
endfor
```

The idea is to draw a path repeatedly with lines that get thinner and use a lighter shade Notice the colour fades from white to dark blue using the mediation syntax. You might like to experiment with linear or quadratic transitions until you get a gradient that you like. In the loop above as i varies from 4 down to $\frac{1}{2}$, the mediation fraction varies from 1 down to $\frac{1}{64}$, like this:

i :	4.00	3.75	3.50	3.25	3.00	2.75	2.50	2.25	2.00	1.75	1.50	1.25	1.00	0.75	0.50
$i^2/16$:	1.00	0.88	0.77	0.66	0.56	0.47	0.39	0.32	0.25	0.19	0.14	0.10	0.06	0.04	0.02

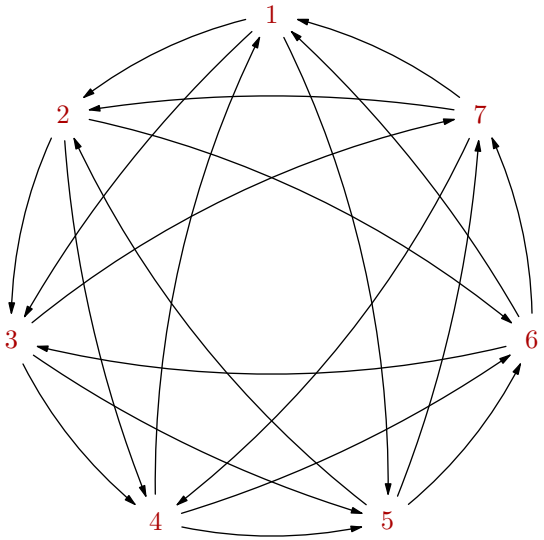
which makes the gradient steeper nearer the edges. If you don't want the rounded ends, then you should use `cutdraw` instead of `draw`.

14.3 Showing crossings

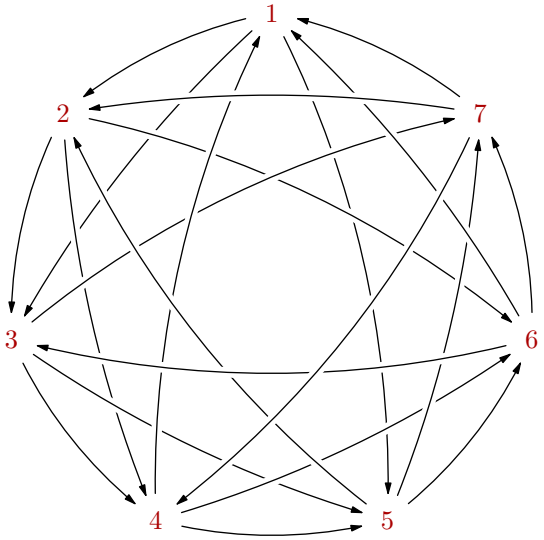
In general it is a good idea to avoid line crossings completely, but occasionally you may end up with a diagram where at least one line has to cross another. The simple way to deal with this is to use the technique shown in §19; capture the line to be drawn in a ⟨path⟩ variable, and then do

```
cutdraw line withpen pencircle scaled 4 withcolor background; draw line;
```

so that you erase behind the line before you draw it. Alternatively you could use the ideas in §9.4.2 to find all the intersections between two paths and mark them appropriately. But the marks might not improve legibility; compare these two ⟶



Do you think it looks better with crossings?



14.4 Using dash patterns with extra precision

As you may know, plain METAPOST provides two built-in dash patterns, so that you can draw a path `dashed withdots` or `dashed evenly`.

The keyword `dashed` gives you access to the PostScript `setdash` command, whose argument is a special ⟨picture⟩ defined with the METAPOST `dashpattern` function. If you look in `plain.mp` you will find these declarations:

```
picture evenly,withdots;
evenly = dashpattern(on 3 off 3); % dashed evenly
withdots = dashpattern(off 2.5 on 0 off 2.5); % dashed withdots
```

The detailed syntax is explained in §9.4 of the METAPOST manual, but essentially `withdots` creates a unit 5 points long with a dot in the middle, and `evenly` creates a unit 6 points long with the dashes 3pt long (plus the round bit at the end of each dash, unless you have changed `linecap`) and gaps 3pt long (minus any round bits).

You might be tempted to get creative with this and make complex dot-dot-dash patterns, but they rarely look very good and they may puzzle your readers. Scaling the two default patterns is probably all you ever need; so if you want a denser dotted line try `dashed withdots scaled 1/2`, or to get very long dashes you could use `dashed evenly scaled 4`.

But you may also notice that the dash patterns (particularly the longer ones) do not always fit your paths exactly – this is especially noticeable with closed paths, where you may end up with one unsightly long dash or a very short gap at the point where the path begins and ends.

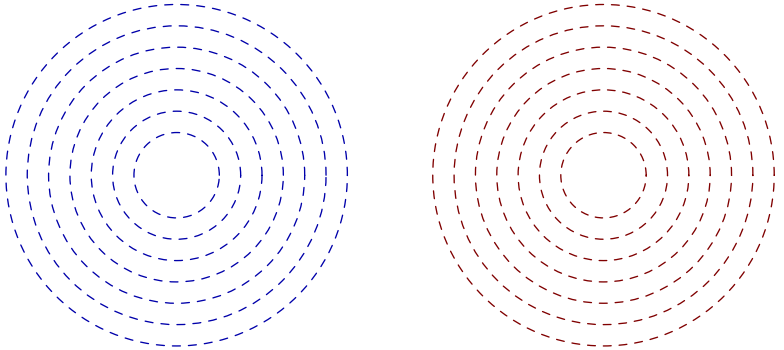
There is a simple solution: adjust the length of the dash pattern so that an integer number of dash units exactly fit your path.

```
vardef exactly(expr a, u) =
    save m; numeric m; 2m = (a-u) / round(a/u);
    dashpattern(on m off m)
enddef;
```

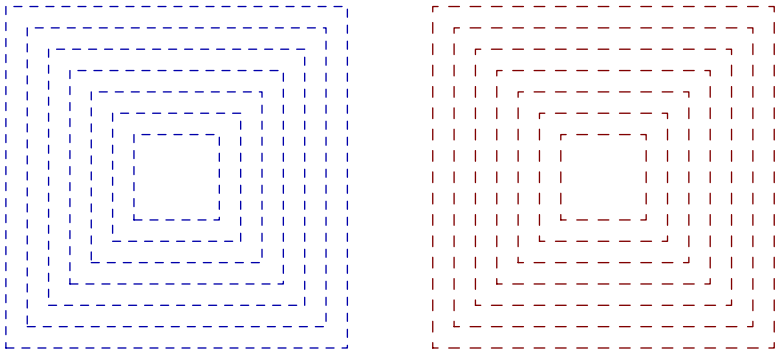
Here a is supposed to be the `arclength` of your path, and u the desired unit size, so you can use it like this:

```
path c; c = fullcircle scaled 200;
draw c dashed exactly(arclength c, 6);
```

to get a close approximation to `dashed evenly` that exactly fits the path.



The blue circles on the left were drawn with `dashed evenly`, and the uneven gaps are noticeable at the “three o’clock” positions where the paths begin and end. As you can see the default dash spacing looks fine at some sizes but bad on others. On the right you can see the same circular paths coloured red, and drawn with `dashed exactly(arclength c, 6)`.



Other paths may require a bit more ingenuity and thought. Because the square paths have four equal sides, they work better with a target dash length that is a multiple of 4. Here the blue squares on the left use the default `dashed evenly`, and the red squares on the right were done with:

```
dashed (exactly(arclength s, 8) shifted 6 right)
```

The right shift made the corners look better.

14.5 Decorating a path

A LITTLE DECORATION generally goes a long way, so you may want to restrain your creativity before you apply too many of the ideas from this page. There are two basic techniques shown here: creative use of dash patterns; and drawing shapes along the path.

With a curved path S, the first "with a dash pattern" was drawn like this:

```
draw S dashed dashpattern(on 4 off 2 on 1 off 2 on 1 off 2);
```

Notice that the default rounded pen makes dots and dashes with rounded ends. The second line "with a sharp dash pattern" uses `cutdraw` to change the line ends.

```
cutdraw S dashed dashpattern(on 4 off 1 on 1 off 1 on 1 off 1);
```

The "railway line" uses a combination of three drawing operations:

```
cutdraw S withpen pencircle scaled 2;
undraw S withpen pencircle scaled 5/4;
cutdraw S dashed evenly scaled 2 shifted right withpen pencircle scaled 5/4;
```

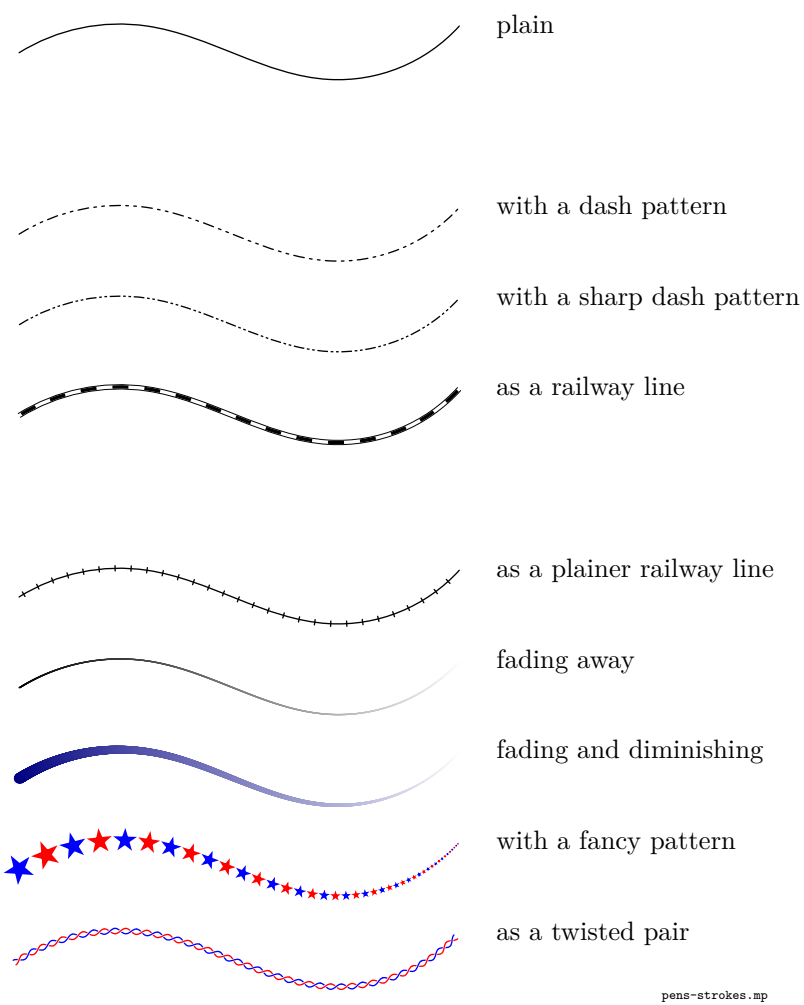
The "plainer railway line" was done like this:

```
draw S;
for a = 2 step 6 until arclength S:
  numeric t; t = arctime a of S;
  draw (down--up) rotated angle direction t of S shifted point t of S;
endfor
```

Note that it's essential to use `arctime` and `arclength` in order to get the markers evenly spaced. But you don't need to worry about the rotation in order to get the "fading away" effect:

```
numeric A; A = arclength S;
for a = 0 step 1/8 until A:
  draw point arctime a of S of S withcolor (a / A);
endfor
```

The remaining three are just fancy variations on the same theme. You might like to try to re-create them as an exercise, or you can look in the source file.



14.6 Morphing a path

A more flexible, but more complicated, decoration technique is to use a macro to morph your path before you actually draw it. This is how the venerable `feynmp` package marks photons and gluons etc in Feynman diagrams. Since `feynmp` is a standard part of the base METAPOST distribution you can use these macros in normal drawings; so to draw a zigzag line you can do:

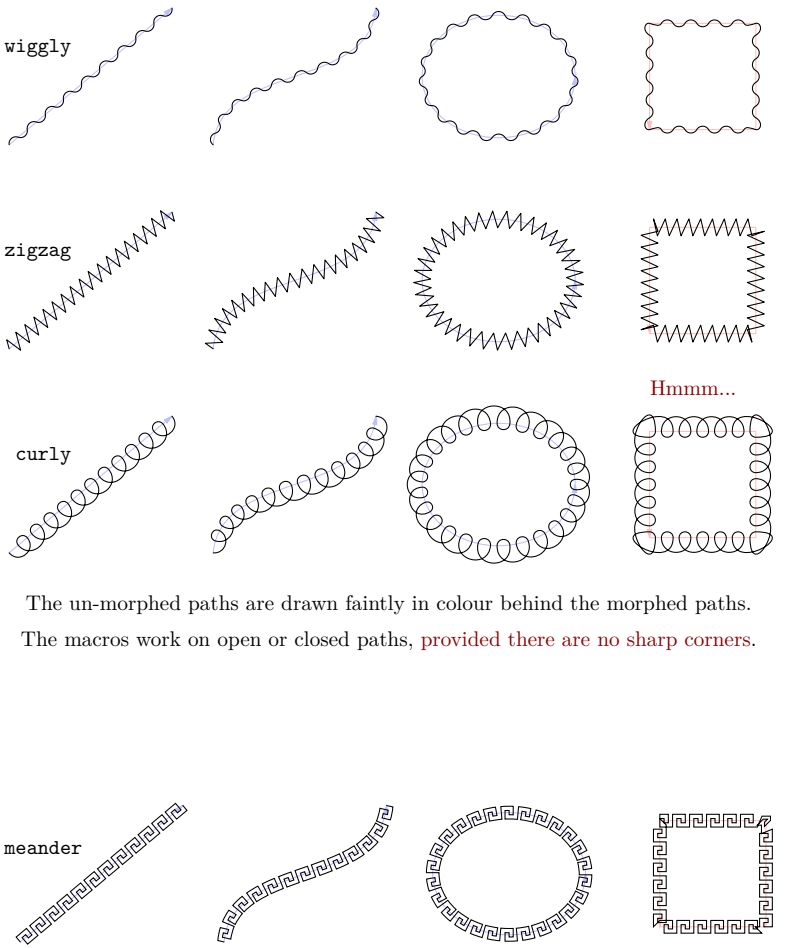
```
input feynmp
path S; S = (left {dir 30} .. right {dir 30}) scaled 100;
draw zigzag S;
```

The package also provides `curly`, and `wiggly`, as shown at the right, and defines a number of parameters to control the sizes of the shapes:

- `curly_len` sets the wave length of the loops, default 8.5
- `wiggly_len` ditto for waves, default 11.34
- `zigzag_len` ditto for zigs, default 5.67
- `wiggly_slope` steepness of waves, default 60°
- `zigzag_width`, amplitude of zigs, default 4

You might like to try your hand at defining your own. Here is one that does a vaguely hellenic meander pattern, adapted to cope with curved paths.

```
vardef meander expr p =
  save a, u, v, d, dy; numeric a, u, v, d; pair dy;
  d = 4 xpart urcorner makepath currentpen;
  a = arclength p; v = round(a/8d); u = a if v > 1: / v fi;
  if not cycle p: point 0 of p -- fi for t = 0 step u until a-4eps:
    hide(dy := d*unitvector(direction arctime t+1/2u of p of p rotated 90))
    subpath(arctime t of p, arctime t+u-2d of p) of p shifted 3dy
    --
    subpath(arctime t+u-2d of p, arctime t+1/2 u of p) of p shifted -dy
    --
    subpath(arctime t+1/2u of p, arctime t+2d of p) of p shifted dy
    --
    subpath(arctime t+2d of p, arctime t+u of p) of p shifted -3dy
    --
  endfor if cycle p: cycle else: point infinity of p fi
enddef;
```



The un-morphed paths are drawn faintly in colour behind the morphed paths.

The macros work on open or closed paths, provided there are no sharp corners.

14.7 Arrow styles

Plain METAPOST provides just two commands for drawing arrows: `drawarrow` and `drawdblarrow`. The default arrows are shown at (1) in the drawing on the right.→

There are two parameters that you can set to control the shape. The length of the arrow head is defined by `ahlength` which is set to 4pt, and the angle is defined by `ahangle` which starts at 45°. In some diagrams your arrows may look more elegant if they are a bit longer and slightly sharper. The arrows shown at (2) were created by setting `ahangle := 20;` and `ahlength := 6;` (note that you need to use the assignment operator to update them).

If your diagram needs a wider range of arrow head styles, perhaps because you are drawing UML, then you can use the `mparrows` package from CTAN. If you have a complete TEX distribution installed, you can just put `input mparrows` near the top of your program, and then use the `setarrows` macro to change the arrow style. In the drawing on the right

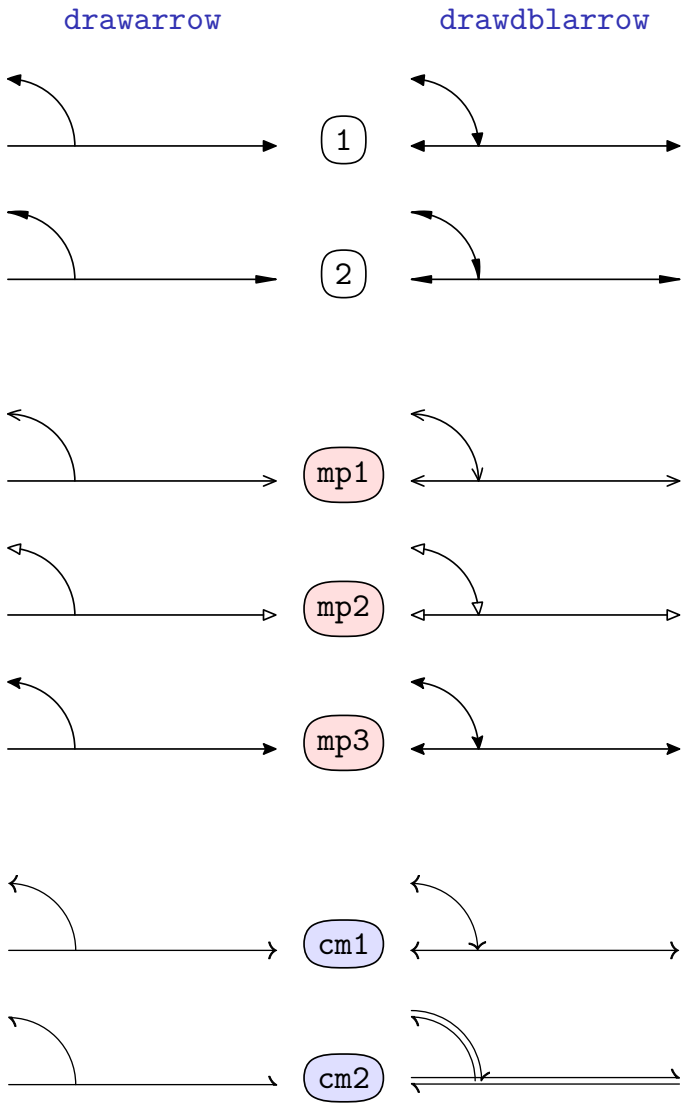
- (mp1) shows the result of `setarrows(open)`,
- (mp2) shows `setarrows(defaultunfilled)`, and
- (mp3) shows `setarrows(barbed)`.

The package uses the same length and angle parameters as the default arrows. It also provides an extra parameter called `barbedarrowindent` to control the shape of the barbed arrows. For full details try: `texdoc mparrows`

If you would rather have arrows in your drawings that match the various arrows provided by the Computer Modern font, then you can use the `cmarrows` package from CTAN. You can include this package by adding `input cmarrows` near the top of your program. It is slightly more complicated to control, but the details are explained in the manual. At (cm1) you can see the result of

```
setup_cmarrows(macro_name="drawarrow"; arrow_name="texarrow"; ... );
setup_cmarrows(macro_name="drawdblarrow"; arrow_name="twowayarrow"; ...);
```

Here I have chosen to override the built-in command, but you can assign a different macro name if you prefer. The arrow names used at (cm2) were `lefthalfarrow` and `paralleloppositelefthalfarrows`.



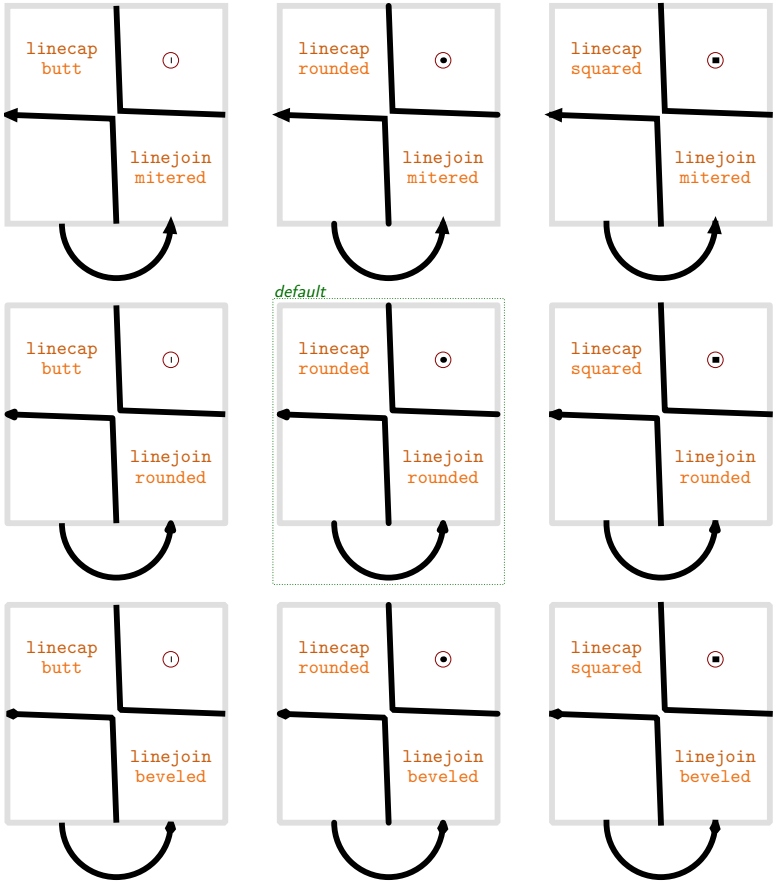
14.8 Line caps and line joins

The PostScript language defines parameters that affect how the ends of each line are drawn and how lines are joined together. Plain METAPOST provides access to these parameters through internal variables called `linecap` and `linejoin`; it sets both of them to the value `rounded` at the start of each job.

The figure on the right shows the effect of the different settings, using an exaggerated line width of 2 points (instead of the usual 0.5 points). Some observations to note:

- When `linecap := squared` then `drawdot` makes diamond-shaped dots, even when you are drawing with the default circular pen.
- When `linecap := butt` then `drawdot` produces invisible dots. They still count towards the bounding box of the picture but there's no mark on the page.
- When `linecap := squared` then `drawarrow` produces some unpleasant results; even when `linejoin := mitered`, you can still see small jaggies on the slopes of the arrows.
- The arrows are nice and sharp when `linejoin := mitered`, but they over shoot the mark slightly.
- If you zoom in, you can see the effect of `linejoin` on the corners in the centre as well as on the arrow heads, but you might not notice the difference when the picture is printed unless you have a very high resolution printer.
- This drawing was done with `pencircle scaled 2`, so that the dots would be easy to see. This does make the arrows drawn with the default line modes (rounded caps and rounded joins) looks a bit fat; they look better with the usual `pencircle scaled .5`, Like this: ⟷

There is one more PostScript parameter affecting line joins. METAPOST makes it available as `miterlimit` and it affects how much a mitered join is allowed to stick out at each corner. Plain METAPOST sets `miterlimit := 10;` which works well for most drawings. If you set `miterlimit := 0;` then the mitered line join mode becomes more or less the same as the beveled mode.



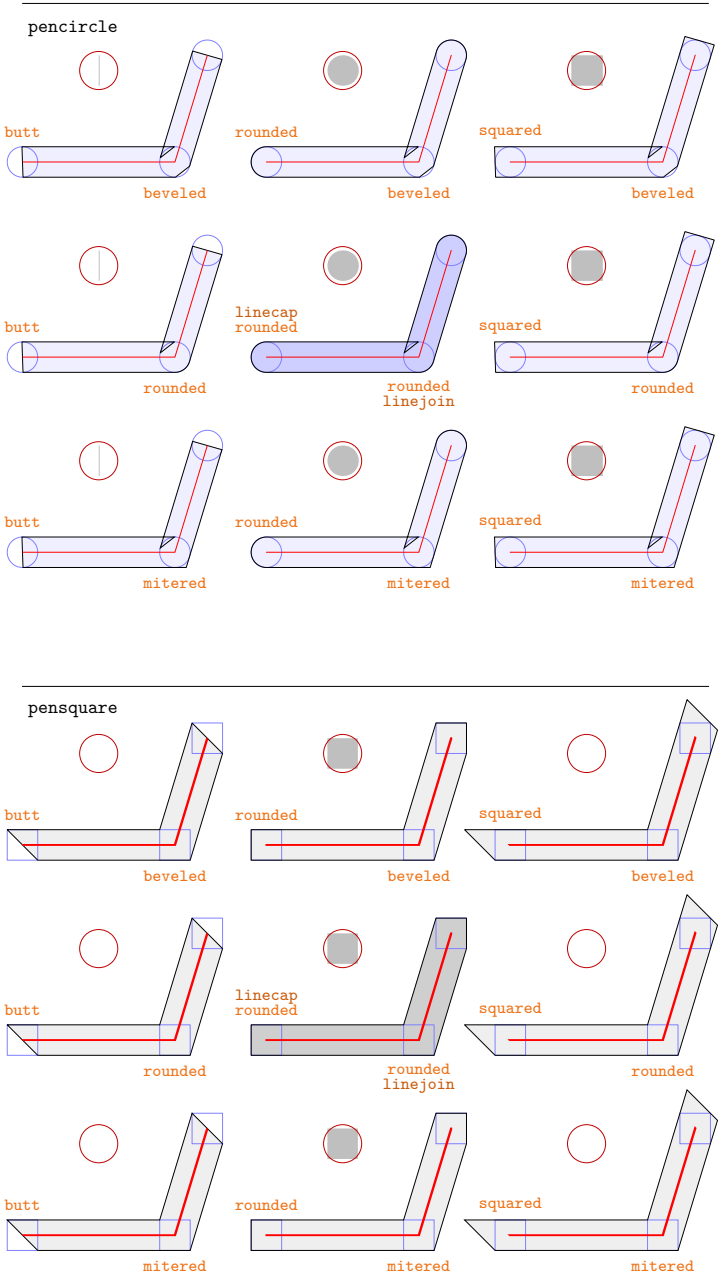
14.9 Line caps and line joins with square pens

THE PARAMETERS EXPLAINED in the previous section were designed (or at least named) for use with the default pen `pencircle`. You will get some rather odd shapes if you use them with any other pen, such as `pensquare`, especially if you use a large pen. In the drawings on the right, the thin red lines are drawn with each pen scaled to 0.5 bp (the default size), and the large grey and blue areas show what would be draw with the same pens scaled to 16 bp, so that you can see the artefacts more clearly. From these drawings you can see:

- The default setting `linecap := rounded` actually seems to mean "draw a dot with the current pen at each end of the path", so with `pensquare` you get a square dot on the end and it's not rounded at all.
- With `linecap := butt`, METAPOST appears to draw the terminal dots using half of the current pen. With the square pen, this has the unfortunate effect of cutting the pen at 45°, so that the end of the lines appear bevelled rather than squared off. Notice also that the cut is correctly rotated with the circular pen, but remains at 45° with the square pen.
- You get a similar effect with `linecap := squared`. METAPOST uses the same cut "across" the pen, but pushes it out so that it just touches the far edge of the pen-sized dot that would be drawn with `linecap := rounded`.

You can also see from the drawings that the effect of the `linejoin` setting, which is admittedly pretty subtle with a circular pen, is null with the square pen. And that the single point dots disappear unless you have `linecap := rounded`.

If you need to use a large square pen, then you can mitigate some of the artefacts if you rotate the pen by 45°. Using `pensquare rotated 45` corrects most of the faults, except that `linecap := rounded` will give you lines with pointed ends, and that the squared off ends of the lines with the other settings may not be exactly orthogonal to the direction of the path at each end.



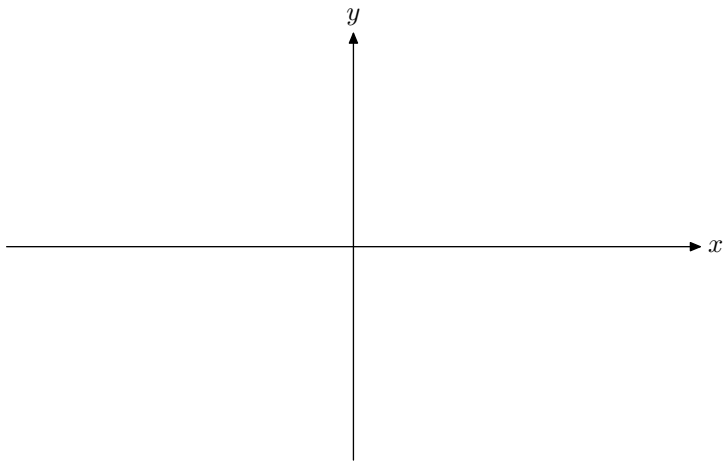
15 Plotting functions

A SELECTION of graphs of mathematical functions is presented in this section, taken from real examples collected over several years. For data visualizations, see §18. As ever in this document, the focus is on plain METAPOST; the plain format provides no built-in facilities for graphs so you have to do everything from scratch; but on the other hand there are no new macros or commands to learn, you get full control of what goes on the page, and you will not spend hours scratching your head wondering how to adjust the axis labels.

15.1 Making axes

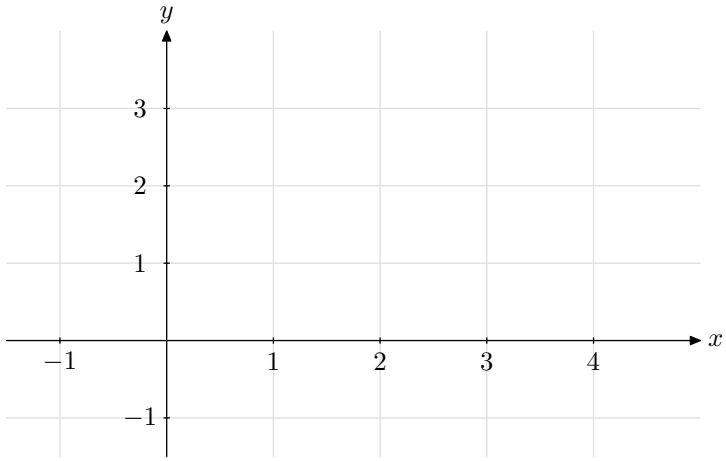
You can start by drawing a simple set of axes.

```
path xx, yy;
xx = (left -- right) scaled 130;
yy = (down -- up) scaled 80;
drawarrow xx; label.rt(TEX("$x$"), point 1 of xx);
drawarrow yy; label.top(TEX("$y$"), point 1 of yy);
```



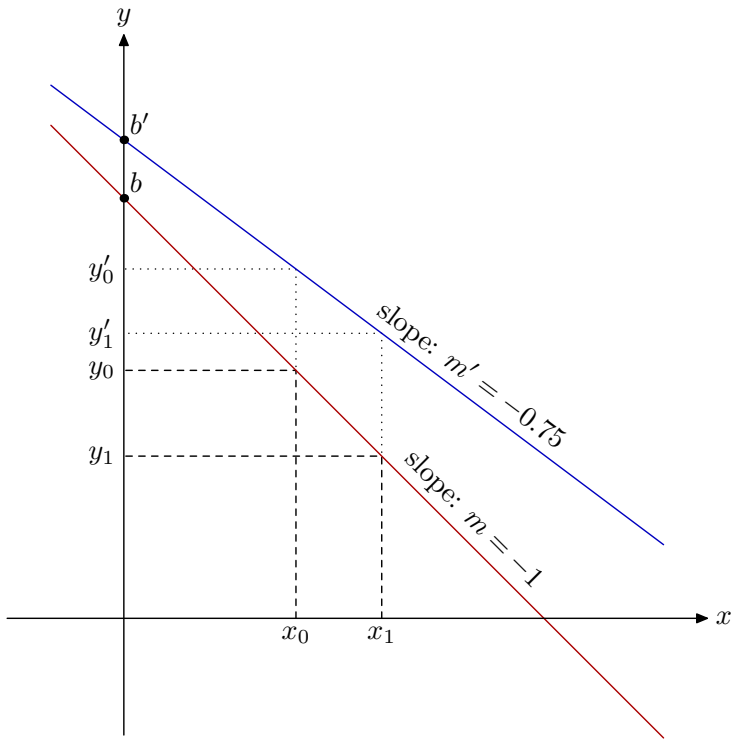
Here the axes are scaled arbitrarily to 130 pt and 80 pt, but you will probably find it useful to set consistent units, and express sizes in terms of them. Purely from habit, I use u for the horizontal unit and v for the vertical unit. This makes it more convenient when you want to add a grid and/or a number scale.

```
numeric u, v; u = 40; v = 29; path xx, yy;
xx = (3/2 left -- 5 right) scaled u;
yy = (3/2 down -- 4 up) scaled v;
for x=-1, 1, 2, 3, 4:
  draw yy shifted (x * u, 0) withcolor 7/8;  % grid
  draw (down--up) shifted (x * u, 0);        % ticks
  label(TEX("$" & decimal x & "$"), (x * u, -8));
endfor
for y=-1, 1, 2, 3:
  draw xx shifted (0, y * v) withcolor 7/8;  % grid
  draw (left--right) shifted (0, y * v);     % ticks
  label(TEX("$" & decimal y & "$"), (-10, y * v));
endfor
drawarrow xx; label.rt(TEX("$x$"), point 1 of xx);
drawarrow yy; label.top(TEX("$y$"), point 1 of yy);
```



15.2 Drawing linear functions

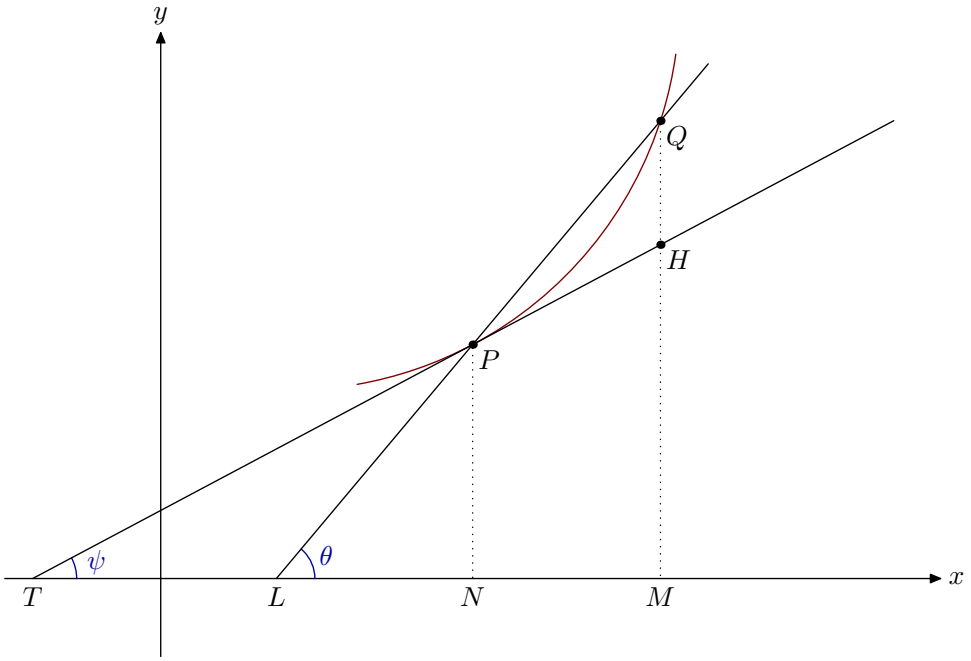
For simple linear graphs, you just need to define two points and draw a line between them; it is tempting to try to make some generalized macro to do this, but it is hard to make something completely general, so for most graphs it is easier just to specify two points and use `draw`; often it is handy to making your line longer than you need, then trim it using `cutbefore` and/or `cutafter` so that it fits neatly.



But in this example it was easier to calculate them using $y = mx + b$. Note also that spaces are allowed in suffixes, which makes the loop a bit simpler.

```
beginfig(1); % Using \mplibtextextlabel{enable} ...
  numeric u, m, m', b, b';
  u = 1.44cm;
  b = 3.6u;  b' = b + 1/2 u;
  m = -1;  m' = 3/4 m;

  path xx, yy;
  xx = (left -- 5 right) scaled u;
  yy = xx rotated 90;

  numeric minx, maxx; path ff, gg;
  minx = xpart point 1/16 of xx;
  maxx = xpart point 15/16 of xx;
  ff = (minx, minx * m + b) -- (maxx, maxx * m + b);
  gg = (minx, minx * m' + b') -- (maxx, maxx * m' + b');

  z0 = point 0.4 of ff;
  z1 = point 0.54 of ff;
  z1 0 = whatever [point 0 of gg, point 1 of gg]; x1 0 = x0;
  z1 1 = whatever [point 0 of gg, point 1 of gg]; x1 1 = x1;

  forsuffixes @=0, 1:
    draw (x@, 0) -- z@ -- (0, y@) dashed evenly scaled 3/4;
    draw z@ -- z1 @ -- (0, y1 @) dashed withdots scaled 1/2;
    label.bot("$x_{" & decimal @ & "}$", (x@, 0));
    label.lft("$y_{" & decimal @ & "}$", (0, y@));
    label.lft("$y'_{" & decimal @ & "}$", (0, y1 @));
  endfor

  draw ff withcolor 2/3 red;
  draw gg withcolor 3/4 blue;
  drawarrow xx; drawarrow yy;

  label.rt("$x$", point 1 of xx);
  label.top("$y$", point 1 of yy);

  dotlabel.urt("$b$", (0, b));
  dotlabel.urt("$b'$", (0, b'));

  draw thelabel("slope: $m=" & decimal m & "$", 7 up)
    rotated angle (1, m) shifted point 2/3 of ff;
  draw thelabel("slope: $m'=" & decimal m' & "$", 7 up)
    rotated angle (1, m') shifted point 2/3 of gg;
endfig;
```

In this second example of a drawing with linear functions, the emphasis of the diagram was on the angles at the x-axis made by the two lines, so the lines were defined using `rotated`, `shifted`, and `cutbefore` instead.



The lines are both trimmed to a convenient path (*boundary*) when they are drawn.

```
beginfig(1);
  numeric u; u = 1cm;
  path xx; xx = (2 left -- 10 right) scaled u;
  path yy; yy = (down -- 7 up) scaled u;

  numeric theta, psi; psi = 28; theta = 50;
  pair P, Q, H, N, M, T, L;  P = (4, 3) scaled u;
```

...continued

```
  path ell, tee, arc;
  ell = (left--right) scaled 10u rotated theta shifted P cutbefore xx;
  tee = (left--right) scaled 10u rotated psi   shifted P cutbefore xx;
  arc = subpath (1.6, 3.2) of halfcircle rotated -180 shifted 1/2 up
        scaled 10u shifted P rotatedabout(P, psi);

  Q = ell intersectionpoint subpath (1.5, 3) of arc;
  H = P + whatever * dir psi;
  xpart H = xpart Q = xpart M;  ypart M = ypart N = 0;
  xpart N = xpart P;
  L = point 0 of ell;
  T = point 0 of tee;

  % now we can get with the drawing
  draw arc withcolor 1/2 red;
  draw P--N dashed withdots scaled 1/2;
  draw Q--M dashed withdots scaled 1/2;
  drawarrow xx; label.rt("$x$", point 1 of xx);
  drawarrow yy; label.top("$y$", point 1 of yy);
  drawoptions(withcolor 2/3 blue);
    draw fullcircle scaled 32 shifted T cutafter tee;
    draw fullcircle scaled 28 shifted L cutafter ell;
    label("$\psi$", 24 right rotated 1/2 psi shifted T);
    label("$\theta$", 20 right rotated 1/2 theta shifted L);
  drawoptions();

  % draw the lines trimmed neatly
  path invisible_boundary;
  z1 = point .95 of xx;
  z2 = point .95 of yy;
  invisible_boundary = z1--(x1,y2)--z2;
  draw ell cutafter invisible_boundary;
  draw tee cutafter invisible_boundary;

  % and finally label the points.
  forsuffixes @ = T, L, N, M: label.bot("$" & str @ & "$", @); endfor
  forsuffixes @ = H, P, Q: dotlabel.lrt("$" & str @ & "$", @); endfor
endfig;
```

15.3 Making curves for functions with a loop

To plot a function you can construct a suitable path using an in-line **for** loop like this:

```
vardef f(expr x) = x ** 2 enddef;
path ff;
ff = (for x = minx step s until maxx - s:
        (x, f(x)) ..
    endfor (maxx, f(maxx))) xscaled u yscaled v;
```

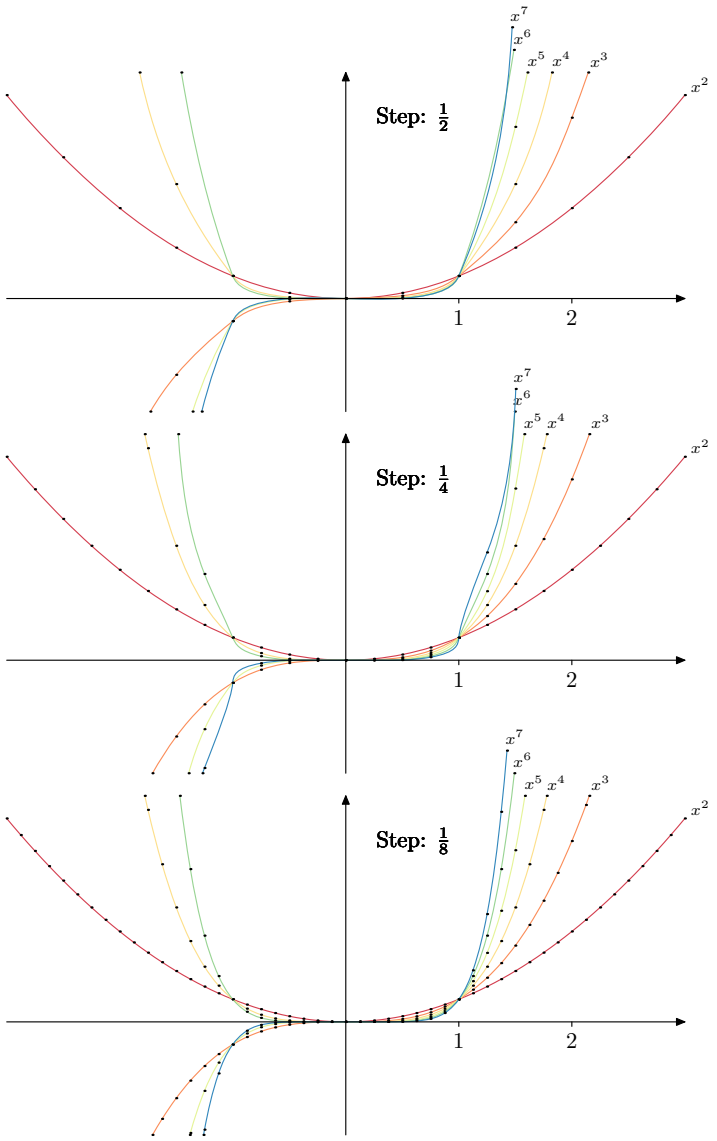
provided you have first defined variables *minx* and *maxx* to represent the domain of x, and worked out appropriate values for horizontal and vertical units, u, and v so that the range of $f(x)$ fits neatly on your graph.

The loop above also uses a variable s to control the number of points used to define the path. The figures on the right show that over the domain -3 to 3 a step of $\frac{1}{2}$ gives enough points for METAPOST's Bezier curve fitting routines to draw the functions x^2 and x^3 accurately, but that you need a step size of $\frac{1}{8}$, and hence four times as many points, for x^6 and x^7. On modern machines it does not really hurt to calculate dozens of points, but a step size that generates 1000s of points will be slow to compile.

There are two other techniques to improve the shape of the curve produced by these loops: you can increase the tension between each point by using `...` or `--` instead of `..` in the loop; and if you know how to differentiate your function, you can add a direction at each step using the {⟨pair⟩} syntax:

```
vardef f(expr x) = x ** 2 enddef;
vardef fp(expr x) = 2x enddef; % NB "fp" because "f'" is illegal
path ff;
ff = (for x = minx step s until maxx - s:
        (x, f(x)){1, fp(x)} ..
    endfor (maxx, f(maxx))) xscaled u yscaled v;
```

☞ However in general it is simpler just to increase the number of samples by making a smaller step size.

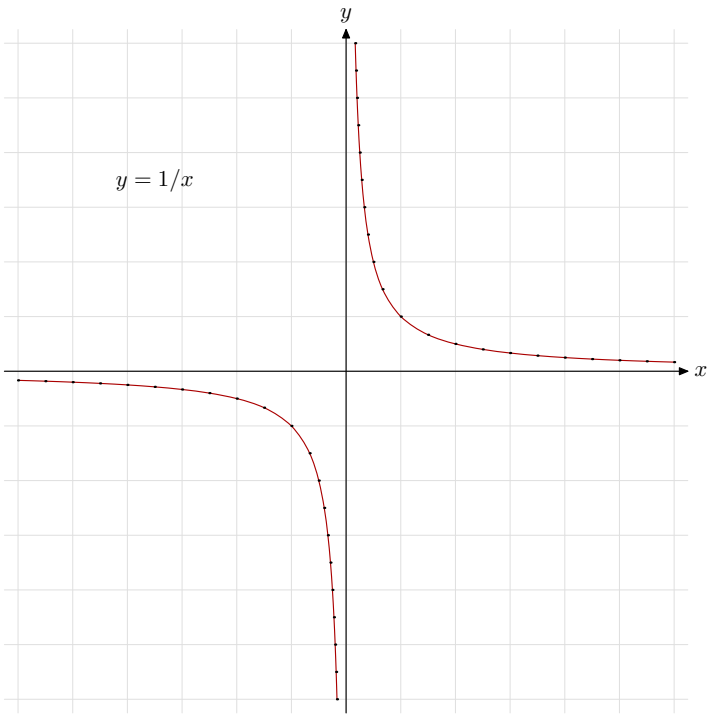


15.4 Making curves for functions from path pieces

In some situations, you might find it easier to stitch together various ⟨path⟩ pieces to make your curve. This can be especially elegant if there is a symmetry in the path. For example:

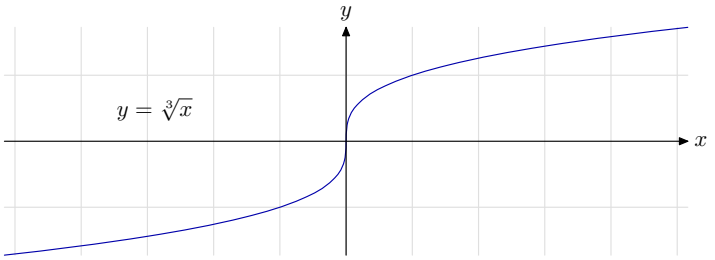
```
path ff, negative_ff;

ff = (1,1) for x = 3/2 step 1/2 until 6: ... (x, 1/x) endfor;
ff := reverse ff reflectedabout(origin, dir 45) & ff;
ff := ff scaled 24;
negative_ff = ff reflectedabout(origin, dir -45);

draw ff withcolor 2/3 red;
draw negative_ff withcolor 2/3 red;
```

(Omitting code for the axes, the grid, and the dots at each `point` of the paths).



- Notice that you can update the path using the assignment operator "`:=`".
- You need to `reverse` the reflected portion so that the two ends coincide.
- The two path segments are spliced together with `&`. You could use a path join like `..` instead, but then the joined path would have an extra `point` at `(1,1)`.
- Notice also how the vertical part has the same equal spacing of points as the more horizontal part.

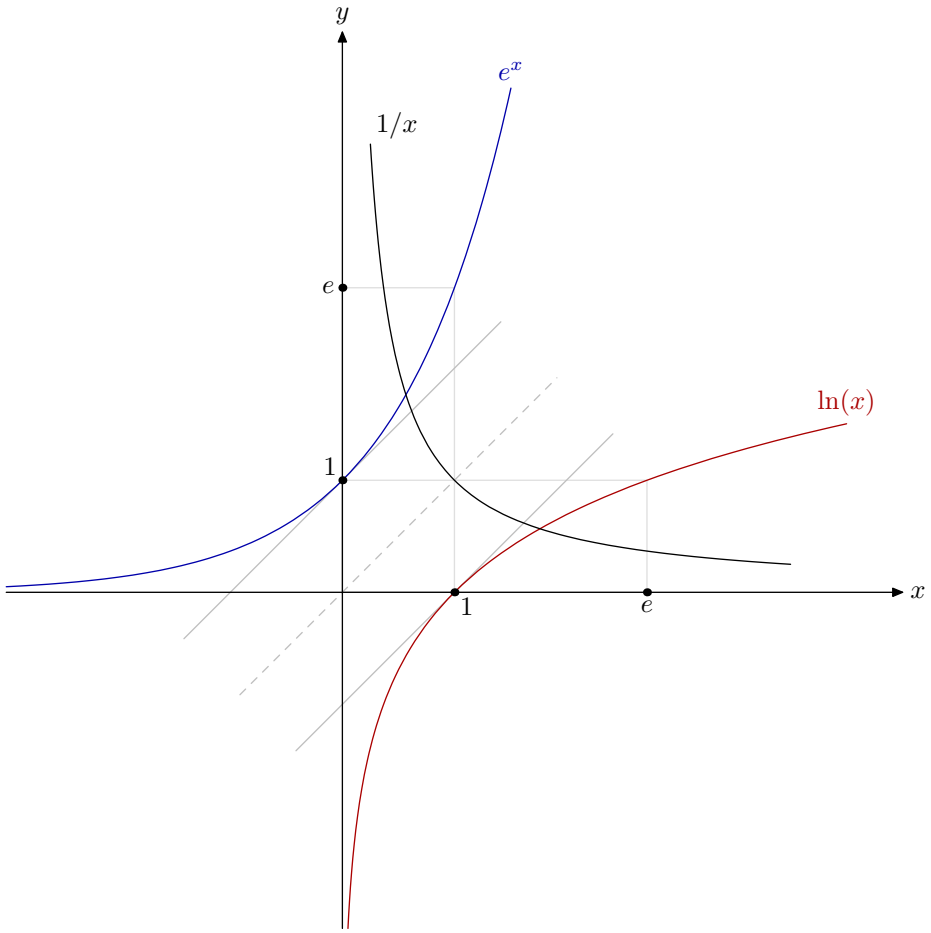
Reflection of a function in the line at 45° gives the inverse of the function, which is especially useful for $y = 1/x$, but it applies to functions generally. So if you want to plot $y = \sqrt{x}$ it may be easier to define a path for $y = x^2$ and then reflect it. This is particularly useful if you want to plot, say, $y = \sqrt[3]{x}$, over a domain that includes negative numbers, because METAPOST will not calculate reciprocal powers of negative numbers. The curve in this chart was created by reflecting the line $y = x^3$.



15.4.1 Exponential and logarithm functions by reflection

A FURTHER EXAMPLE of creating paths by transformation.

```
beginfig(1);
  numeric u, minx, s, maxx;
  u = 42; minx = -3; s = 1/4; maxx = 1/256 mlog(4.5);
  path xx; xx = (-3u, 0) -- (5u, 0);
  path yy; yy = xx rotated 90;
  path ee, ll, nn;
  ee = (for x = minx step s until maxx - s:
    (x, mexp(256x)){1, mexp(256x)} ...
  endfor (maxx, mexp(256 maxx)){1, mexp(256 maxx)}) scaled u;
  ll = ee reflectedabout(origin, dir 45);
  nn = (for x=1 step s until 4-s: (x, 1/x) ... endfor (4, 1/4)) scaled u;
  nn := reverse nn reflectedabout(origin, dir 45) & nn;
  drawoptions(withcolor 7/8);
    draw unitsquare xscaled mexp(256) scaled u;
    draw unitsquare yscaled mexp(256) scaled u;
  drawoptions(withcolor 3/4);
    forsuffixes $ = ee, ll:
      path T$; T$ = (left--right) scaled 2u rotated 45
                          shifted directionpoint dir 45 of $;
      draw T$;
    endfor
    draw interpath(1/2, T.ee, T.ll) dashed evenly;
  drawoptions(withcolor 2/3 blue);
    draw ee; label.top("$e^x$", point infinity of ee);
  drawoptions(withcolor 2/3 red);
    draw ll; label.top("$\ln(x)$", point infinity of ll);
  drawoptions();
  draw nn; label.urt("$1/x$", point 0 of nn);
  drawarrow xx; label.rt("$x$", point 1 of xx);
  drawarrow yy; label.top("$y$", point 1 of yy);
  dotlabel.lft("$e$", (0, mexp(256) * u));
  dotlabel.bot("$e$", (mexp(256) * u, 0));
  dotlabel.ulft("$1$", (0, u));
  dotlabel.lrt("$1$", (u, 0));
endfig;
```

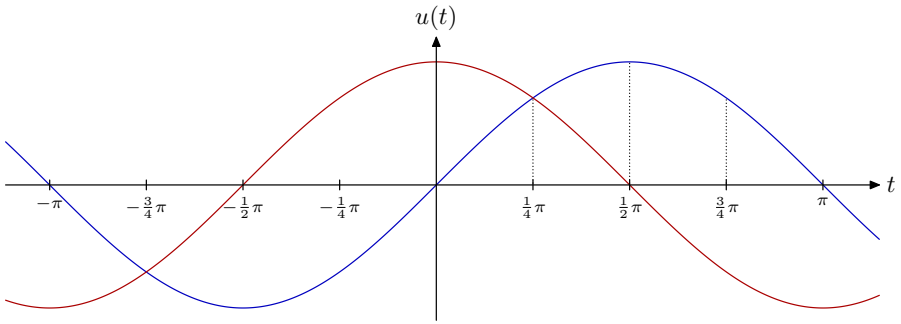


If you prefer more ‘normal’ functions, you can define:

```
vardef exp(expr x) = mexp(256x) enddef;
vardef log(expr x) = 1/256 mlog(x) enddef;
```

15.5 Functions using trigonometric functions

As noted in §10, METAPOST's built-in trigonometric functions work in degrees, this example shows how you might use them in a graph.



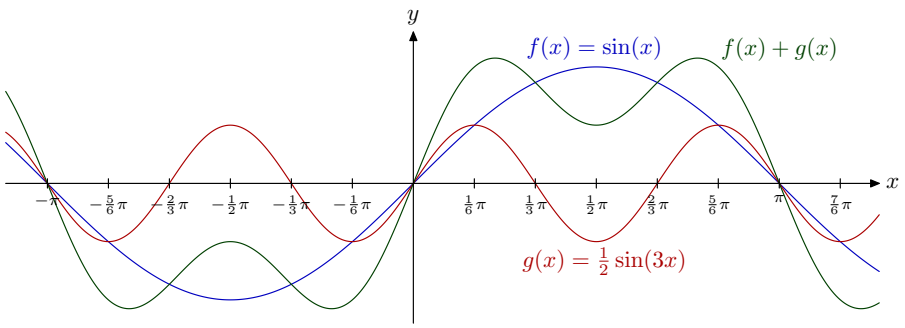
For this diagram, the sine wave path (`ss`, shown in blue) needs to have two complete cycles, so it is constructed in stages. First the section from the origin to 2π is created in a loop; with 360 steps, you can use `--` and still get a smooth path. Secondly the cycle is duplicated by splicing itself to a shifted copy. Thirdly it is x-scaled to radians, and then scaled in both directions to the chosen unit size, and drawn chopped off to the width of the x-axis. The cosine path is the same path, shifted $\frac{1}{2}\pi$ left, drawn in red, and chopped off to fit the same width. The fancy fraction labels were produced with this subroutine:

```
vardef pi_quarters(expr n) =
  save s, f, q; string s, f; numeric q;
  s = if n < 0: "-" else: "" fi; q = abs(n);
  if q mod 4 = 0: f = if q > 4: decimal 1/4 q else: "" fi;
  elseif q mod 2 = 0: f = "\frac{" & decimal 1/2 q & "}{2}";
  else: f = "\frac{" & decimal q & "}{4}";
  fi TEX("$\scriptstyle" & s & f & "\pi$")
enddef;
```

```
beginfig(1);
  numeric u, pi; u = 50; pi = 3.141592653589793;

  path xx, yy;
  xx = (3.5 left -- 3.6 right) scaled u;
  yy = (1.1 down -- 1.2 up) scaled u;

  path ss;
  ss = origin for t=1 upto 360: -- (t, sind(t)) endfor;
  ss := ss shifted 360 left & ss;
  ss := ss xscaled (pi/180) scaled u;

  drawoptions(dashed withdots scaled 1/4);
  draw ((1/4 pi, 0) .. (1/4 pi, sind(45))) scaled u;
  draw ((1/2 pi, 0) .. (1/2 pi, sind(90))) scaled u;
  draw ((3/4 pi, 0) .. (3/4 pi, sind(135))) scaled u;
  drawoptions();

  draw ss
    cutbefore yy shifted point 0 of xx
    cutafter  yy shifted point 1 of xx
    withcolor 3/4 blue;

  draw ss shifted (-1/2 pi * u ,0)
    cutbefore yy shifted point 0 of xx
    cutafter  yy shifted point 1 of xx
    withcolor 2/3 red;

  drawarrow xx; label.rt(TEX("$t$"), point 1 of xx);
  drawarrow yy; label.top(TEX("$u(t)$"), point 1 of yy);

  for i=-4, -3, -2, -1, 1, 2, 3, 4:
    draw (down--up) scaled 2 shifted (pi * i/4 * u, 0);
    label.bot(pi_quarters(i), (pi * i/4 * u, -2));
  endfor
endfig;
```

15.6 Manipulating functions

This second example with trigonometric functions shows one way to add two functions, by combining the METAPOST paths themselves.



Notice how the same extension, scaling, and trimming operations can be applied to all three paths using a `forsuffixes` loop. Note that you can use any regular variable name for the loop index; you don't have to use `$`, but like `@` it is a valid variable name in METAPOST, and it looks a bit like a placeholder marker in other languages.

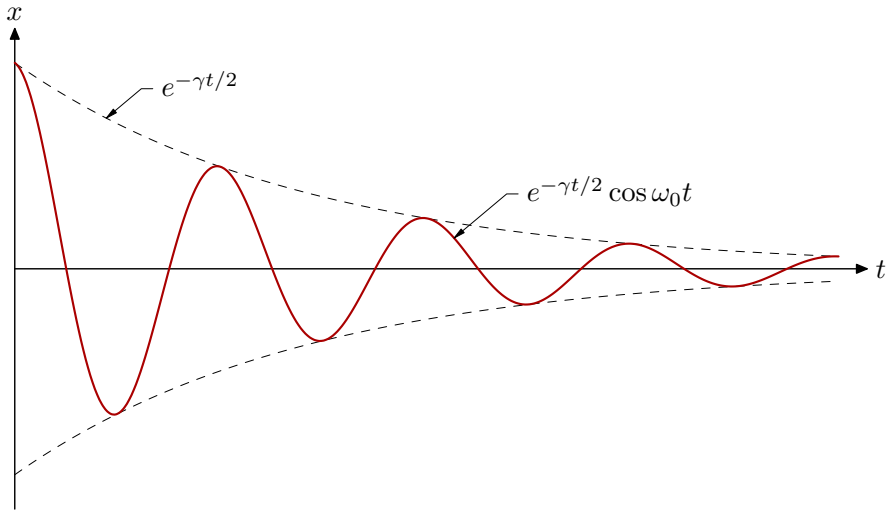
Note that you need to start with the first point of the path outside the loop so that you don't end up with a dangling `--` path connector. Using `origin` is just a short cut for writing `(0, sind(0))`. If you were plotting a different function this would not work. For example, `(0, cosd(0))` is `(0,1)`.

☞ The missing `pi_sixths` macro is left as an exercise for the reader. Hint: you can adapt the `pi_quarters` on the previous page, allowing for halves, thirds, and sixths instead of halves and quarters.

```
numeric u, pi; u = 50; pi = 3.141592653589793;

path xx, yy;
xx = (3.5 left -- 4 right) scaled u;
yy = (1.2 down -- 1.3 up) scaled u;

path ss, tt, uu;
ss = origin for x=1 upto 360: -- (x, sind(x)) endfor;
tt = origin for x=1 upto 360: -- (x, 1/2 sind(3x)) endfor;
uu = origin for x=1 upto 360:
    -- (x, ypart point x of ss + ypart point x of tt)
endfor;

forsuffixes $=ss, tt, uu:
  $ := $ shifted 360 left & $;
  $ := $ xscaled (pi/180) scaled u;
  $ := $ cutbefore yy shifted point 0 of xx
         cutafter  yy shifted point 1 of xx;
endfor

drawoptions(withcolor 3/4 blue);
  draw ss; label.top("$f(x)=\sin(x)$", point 290 of ss);
drawoptions(withcolor 2/3 red);
  draw tt; label.bot("$g(x)=\frac12 \sin(3x)$", point 295 of tt);
drawoptions(withcolor 1/4 green);
  draw uu; label.urt("$f(x) + g(x)$", point 350 of uu);
drawoptions();

drawarrow xx; label.rt("$x$", point 1 of xx);
drawarrow yy; label.top("$y$", point 1 of yy);

for i=-6, -5, -4, -3, -2, -1, 1, 2, 3, 4, 5, 6, 7:
  draw (down--up) scaled 2 shifted (pi * i/6 * u, 0);
  label.bot(pi_sixths(i), (pi * i/6 * u, -2));
endfor
```

Manipulating functions arithmetically

This third example with trigonometric functions shows a second way to add two functions. This approach is more mathematical, and perhaps more natural.



- Notice how the inline for-loops help you organize the code visually.
- In this example you know that $e^0 = \cos 0 = 1$, so the first point of each curve is going to be $(0, 1)$ before scaling — which you can write as `up`.
- Recall also that `mexp(x)` is defined as $e^{x/256}$ so with `mexp(-4t)`, you have $\gamma/2 = 4/256 = 1/64$ — this controls steepness of the exponential curve.
- 180 points is plenty to get nice smooth curves, and `cosd(8t)` ensures that there are four full cycles in the graph.

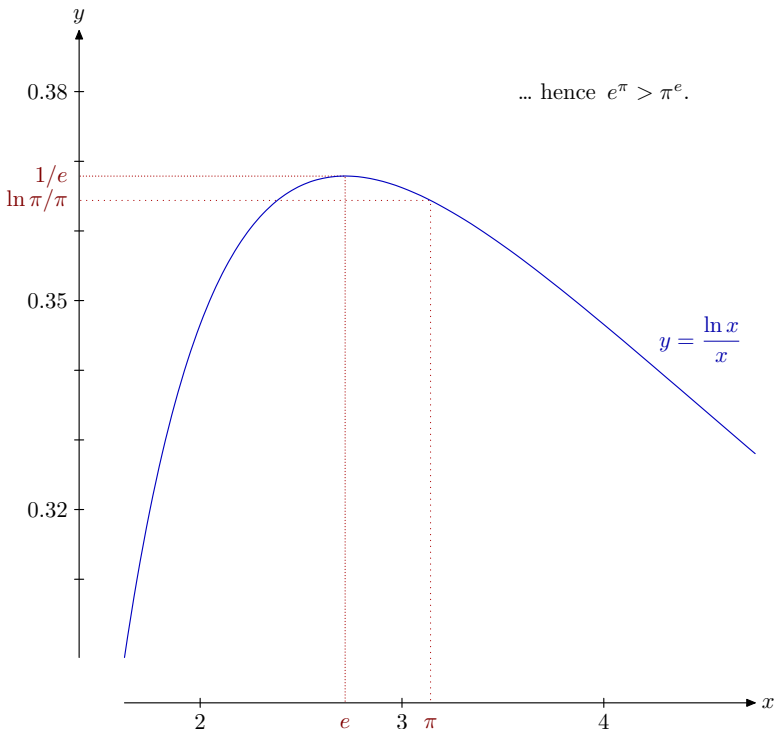
```
path ee, ff, xx, yy;
ee = up for t = 1 upto 180: .. (t, mexp(-4t)) endfor;
ff = up for t = 1 upto 180: .. (t, mexp(-4t) * cosd(8t)) endfor;

ee := ee xscaled 1.6 yscaled 72;
ff := ff xscaled 1.6 yscaled 72;

xx = origin -- (10 + xpart point length ff of ff, 0);
yy = 84 down -- 84 up;

drawarrow xx; label.rt("$t$", point 1 of xx);
drawarrow yy; label.top("$x$", point 1 of yy);

drawoptions(dashed evenly withpen pencircle scaled 1/4);
  draw ee;
  draw ee reflectedabout (left, right);

drawoptions(withpen pencircle scaled 3/4 withcolor 2/3 red);
  draw ff;

drawoptions();

vardef pin(expr t, w, o) =
  interim ahangle := 24;
  drawarrow w shifted o shifted 4 right -- w shifted o -- w
    withpen pencircle scaled 1/4;
  label.rt("\strut" & t, w shifted o shifted 4 right);
enddef;

pin("$e^{-\gamma t/2}$", point 20 of ee, 16 dir 40);
pin("$e^{-\gamma t/2}\cos\omega_0t$", point 96 of ff, 24 dir 40);
```

15.7 Focus on a specific region of a function

This visual proof required a large y-axis scale. The axes are separated to show the discontinuity in scales, and that the origin is not on the chart.

```
numeric minx, maxx, s, u, v;
minx = 13/8; s = 1/16; maxx = 19/4; u = 89; v = 3072;

def f(expr x) = 1/256 mlog(x) / x enddef;

path ff, xx, yy;
ff = for x=minx step s until maxx-s: (x, f(x)) .. endfor (maxx, f(maxx));
ff := ff xscaled u yscaled v;
xx = origin -- right scaled (maxx-minx) scaled u;
yy = origin -- up scaled 0.09v;
xx := xx shifted point 0 of ff shifted 20 down;
yy := yy shifted point 0 of ff shifted 20 left;

numeric pi, e, fpi, fe;
pi = 3.141592653589793 u;  fpi = f(3.141592653589793) * v;
e = 2.718281828459045 u;   fe = f(2.718281828459045) * v;
path ee, pp;
ee = (e, ypart point 0 of xx) -- (e, fe) -- (xpart point 0 of yy, fe);
pp = (pi, ypart point 0 of xx) -- (pi, fpi) -- (xpart point 0 of yy, fpi);
draw ee dashed withdots scaled 1/4 withcolor 2/3 red;
draw pp dashed withdots scaled 1/2 withcolor 2/3 red;
draw ff withcolor 3/4 blue;
drawarrow xx;
drawarrow yy;

for x=2 upto 4:
    draw (down--up) scaled 2 shifted (x * u, ypart point 0 of xx);
    label.bot("$" & decimal x & "$", (x * u, ypart point 0 of xx - 2));
endfor
for y=.31, .32, .33, .34, .35, .36, .37, .38:
    draw (left--right) scaled 2 shifted (xpart point 0 of yy, y*v);
endfor
for y=.32, .35, .38:
    label.lft("$" & decimal y & "$", (xpart point 0 of yy-2, y*v));
endfor
```

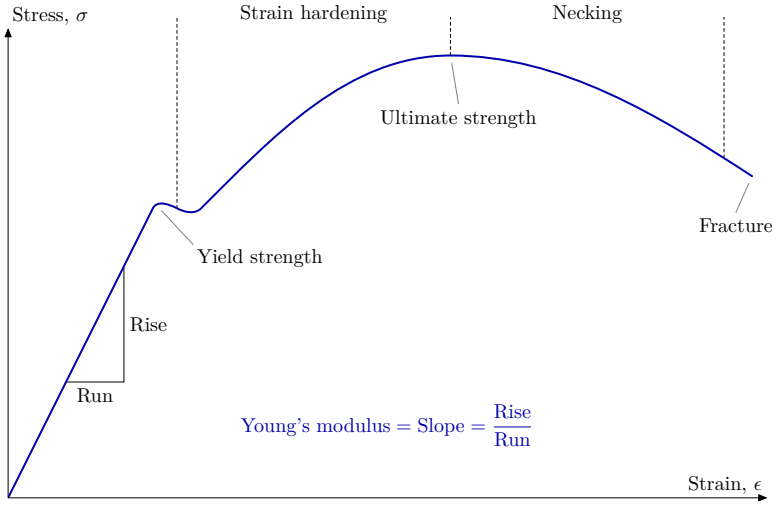
...continued




```
drawoptions(withcolor 1/2 red);
label.bot("$e$", point 0 of ee shifted 4 down);
label.lft("$1/e$", point 2 of ee shifted 2 left);

label.bot("$\pi$", point 0 of pp shifted 4 down);
label.lft("$\ln\pi/\pi$", point 2 of pp shifted 2 left);

drawoptions(withcolor 2/3 blue);
label.urt("$\displaystyle y={\ln x\over x}$", point 42 of ff);

drawoptions();
label.rt("$x$", point 1 of xx);
label.top("$y$", point 1 of yy);
label("\dots\ hence\enspace $e^\pi > \pi^e$.", (4u, 0.38v));
```

15.8 Approximate function diagrams

SOMETIMES you may need to plot a function that does not have a simple mathematical definition. You can a METAPOST⟨path⟩ to make a likely looking approximation.



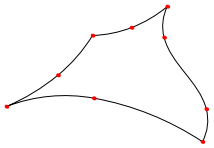

```
beginfig(1);
  z1 = 377 right;  z2 = 233 up;
  path ff; ff = origin .. (72, 144){1,2} .. (84, 144) ..
        (96, 144){1,1} .. (220, 220){right} .. (370, 160){2,-1.3};

  for t=2, 4, 4.9:
    draw point t of ff -- (xpart point t of ff, y2 + 6) d
      ashed evenly scaled 1/2;
  endfor
  label.top("Strain hardening",
      (1/2 (xpart point 2 of ff + xpart point 4 of ff), y2));
  label.top("Necking",
      (1/2 (xpart point 4 of ff + xpart point 4.9 of ff), y2));

  path rr;
  rr = point 0.4 of ff -- (xpart point 0.8 of ff, ypart point 0.4 of ff) -- point 0.8 of ff;
  draw rr; label.bot("Run", point 1/2 of rr); label.rt("Rise", point 3/2 of rr);

  vardef pin_label@#(expr p, a, b)=
    draw a -- b cutbefore fullcircle scaled 8 shifted a withpen pencircle scaled 1/4 withcolor 1/2 white;
    label@#(p, b);
  enddef;
  pin_label.lrt("Yield strength",     point 1.2 of ff, point 2 of ff + (8, -18));
  pin_label.bot("Ultimate strength",  point 4   of ff, point 4 of ff + (4, -24));
  pin_label.bot("Fracture",           point 5   of ff, point 5 of ff + (-8, -18));

  draw ff withpen pencircle scaled 1 withcolor 2/3 blue;
  clip currentpicture to unitsquare scaled 400; % clip thick pen at origin

  drawdblarrow z1 -- origin -- z2;
  label.ulft("Strain, $\epsilon$", z1);
  label.urt("Stress, $\sigma$", z2);

  label("$\displaystyle\hbox{Young's modulus} = \hbox{Slope} = {\hbox{Rise}\over\hbox{Run}}$", 1/2 z1 shifted 36 up) withcolor 2/3 blue;
endfig;
```

15.8.1 Taming Bezier paths with controls

It takes some practice to translate a sketch of a curve into a smooth path. Plain METAPOST inherits from METAFONT a useful *flex* macro, that takes a list of ⟨pair⟩s (any number of them) and produces a pleasing path through them. As Knuth says in the *The METAFONTbook*: "The idea is to specify two endpoints, z_1 and z_n, together with one or more intermediate points where the path is traveling in the same direction as the straight line from z_1 to z_n; these intermediate points are easy to see on a typical curve, so they are natural candidates for key points." For example:

```
draw flex(z1,z2,z3) & flex(z3,z4,z5)
flex(z5,z6,z7) & flex(z7,z8,z9,z1) & cycle;
```

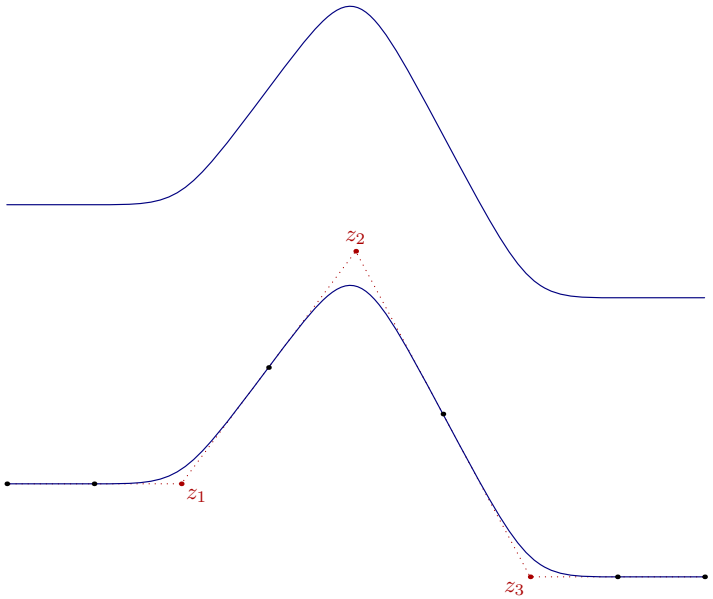
(with appropriate definitions of the points), produces this:



Another approach is just to define the end-points and some control points, and then define a path that is shaped by the control points but does not actually go through them. Consider this program:

```
vardef pulse(expr w, h, d) =
  clearxy; % protect (x,y) values
  x0 = 0; x1 = 1/4w; x2 = 1/2w; x3 = 3/4w; x4 = w;
  y0 = 0; y1 = 0; y2 = h; y3 = d; y4 = d;
  z0 .. 1/2[z0, z1] .. controls z1
     .. 1/2[z1, z2] .. controls z2
     .. 1/2[z2, z3] .. controls z3
     .. 1/2[z3, z4] .. z4
enddef;
path p; p = pulse(300, 100, -40); draw p withcolor 1/2 blue;
```

This produces the smooth blue line shown on the right. A second copy of the line is shown below, decorated with the three control points z_1, z_2, and z_3 in red, and showing the six points of the path as small black circles. You can tweak this curve by adjusting the controls left or right, or changing the mediation parameters so that the points on the path are closer to one control point than the other.



15.9 Parametric plots

If you want to plot one function against another, then you can make each coordinate a function of an independent variable. All functions can be converted, trivially, to this form:

```
vardef f(expr x) = x enddef;
vardef g(expr x) = sind(x) enddef; % or whatever function ...
path ff; ff = for t = mint step s until maxt - s:
    (f(t), g(t)) ..
endfor (f(maxt), g(maxt));
```

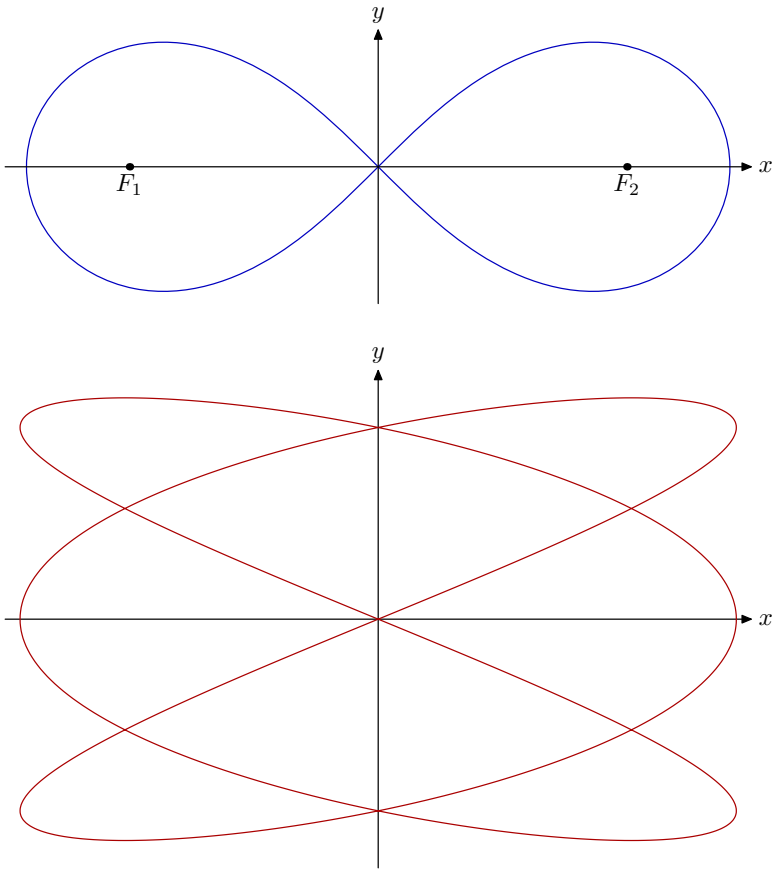
But you can make more complicated curves, for example curves that can have more than one value for y for a given x, if you change $f(x)$ and $g(x)$ appropriately. The first example shows the lemniscate of Bernoulli and was drawn like this:

```
beginfig(1);
  numeric a, c; c = 100; a = sqrt(2) * c;
  vardef f(expr x) = a * cosd(x) / (1 + sind(x) ** 2) enddef;
  vardef g(expr x) = f(x) * sind(x) enddef;
  numeric mint, maxt, s; mint = 0; s = 30; maxt = 360;
  path p; p = for t = mint step s until maxt - s:
    (f(t), g(t)) ...
  endfor cycle;
  draw p withcolor 3/4 blue;
  path xx; xx = (left -- right) scaled 150;
  path yy; yy = (down -- up) scaled 55;
  drawarrow xx; label.rt("$x$", point 1 of xx);
  drawarrow yy; label.top("$y$", point 1 of yy);
  dotlabel.bot("$F_1$", c * left);
  dotlabel.bot("$F_2$", c * right);
endfig;
```

Although sometimes, especially when you know the domain is 0° to 360° and that the path is cyclic, it is simpler to write the two expressions directly in the loop:

```
path p;
p = for t = 0 upto 360: (144 cosd(3t), 89 sind(2t)) ... endfor cycle;
draw p withcolor 2/3 red;
```

which produces this Lissajous curve ⟶

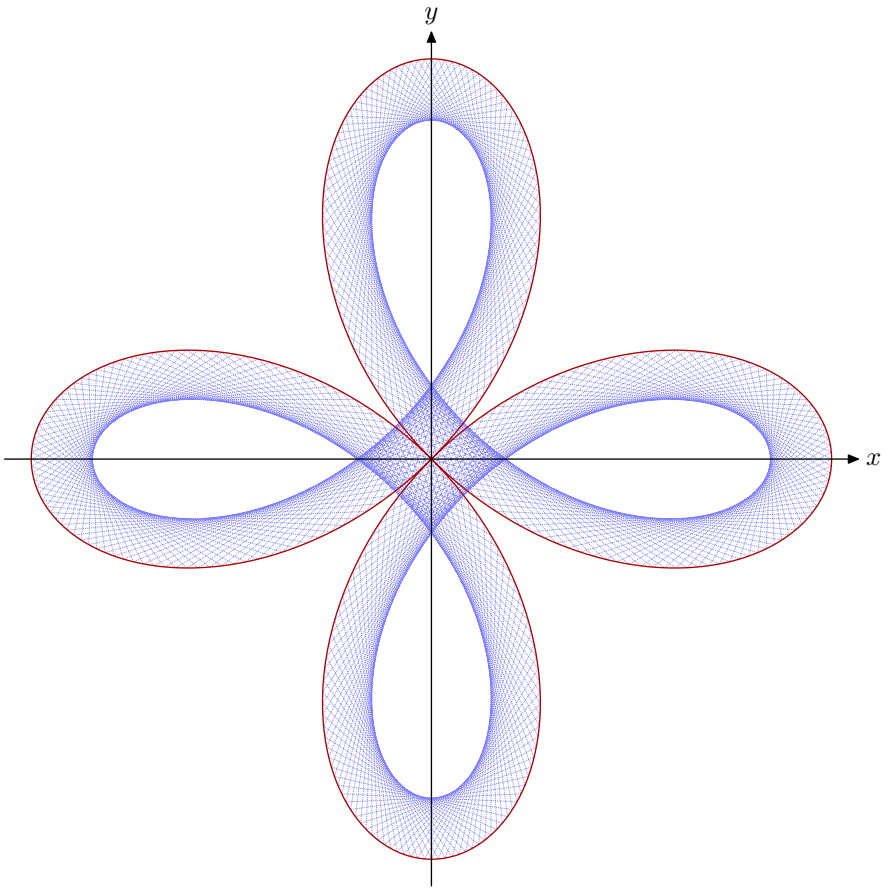


Parametric plots with polar coordinates: Maurer roses

Instead of defining separate functions for the x and y coordinates in a parametric plot, it is sometimes convenient to use METAPOST's polar coordinate notation (discussed in §6.1). The family of "rose" plots, based on $r = \cos(n\theta)$, is easy to do in this way. Here is a Maurer rose, based on $r = \cos(2\theta)$ and connecting every 29th point on the curve.

```
beginfig(1);
  path r, k;
  r = for t = 1 upto 360: cosd 2t * dir t .. endfor cycle;
  r := r scaled 150;
  k = for t = 1 upto 360: point 29t of r -- endfor cycle;

  draw k dashed withdots scaled 1/8
    withpen pencircle scaled 1/4
    withcolor 1/2[blue, white];

  draw r withcolor 2/3 red;

  path xx, yy;  % you might nor need the axes...
  xx = (left -- right) scaled 160;
  yy = (down -- up) scaled 160;
  drawarrow xx; label.rt("$x$", point 1 of xx);
  drawarrow yy; label.top("$y$", point 1 of yy);
endfig;
```

Different values of the parameters in r and k give an endless variety of patterns. But note that if you put anything higher than `point 91t of r` in k, you will need to use `-numbersystem=double` to avoid arithmetic overflow (because $360 \times 92 > 32768$). You get prettier curves if the parameter in k is a prime number.



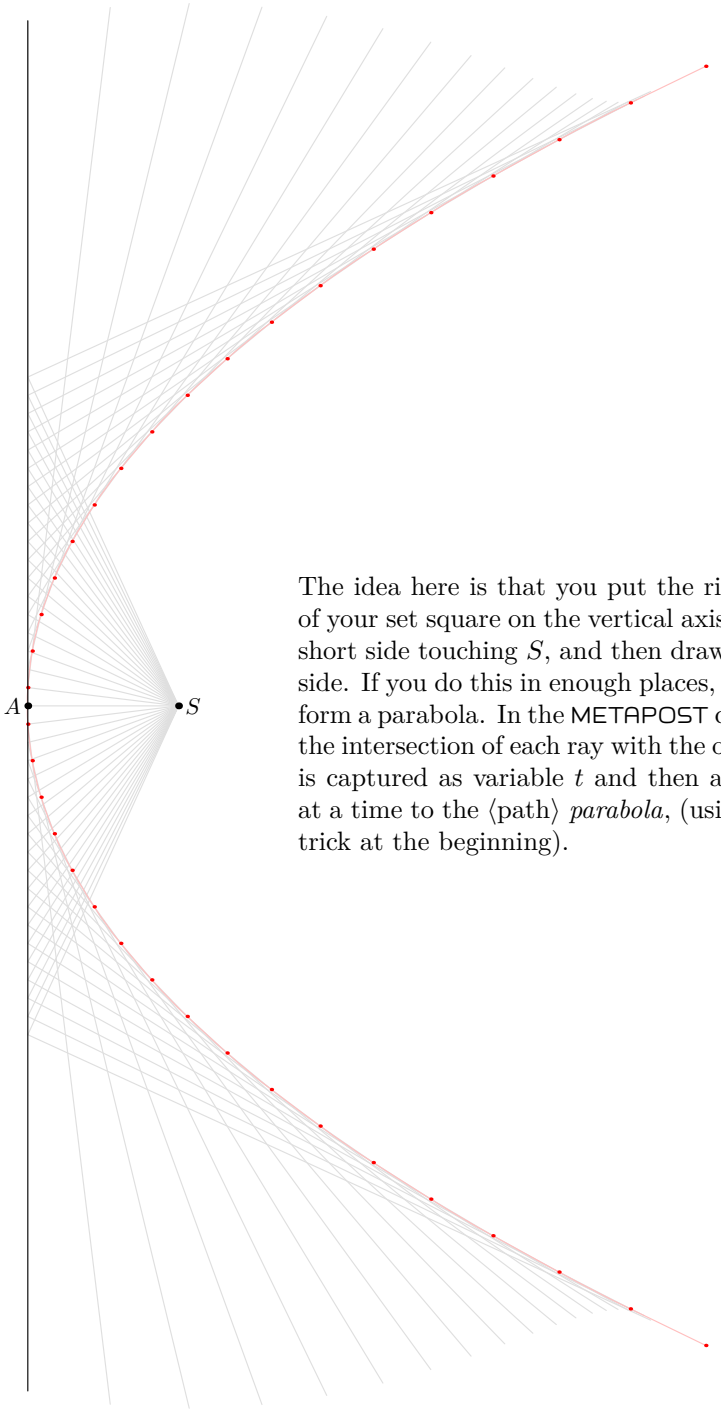
16 Drawing plane curves

> Plane curves offer a rich … field of study which may be approached from a quite elementary level. Anyone who can draw a circle with a given centre and a given radius can draw a cardioid or a limaçon. Anyone who can use a set square can draw a parabola or a strophoid — *A Book of Curves*, E. H. Lockwood

16.1 Parabola

The simplest way to get a parabola curve is to plot $y = x^2$ over $-1 \leq x \leq 1$ and then transform as required (see next page), but it can be illuminating to follow more traditional constructions, such as that shown on the right.

```
beginfig(1);
  pair A, S; A = origin; S = 66 right;
  path parabola, last;
  for q = -144 step 8 until 144:
    pair Q; Q = q * up; path ray;
    ray = Q -- Q + 300 unitvector(S - Q) rotated if q < 0: - fi 90;
    draw S -- ray withcolor 7/8;
    if known last:
      pair t; t = whatever[point 0 of ray, point 1 of ray]
                = whatever[point 0 of last, point 1 of last];
      parabola := if known parabola: parabola .. fi t;
    fi
    last := ray;
  endfor
  draw parabola withcolor 3/4[red, white];
  for t=0 upto length parabola:
    draw point t of parabola withpen pencircle scaled 3/2 withcolor red;
  endfor
  draw (up--down) scaled 300;
  dotlabel.lft("$A$", A); dotlabel.rt("$S$", S);
endfig;
```



The idea here is that you put the right angle of your set square on the vertical axis with the short side touching S, and then draw the long side. If you do this in enough places, the edges form a parabola. In the METAPOST code here, the intersection of each ray with the one before is captured as variable t and then added one at a time to the ⟨path⟩ *parabola*, (using a neat trick at the beginning).

Parabola from directrix and focus

The classical definition of the parabola is the locus of points that are equidistant from a given line (the *directrix*, shown as $A\,..\,B$ on the right) to a given focus point (shown as S). Each point on the parabola path is related to each point on the directrix, and you can construct an equilateral parallelogram at each point as shown. This leads to a macro that generates a parabola given two ⟨pair⟩ variables to define the directrix and another to define the focus:

```
vardef parabola(expr A, B, S) =
  save m, q, n, parabola;
  pair n;  % the point on A--B nearest to S
  n = whatever[A, B];
  n - S = whatever * (A-B) rotated 90;
  path parabola;
  for t=0 step 1/64 until 1:
    pair m, q;
    m = 1/2[S, t[A, B]];
    q = whatever[S, n]; q - m = whatever * (S - m) rotated 90;
    parabola := if known parabola: parabola -- fi
      q reflectedabout(S, m);
  endfor
  parabola
enddef;
```

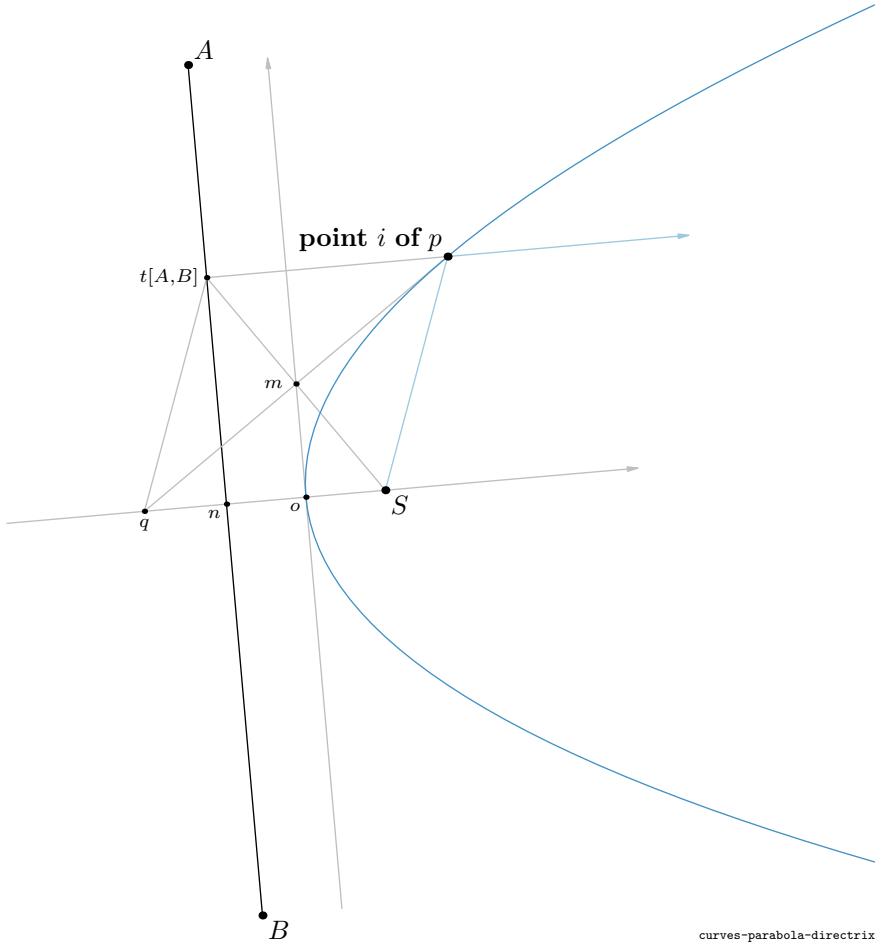
Parabola from $y = x^2$ and $dy/dx = 2x$

Alternatively you could define a "unit parabola" like this:

```
path ff; ff = (-1, 1){1, -2} .. (-1/2, 1/4){1, -1} ..
  (0, 0){right} .. (1/2, 1/4){1, 1} .. (1, 1){1, 2};
```

and then — using the points defined above, where o is the mid-point of $n\,..\,S$ — scale it and place it like this:

```
draw ff scaled 4 abs(S-o) rotated angle (B-A) shifted o;
```



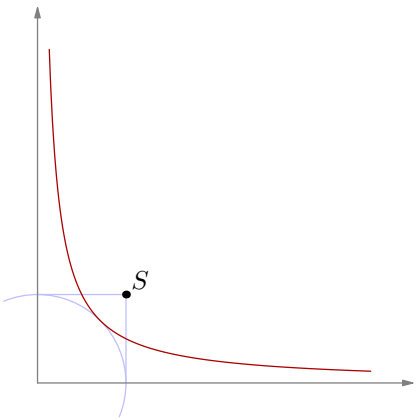
16.2 Hyperbola

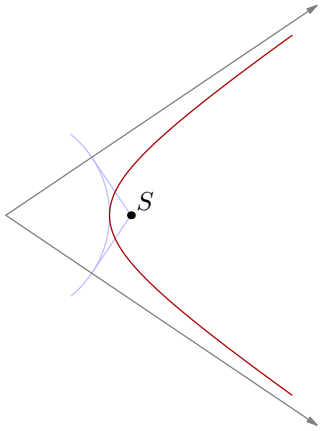
THE TRADITIONAL CONSTRUCTION for the hyperbola is identical to the construction for the parabola given above, except that the base line is a circle rather than a straight line. As a result, the shape of the curve changes depending on the radius of the base circle, unlike the parabola. The curve is bounded by the two asymptotes, which are the lines from the centre of the circle O through the tangent points from the focus S. When the ratio of $OS/OA = \sqrt{2}$, the asymptotes are at right angles.

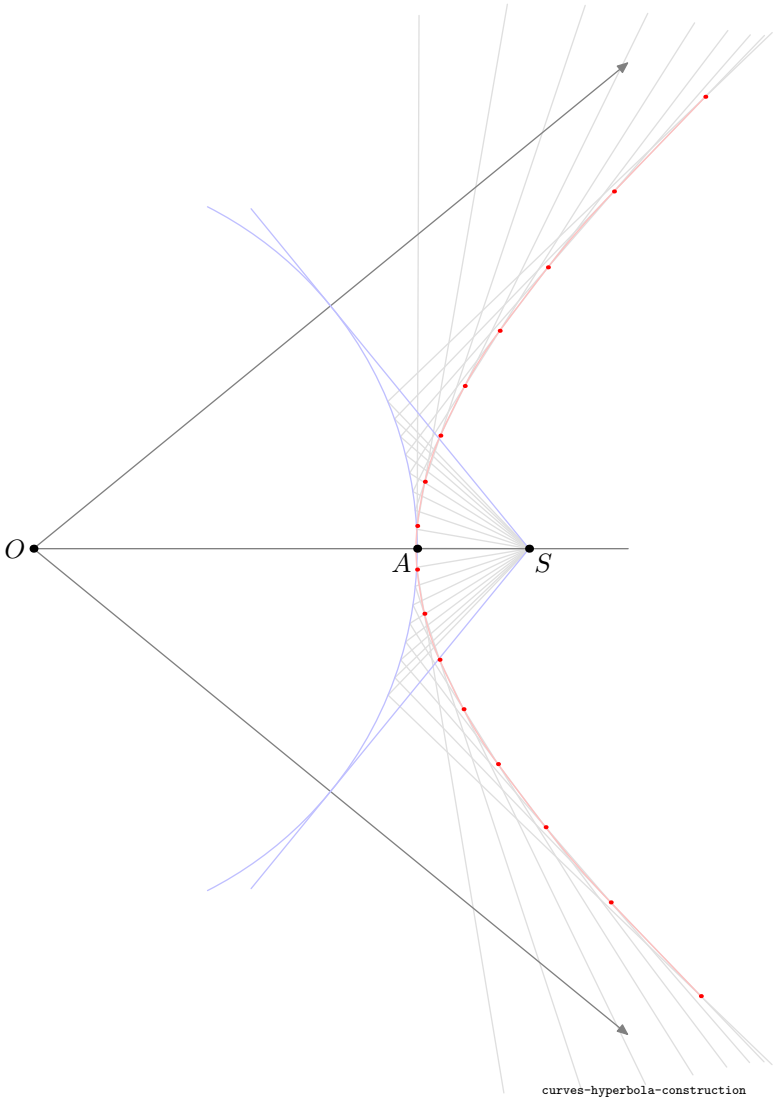
You can also draw the hyperbola as the function $y = 1/x$ (as shown in §15.4), which can be transformed to any desired shape. The untransformed function is shown on the bottom left, with the focus S at the point $(\sqrt{2}, \sqrt{2})$. If the desired angle between the asymptotes is 2α, the transformation can be created like this:

```
numeric alpha; alpha = 34; transform t;
origin transformed t = origin;
right transformed t = dir -alpha;
up transformed t = dir alpha;
```

This can be applied to the hyperbola curve itself and to the axes. But the focus will remain at the same distance from the origin, as shown below right.







16.3 Ellipse

You can draw an ellipse in METAPOST by scaling the standard `fullcircle` path by a different amount in each direction. By convention, the y-axis is the minor axis of an ellipse. The ellipse shown on the right, was defined like this:

```
path ellipse;
ellipse = fullcircle scaled 300 yscaled 5/8 rotated 13;
```

Then the lengths of the semi-major axes, a and b, were extracted like this:

```
numeric a, b;
2a = abs (point 4 of ellipse - point 0 of ellipse);
2b = abs (point 6 of ellipse - point 2 of ellipse);
```

If you already had a and b, then you could use them directly to scale your ellipse; the following snippet would produce the same elliptical path:

```
numeric a, b; a = 160; b = 100; path ellipse;
ellipse = fullcircle xscaled 2a yscaled 2b rotated 13;
```

The "eccentricity", e, of the ellipse is the ratio between the distance from the centre to each focus and the semi-major axis, a. By definition, the distance from F_1 to T to F_2 is constant as T moves round the ellipse and is equal to $2a$. Hence when T lies on the minor axis, you have $TF_1 = a$, and so $a^2 = b^2 + a^2e^2$, and $e^2 = 1 - b^2/a^2$:

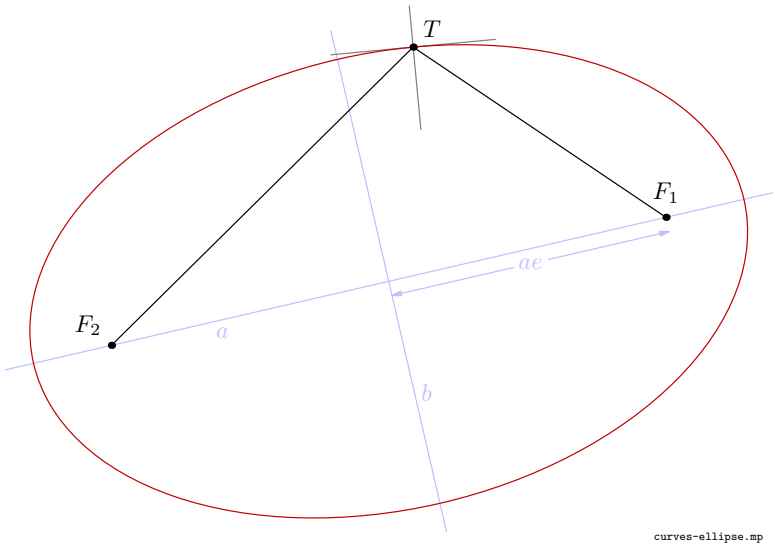
```
numeric e; e = 1 +-+ b/a;  % Pythagorean difference
```

The focus points can then be found like this:

```
z0 = 1/2[point 0 of ellipse, point 4 of ellipse];
z1 = e[z0, point 0 of ellipse];
z2 = e[z0, point 4 of ellipse];
```

Note that in the example above the ellipse is centred at the origin, because the `fullcircle` path was only scaled and rotated, not shifted. But the method shown will find the centre of any ellipse no matter where you have placed it.

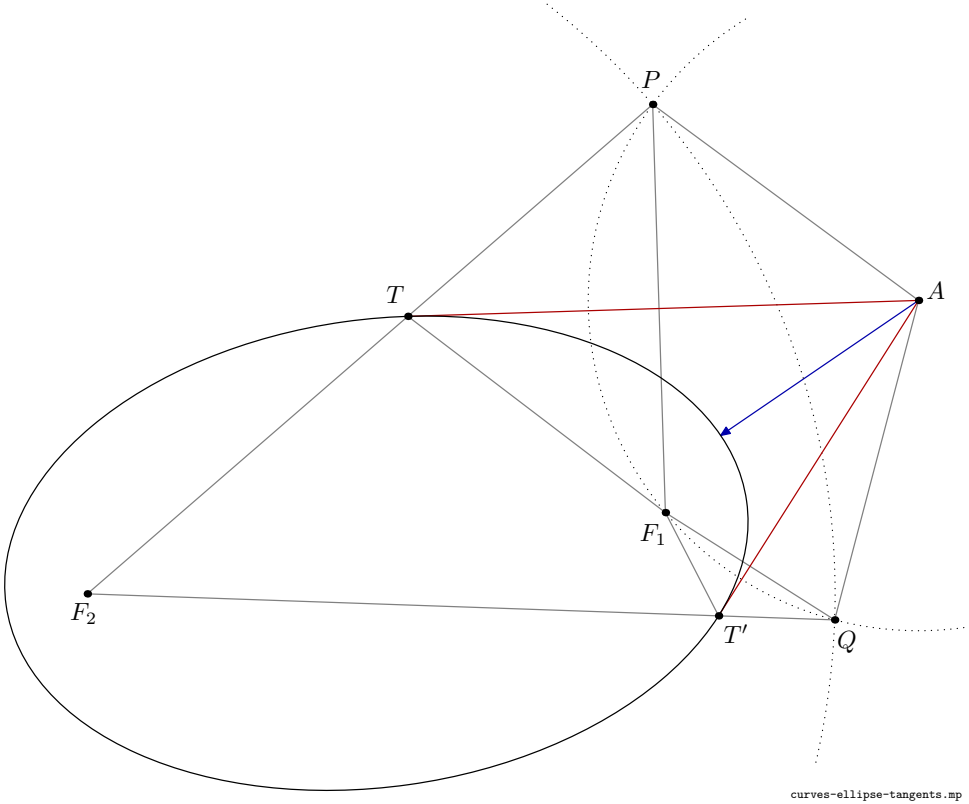
☞ If you already have the two focus points, then the distance between them will be $2ae$, so if you also have a or e you can calculate b and hence draw your ellipse appropriately scaled, rotated, and shifted.



The tangent and the normal at T above were added like this:

```
numeric t; t = 1.732;
drawoptions(withcolor 1/2);
draw (left--right) scaled 34
  rotated angle direction t of ellipse
  shifted point t of ellipse;
draw (1/2 down--up) scaled 34
  rotated angle direction t of ellipse
  shifted point t of ellipse;
```

Tangent from external point to ellipse



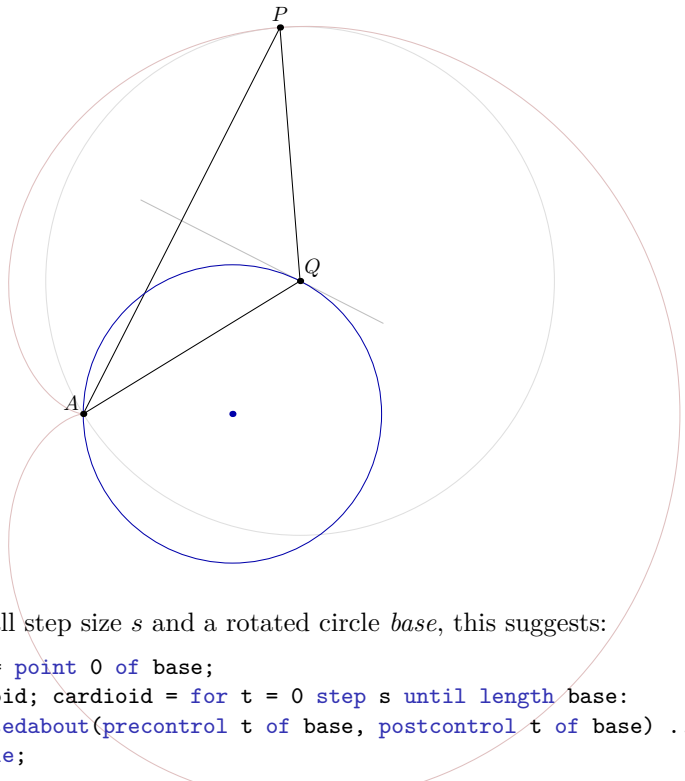
To find the tangent points from an external point A to an ellipse, the classical construction is to draw an arc centred at A through one focus point, then draw a second arc centred at the other focus with radius $2a$. The intersection points, P and Q, of these two arcs are the images of the first focus point in the required tangents (because $F_2TF_1 = 2a$ by definition, and $F_2P = 2a$ by construction), and so the tangent points T and T' are the intersections of F_2P and F_2Q with the ellipse.

There is no such direct construction for the nearest point on an ellipse to a given point, but you can use the macro *solve* to find it numerically. The blue arrow shows the shortest distance from A to the ellipse.

```
path ellipse; ellipse = fullcircle scaled 300 yscaled 5/8 rotated 8;
2a = abs (point 0 of ellipse - point 4 of ellipse);
2b = abs (point 2 of ellipse - point 6 of ellipse);
e = 1 +-+ b/a;

pair F[], A;
F1 = (1/2-e/2)[point 0 of ellipse, point 4 of ellipse];
F2 = (1/2+e/2)[point 0 of ellipse, point 4 of ellipse];
A = 240 dir 25;

path pp; pp = fullcircle scaled 2 abs (F1-A) shifted A;
path qq; qq = fullcircle scaled 4a shifted F2;
pair P, Q, T, T';
z1 = pp intersectiontimes qq; P = point x1 of pp;
z2 = reverse pp intersectiontimes qq; Q = point -x2 of pp;
z3 = ellipse intersectiontimes (F2 -- P); T = point x3 of ellipse;
z4 = ellipse intersectiontimes (F2 -- Q); T' = point x4 of ellipse;

vardef f(expr x) =
  angle (A-point x of ellipse) + 90 > angle direction x of ellipse
enddef;
drawarrow A -- point solve f(0, x3) of ellipse withcolor 2/3 blue;

draw subpath (x1 - 1/2, 17/2 - x2) of pp dashed withdots scaled 1/2;
draw subpath (y2 - 33/4, y1 + 1/4) of qq dashed withdots scaled 1/2;
draw T -- F1 -- P -- F2 -- Q -- F1 -- T' withcolor 1/2;
draw P -- A -- Q withcolor 1/2;
draw T -- A -- T' withcolor 2/3 red;
draw ellipse;

def dotlabelx(expr t, z, o) =
  draw z withpen pencircle scaled dotlabeldiam; label(t, z + o);
enddef;
dotlabelx("$F_1$", F1, 10 dir 241);
dotlabelx("$F_2$", F2, 8 dir 260);
dotlabelx("$A$", A, 8 dir 30);
dotlabelx("$P$", P, 10 dir 94);  dotlabelx("$Q$", Q, 10 dir 300);
dotlabelx("$T$", T, 10 dir 120); dotlabelx("$T'$", T', 10 dir -45);
```

16.4 Cardioid

To draw a cardioid by hand, you can draw a base circle, mark a fixed point A on it, and then draw a circle centred at any point Q on the circle that passes through point A. If you then repeat this for many different positions of Q, the cardioid is the curve that encloses all the circles. But for METAPOST, you want only a single point P from the circumference of each circle; this turns out to be the image of A reflected in the tangent at each point Q, like so:

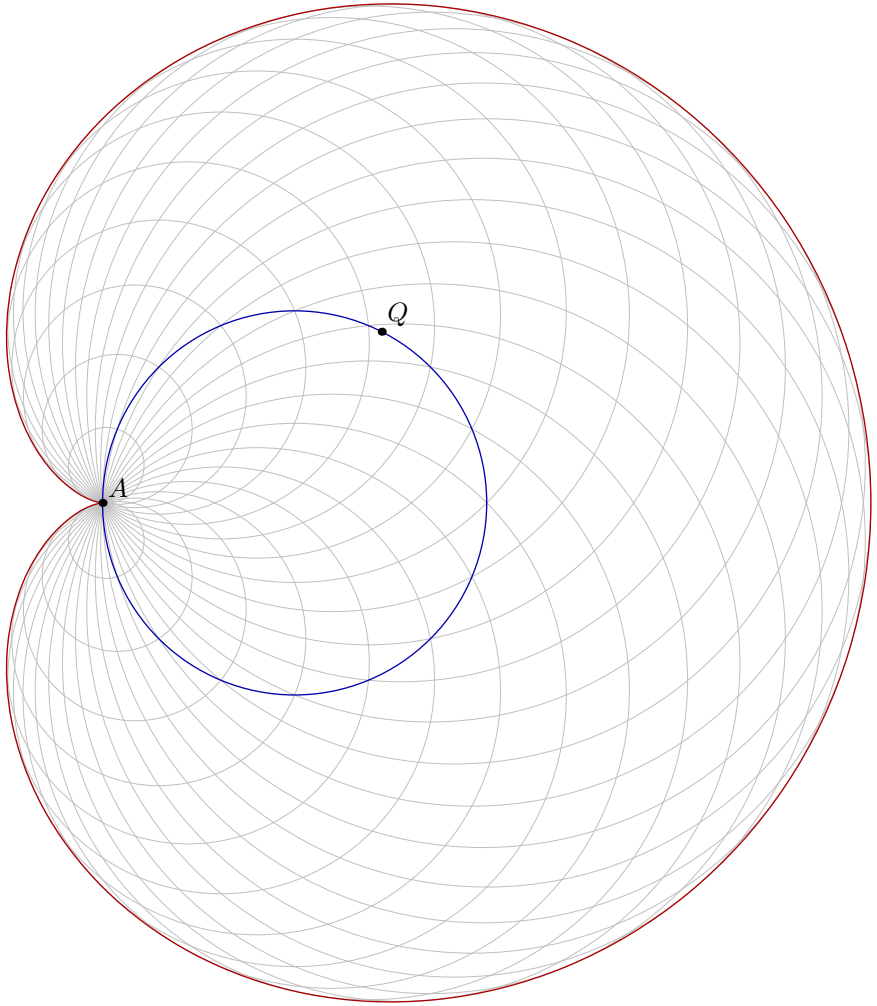


With a small step size s and a rotated circle *base*, this suggests:

```
pair A; A = point 0 of base;
path cardioid; cardioid = for t = 0 step s until length base:
  A reflectedabout(precontrol t of base, postcontrol t of base) ..
endfor cycle;
```

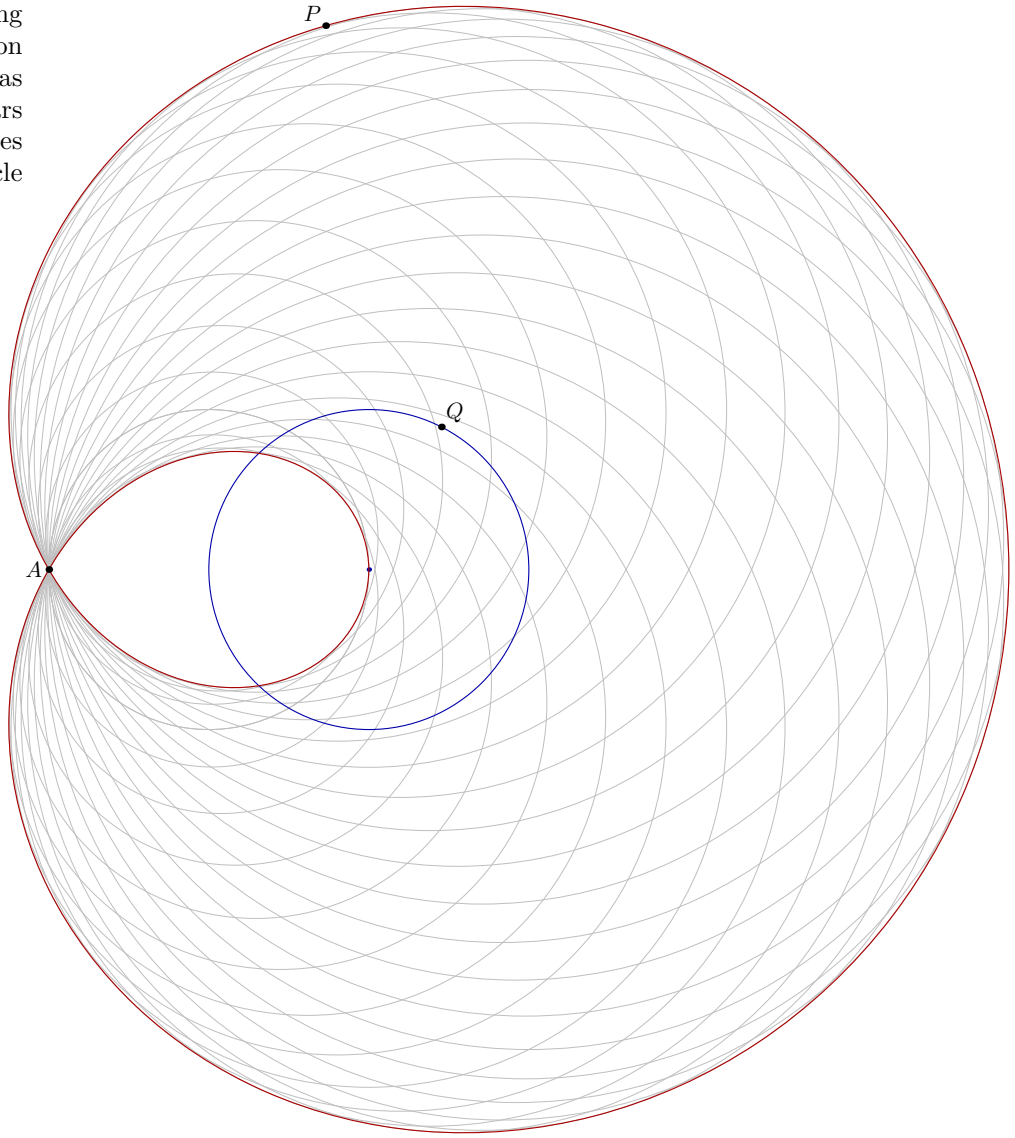
You can also show that $AP = 2a(1+\cos\theta)$, where a is the radius of the base circle and θ is the angle that AP makes with the diameter through A, so you might use:

```
cardioid = for t=0 upto 360: 2a * (1+cosd(t)) * dir t .. endfor cycle;
```



16.5 Limaçon

The limaçon can be seen as a generalization of the cardioid, obtained by moving point A off the base circle. Here A has been moved to the left, but each P on the curve is still A reflected in the tangent at each Q. The "hole" gets larger as A moves away from the *base* circle; when A touches the *base* the hole disappears and the curve becomes the cardioid, as before. Following this ruler-and-compasses approach, the red limaçon path in the figure here was generated from the *base* circle shown in blue.



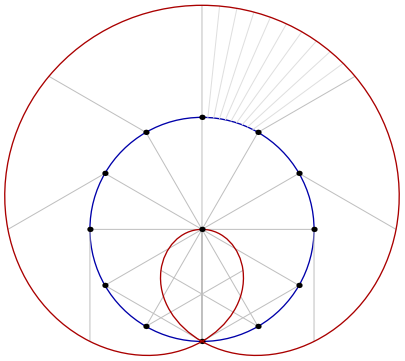

```
pair A; A = 2[center base, point 0 of base];
path limacon; limacon = for t = 0 step s until length base:
  A reflectedabout(precontrol t of base, postcontrol t of base) ..
endfor cycle;
```

Or if you prefer a more trigonometrical approach:

```
limacon = for t=0 upto 359: 2a*(1+2cosd(t))*dir t .. endfor cycle;
```

Here $2a$ is the diameter of the blue *base* circle. Note that if you use `sind` instead of `cosd` you get the same curve rotated 90°.

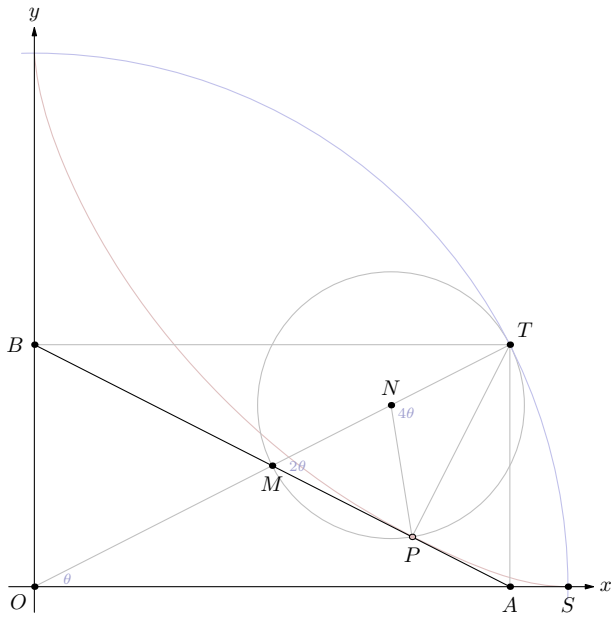
An alternative approach (due to Albrecht Dürer) is shown below. In this diagram, the base circle is divided into 12 parts like a clock face. At 1 o'clock, you draw a line segment of a given length parallel to the radius to 2 o'clock; at 2 you draw the same length segment parallel to 4 o'clock, and so on. The limaçon is the curve through the far ends of each segment (plus any intermediate points required). This approach of doubling the angles makes it more obvious that the limaçon goes round twice, as it were. The path was generated like this:



```
limacon = for t=0 upto length base-1:
   42 dir angle point 2t of base shifted point t of base ..
endfor cycle;
```

16.6 Astroid

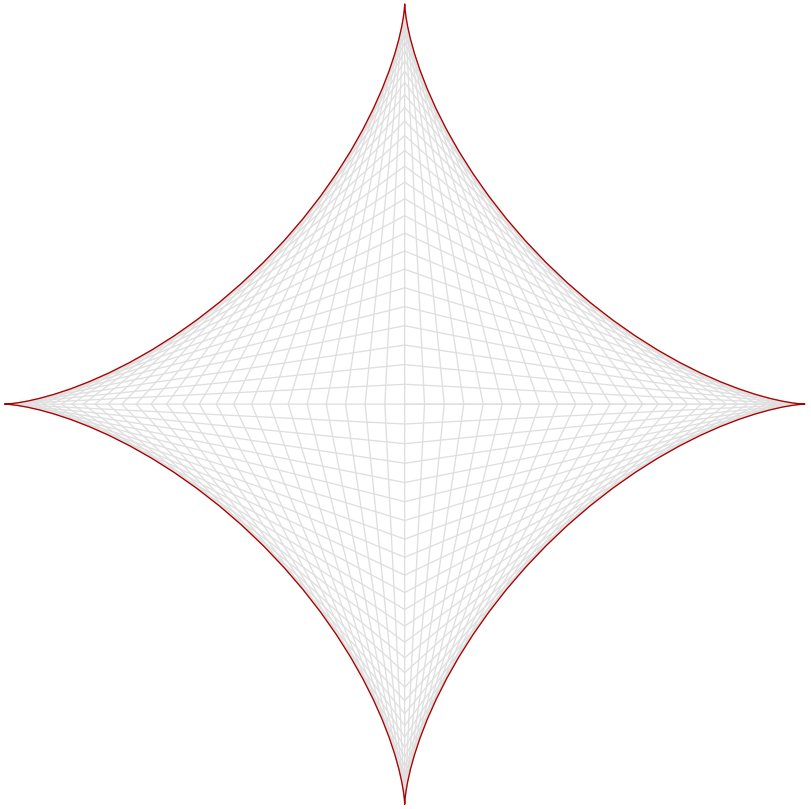
Readers of a certain age may recall threading strings between pegs on a board to make the astroid. It is the envelope of a line of a given length drawn from x-axis to y-axis at all possible points. In METAPOST the simplest way to draw the astroid curve (and the "strings") is to use a base circle and the points A and B at the ends of each line that have the x part and y part of each point T round the base circle.



Then the point P on A .. B that is closest to T will lie on the astroid, so you can make the path with:

```
path astroid; astroid = for t=0 step 1/16 until 8:
  hide(pair a, b, p;
    a = (xpart point t of base, 0); b = (0, ypart point t of base);
    p = whatever[a,b]; p - point t of base = whatever * (a-b) rotated 90;
  ) p -- endfor cycle;
```

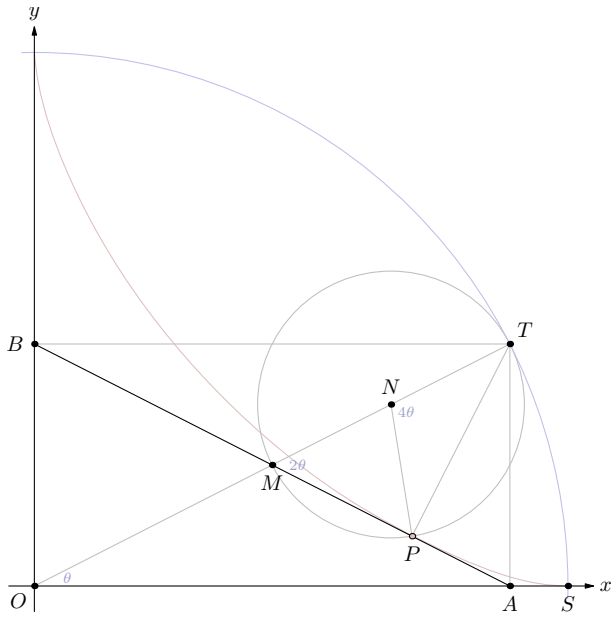
Note that you need to use "`--`" so that the cusps stay neatly pointed.



☞ The geometry of the subtended angles shows that the length of the arc T .. S equals the length of the arc from T .. P on the quarter-sized circle through T and M. So the astroid is also the path of a point on the smaller circle rolling around the inside of the base circle.

Astroid and cousins

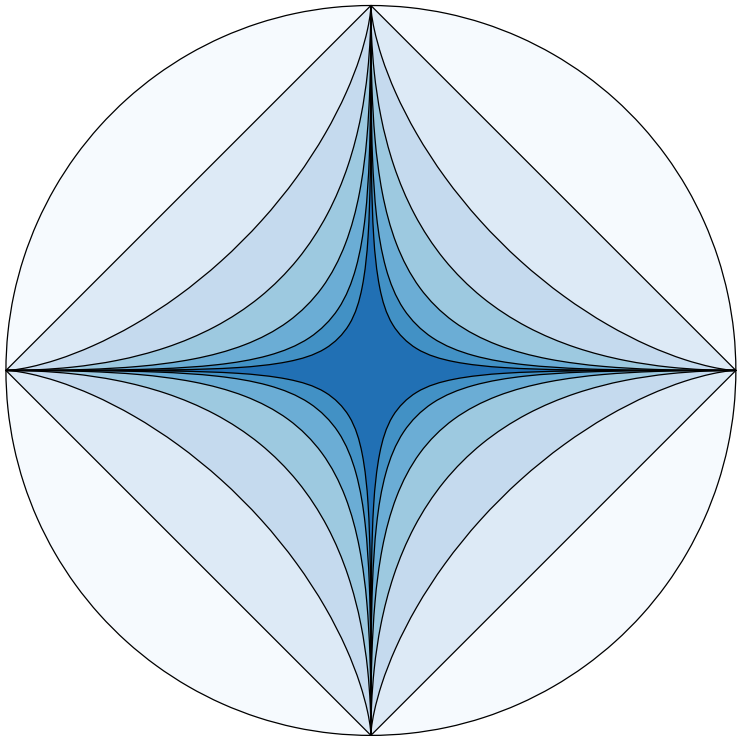
The geometry of the astroid also allows us to define a simple parametric equation for the point P.



If the distance $OT = a$, then $OA = BT = a\cos\theta$. But then $BP = BT\cos\theta = a\cos^2\theta$, and the x-coordinate of $P = BP\cos\theta = a\cos^3\theta$. By a similar argument the y-coordinate is $a\sin^3\theta$, so the parametric equations for $P = (x, y)$:

$$x = a\cos^3\theta \quad \text{and} \quad y = a\sin^3\theta$$

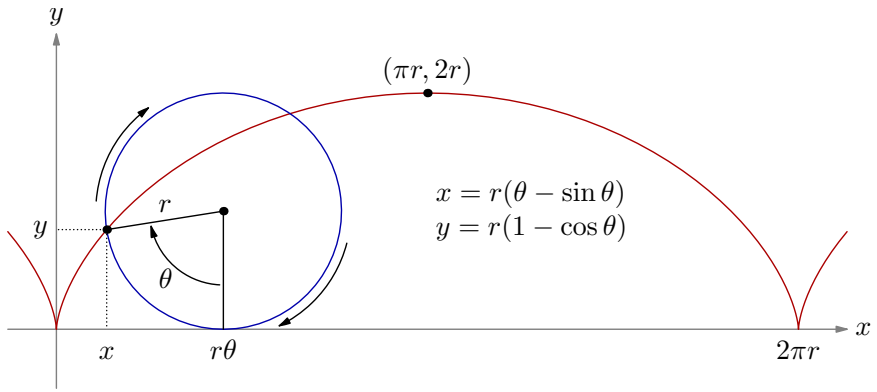
This is used to make this rather psychedelic family of astroid cousins ⟶



```
input colorbrewer-rgb
for n=1 upto 7:
  path p; p = (right for t=6 step 6 until 90:
    .. (cosd(t) ** n, sind(t) ** n)
  endfor) scaled 144;
  p := for i=0 upto 3: p rotated 90i & endfor cycle;
  fill p withcolor Blues[9][n]; draw p;
endfor
```

16.7 Cycloid

Cycloids are the curves made by points on the circumference of a rolling wheel. In the first diagram the cycloid is drawn in red and the corresponding rolling wheel in blue. The main idea in this diagram is to make the whole drawing depend on just a few parameters; here there are two: the radius r and the amount of rotation θ. If we make r bigger, the drawing will be scaled up; if we change θ, the wheel will appear to have rolled along.

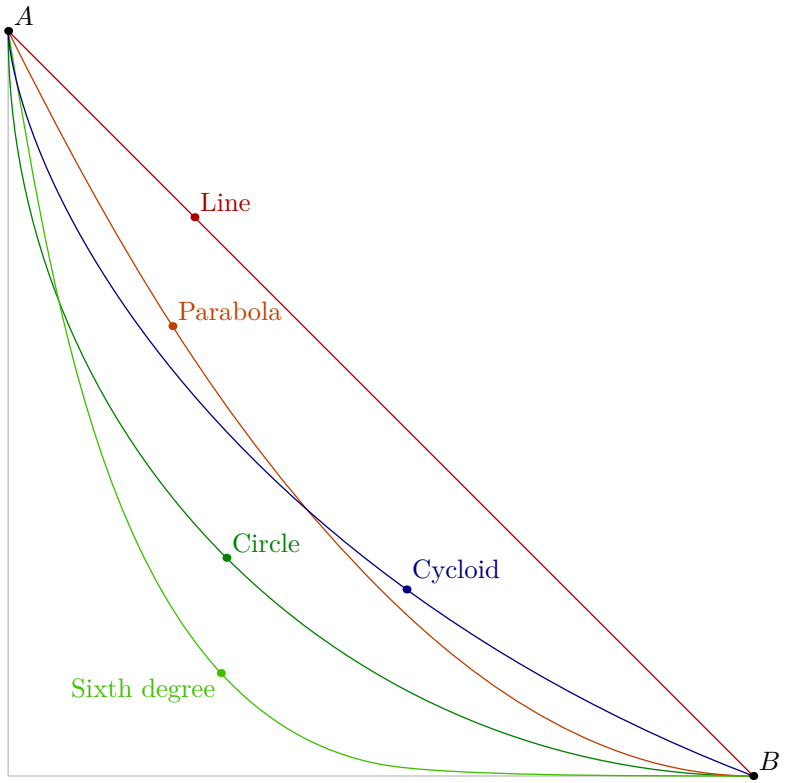


- The path of the cycloid is defined using an inline `for` loop, using a neat trick to avoid a leading `--` in the path. The strange numbers here are because we are going from a rotation of −80° to +440°; 360° corresponds to one hop of the cycloid.
- The axes are done in the usual way, except that we use `xpart` and the `point .. of ..` notation to make the x-axis neatly line up with the ends of the cycloid path.
- To label points with dots but no text it's convenient just to draw the point with `pencircle scaled dotlabeldiam`; this internal parameter is the current size to be used for the dots in `dotlabel`.

```
beginfig(1);
  numeric pi, r, theta; pi = 3.141592653589793; r = 42; theta = 81;
  path cycloid;
  for t=-80 step 5 until 440:
    cycloid := if known cycloid: cycloid -- fi
        (0, -r) rotated -t shifted (t/180*pi*r, r);
  endfor;
  path wheel;
  z1 = (theta/180*pi*r, r);
  wheel = reverse fullcircle scaled 2r rotated -(90 + theta) shifted z1;
  z2 = point 0 of wheel;
  path a[]; u = 1/16; % u = a little shortening
  a1 = subpath -(theta/45-u,u) of wheel shifted -z1 scaled 5/8 shifted z1;
  a2 = subpath (.3, 1.4) of wheel shifted -z1 scaled 1.08 shifted z1;
  a3 = a2 rotatedabout(z1, 170);
  path xx, yy;
  xx = (xpart point 0 of cycloid, 0) -- (xpart point infinity of cycloid,0);
  yy = (down -- 5 up) scaled 1/2 r;

  ahangle := 30;
  drawarrow xx withcolor 1/2; label.rt (btex $x$ etex, point 1 of xx);
  drawarrow yy withcolor 1/2; label.top(btex $y$ etex, point 1 of yy);

  draw cycloid withcolor .67 red;
  draw wheel withcolor .67 blue;
  drawarrow a1; drawarrow a2; drawarrow a3;
  draw (0,y2) -- z2 -- (x2,0) dashed withdots scaled 1/4;
  draw z2 -- z1 -- (x1,0);
  draw z1 withpen pencircle scaled dotlabeldiam;
  draw z2 withpen pencircle scaled dotlabeldiam;

  label(btex $\theta$ etex, z1 + 3/4r * dir (270 - 1/2 theta));
  label.bot(btex $\mathstrut 2\pi r$ etex, (2pi*r,0));
  label.bot(btex $\mathstrut r\theta$ etex, (x1,0));
  label.bot(btex $\mathstrut x$ etex, (x2,0));
  label.lft(btex $y$ etex, (0,y2));
  label.top(btex $r$ etex, 1/2[z1, z2]);
  dotlabel.top(btex $(\pi r,2r)$ etex, (pi*r,2r));
  label.rt(btex $\vcenter{\halign{&$#$\hfil\cr
    x=r(\theta-\sin\theta)\cr
    y=r(1-\cos\theta)\cr}}$ etex, (pi*r,r));
endfig;
```

The cycloid compared to other curves

You can't easily draw a cycloid through two arbitrary points, but taking the equations for x and y from the previous page, we can use `solve` to find a value a for $\theta > 0$ where $\theta - \sin\theta = 1 - \cos\theta$, and then you have two points for $\theta = 0$ and $\theta = a$ which are at each end of a quarter circle, and it's easy to draw other curves through them. The cycloid is drawn here inverted, to make the *brachistochrone* (the curve joining two points such that a body travelling along it under gravity takes a shorter time than is possible along any other curve between the points).




```
beginfig(1);
  path Y, L, C, P, S;

  % solve parameters for cycloid
  vardef u(expr x) = x - sind(57.295779513 x) enddef;
  vardef v(expr x) = 1 - cosd(57.295779513 x) enddef;
  vardef f(expr t) = u(t) < v(t) enddef;
  tolerance := epsilon; a = solve f(2,3);
  s = 1/64 a;
  Y = (origin for t = s step s until a+eps:
    -- (u(t), -v(t))  % negative v(t) so curve is inverted
  endfor) scaled 160;
  z0 = point 0 of Y;
  z1 = point infinity of Y;

  % define the four other paths
  L = z0 -- z1;
  C = quartercircle rotated 180 scaled 2x1 shifted (x1, y0);
  P = z0{1,-2} ... (1/2[x1, x0], 1/4[y1, y0]){1,-1} ... z1 {1, 0};
  S = z0{1,-6} ... (1/2[x1, x0], 1/64[y1, y0]){1, -6/32} ... z1 {1, 0};

  draw z0 -- (x0,y1) -- z1 withcolor 3/4;
  drawoptions(withcolor 2/3 red);
    draw L; dotlabel.urt("Line", point 1/4 of L);
  drawoptions(withcolor 1/2 green);
    draw C; dotlabel.urt("Circle", point 1 of C);
  drawoptions(withcolor 1/4[red, green]);
    draw P; dotlabel.urt("Parabola", point 1/2 of P);
  drawoptions(withcolor 3/4[red, green]);
    draw S; dotlabel.llft("Sixth degree", point 3/4 of S);
  drawoptions(withcolor 1/2 blue);
    draw Y; dotlabel.urt("Cycloid", point 50 of Y);
  drawoptions();

  dotlabel.urt("$A$", z0);
  dotlabel.urt("$B$", z1);
endfig;
```

16.8 Spirals

D'Arcy Thompson tells us, in *On Growth and Form*, that the spiral of Archimedes "may be roughly illustrated by the way a sailor coils a rope upon the deck; as the rope is of uniform thickness, so in the whole spiral coil is each whorl of the same breadth as that which precedes and as that which follows it". In mathematical terms the radius of the spiral is proportional to the angle turned, so that $r = a\theta$. This is very simple to program in METAPOST.

```
numeric a; a = 1/8; path S;
S = origin for t=1 upto 360: .. a * t * dir t endfor;
```

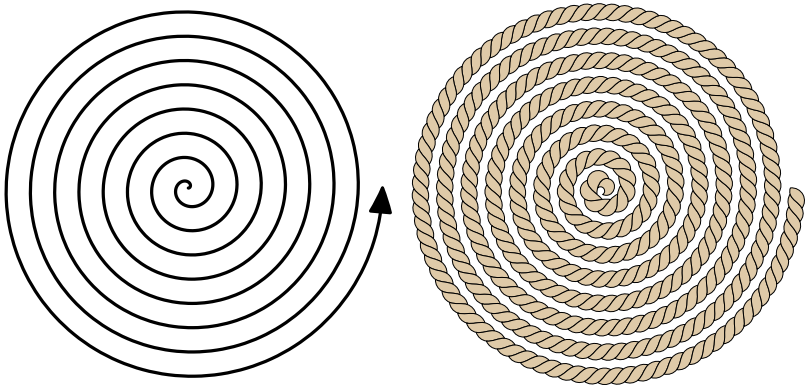
except that you are unlikely to need one point for every degree of turn in your spiral, so you are more likely to code:

```
S = origin for t=1 upto 360: .. a * t * dir 8t endfor;
```

which spreads the points out and gives you eight full turns, or perhaps

```
S = origin for t=1 upto 90: .. 1/12 t * dir 16t endfor;
```

which would give you four complete turns with a tighter spacing.

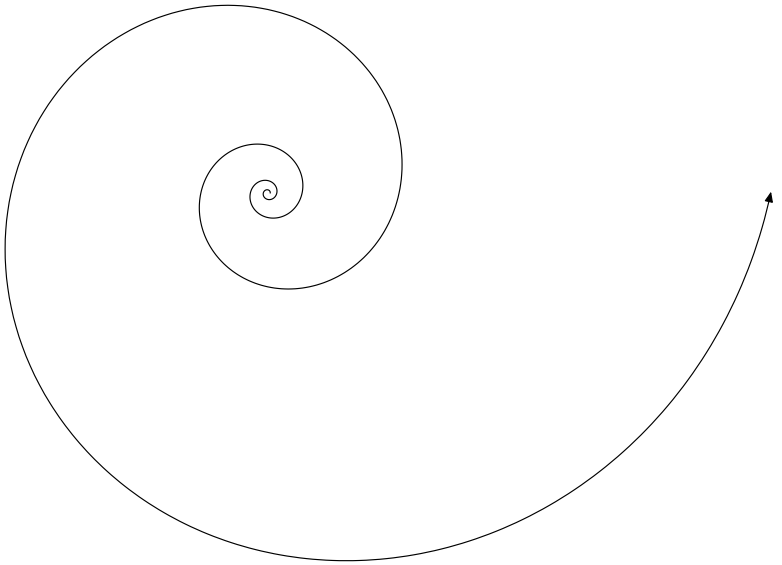


The rope was drawn (quite slowly) with the `rope` macro from §13.10.

The next simplest is the logarithmic spiral where you have $r = a^\theta$. This is also very simple to program in METAPOST, provided you are careful about the scaling. For the spiral shown on the right, the complete program was:

```
numeric a; a = 2.6; path S;
S = right for t=1 upto 360: .. a ** (t/64) * dir 4t endfor;
drawarrow S;
```

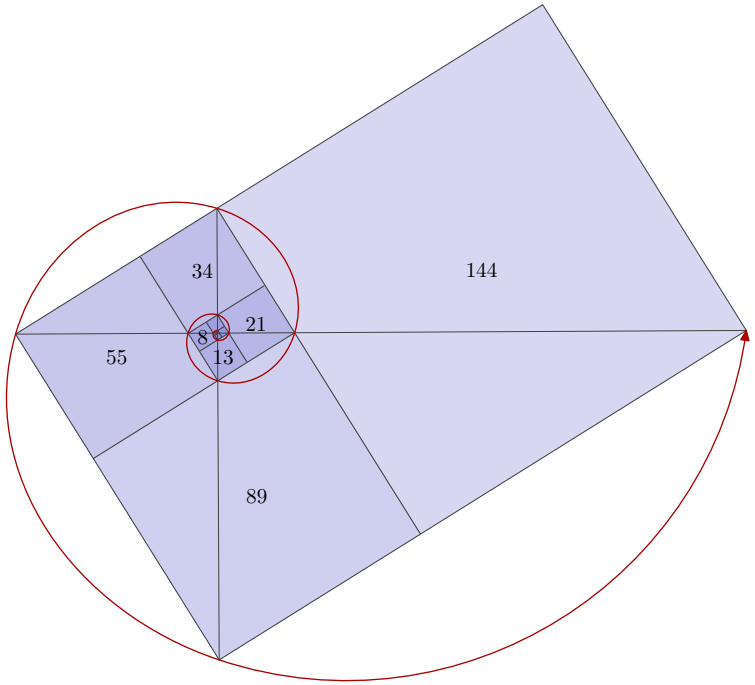
Note that a was carefully chosen to get a curve that would fit the page, and that t has been divided by 64 to bring it into a suitable range to work with the default number system.



Logarithmic spiral and the golden rectangle

The logarithmic spiral is connected to growth in nature. If you start with a small square and keep adding squares scaled to the longer side of the resulting rectangle, you get the golden rectangle and the logarithmic spiral emerges from it.

```
beginfig(1);
  drawoptions(withpen pencircle scaled 1/4 withcolor 1/4);
    path s[]; s0 = unitsquare;
    fill s0 withcolor 1/2[2/3 blue, white]; draw s0;
    numeric a, b, t, n; a = 1; b = 1; n = 11;
    for i = 1 upto n:
      t := b; b := b + a; a := t;  % Fibonacci sequence
      s[i] = unitsquare scaled a;
      s[i] := s[i] shifted (point i of s[i-1] - point i-1 of s[i]);
      fill s[i] withcolor (1/2 + i/32)[2/3 blue, white];
      draw s[i];
    endfor
    % cross hairs
    for i = n-1 upto n:
      draw point i-2 of s[i-2] -- point i of s[i];
    endfor
  drawoptions();
  % Draw the spiral as a red arrow
  drawarrow origin for i=0 upto n: .. point i of s[i] endfor
    withcolor 2/3 red;
  % show the Fibonacci sizes for the larger boxes
  for i = 5 upto n:
    label(TEX(decimal arclength subpath (0, 1) of s[i])
      scaled 0.8 rotated angle point n of s[n], center s[i]);
  endfor
  % Rotate the whole picture to show off the spiral
  currentpicture := currentpicture rotated - angle point n of s[n];
endfig;
```

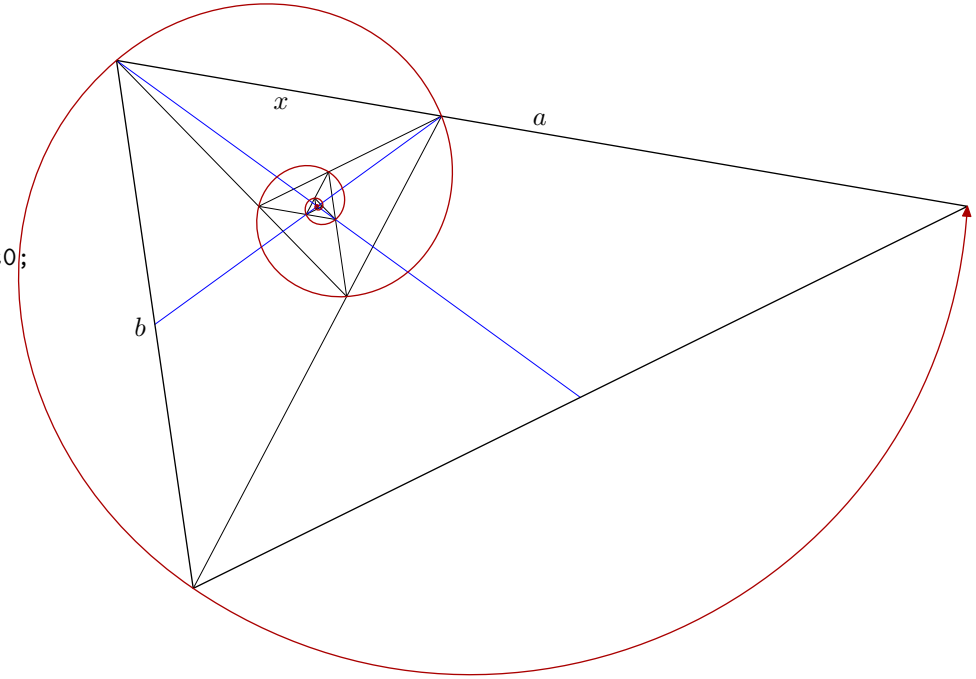


- You can't assign to a pair literal in METAPOST, so you cannot write `(a,b) := (b,a+b)`; use a temporary numeric instead.
- The uses of `point` with the square paths `s[]` exploit the fact the the `unitsquare` path is cyclic, so point 4 is the same as point 0, and so on.
- Note also the rotation of the labels, so that they are horizontal when the whole picture is rotated at the end to show the spiral better.

Logarithmic spiral and similar triangles

THIS DRAWING starts with a large triangle and transforms it to make smaller similar copies. The spiral is drawn through the apex of successive transformed triangles.

```
beginfig(1);
  path t[], base; pair apex;
  base = (left--right) scaled 100;
  apex = whatever * dir +72 shifted point 0 of base
       = whatever * dir -72 shifted point 1 of base;
  t0 = (base -- apex -- cycle);
  transform S;
  r = arclength subpath (0, 1) of t0 / arclength subpath (2, 3) of t0;
  point 0 of t0 transformed S = (r*r)[point 0 of t0, point 2 of t0];
  point 1 of t0 transformed S = point 0 of t0;
  point 2 of t0 transformed S = point 1 of t0;
  n = 16;
  for i=1 upto n:
    t[i] = t[i-1] transformed S;
    draw subpath (2,3) of t[i] withpen pencircle scaled 1/4;
  endfor
  drawoptions(withpen pencircle scaled 1/8 withcolor blue);
    draw point 0 of t0 -- point 3/2 of t0;
    draw point 0 of t1 -- point 3/2 of t1;
  drawoptions();
  draw t0;
  drawarrow point 2 of t[n] for i=n-1 downto 0:
    .. point 2 of t[i] endfor withcolor 2/3 red;
  z0 = whatever[point 0 of t0, point 3/2 of t0]
     = whatever[point 0 of t1, point 3/2 of t1];
  numeric a; a = angle (point 2 of t0 - z0);
  label.lft(TEX("$a$") rotated a, point 5/2 of t0);
  label.bot(TEX("$b$") rotated a, point 1/2 of t0);
  label.rt(TEX("$x$") rotated a, point 1/2 of t1);
  currentpicture := currentpicture rotated -a;
endfig;
```



The ⟨transform⟩ S is defined implicitly. It is sufficient to give equations for three non-collinear points, and METAPOST will work out the rest. In order to do this, you need to know the ratio of x/a, but by definition $a/b = b/x$, so $x = b^2/a$, hence $x/a = (b/a)^2$. The code shown sets $r = b/a$, and then uses r^2 as the required fraction along a.

☞ This construction also works for other triangles, but it looks more elegant with an isosceles with base angles of 72° as shown.

17 Eggs

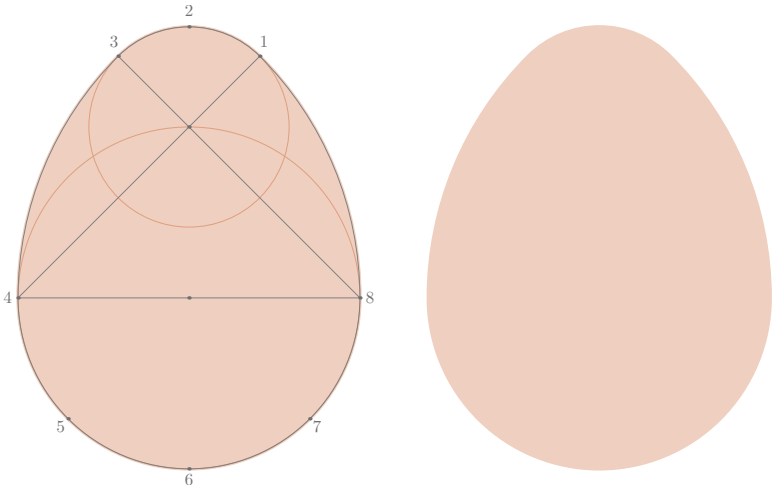
Drawing hens' eggs excites a curious fascination in some people. This section shows some possible ways to make eggs with METAPOST. The first few compass-and-ruler constructions are taken from Robert Dixon's *Mathographics*. They all follow the same basic idea of constructing the egg-shaped path from a series of circular arcs.

17.1 Euclidean egg

The first is made of four circular arcs, defined here as parts of circles a, b, c, & d.

```
input eggs-common
path a, b, c, d, egg; numeric r; r = 100;
a = fullcircle scaled 2r;
b = fullcircle scaled 4r shifted point 4 of a;
c = fullcircle scaled 4r shifted point 0 of a;
d = fullcircle scaled 2 abs (point 2 of a - point 1 of b)
              shifted point 2 of a;
egg = subpath (0, 1) of b .. point 2 of d ..
      subpath (3, 4) of c .. subpath (5, 7) of a .. cycle;

beginfig(1);
  fill egg withpen pencircle scaled 2 withcolor eggshell;
  picture plain_egg; plain_egg = currentpicture;
  drawoptions(withpen pencircle scaled 1/4 withcolor dark_eggshell);
    draw a; draw d;
  drawoptions(withpen pencircle scaled 1/4 withcolor 1/2);
    draw point 1 of egg -- point 4 of egg --
         point 0 of egg -- point 3 of egg;
    draw egg;
  drawoptions(withpen pencircle scaled 2 withcolor 7/16);
    draw center a; draw center d;
    draw numbered_points(egg);
  drawoptions();
  draw plain_egg shifted 240 right;
endfig;
```

In these METAPOST implementations each egg path has 8 points starting with point 0 at "3 o'clock" like `fullcircle`.

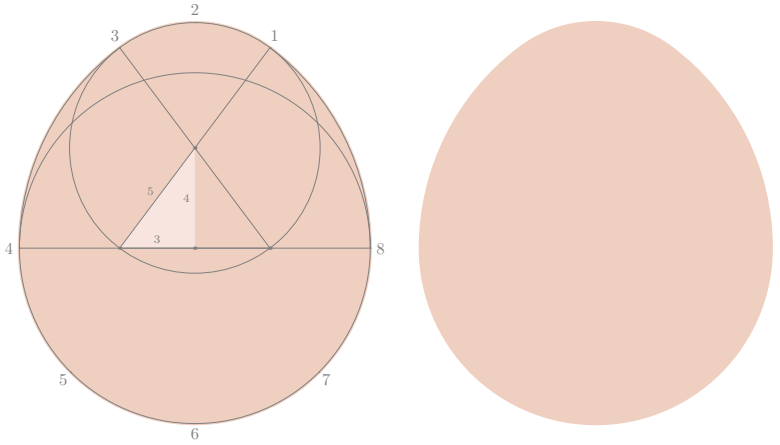


- `eggs-common.mp` defines the colours, and the `numbered_points` routine that is used to show the points of the `egg` path.
- Note that it is not necessary that the parts of the arcs touch; in fact it is better to join them with the `..` connector in case the ends are not close enough for you to use `&`.
- The rigmarole with saving the current picture, is to show a copy of the egg path with and without the construction lines.

17.2 Pythagorean egg

The centres of the arcs are determined by the 3-4-5 triangle at the origin.

```
numeric r, a, b, t; a = 60; b = 45; r = a ++ b;
pair p, q; p = -q = (b, 0);
path base, cap, egg;
base = subpath (4, 8) of fullcircle scaled 2(2r-b);
cap = subpath (0, 4) of fullcircle scaled 2r shifted (0, a)
      cutbefore ((b, 0) -- (b, 2r))
      cutafter  ((-b, 0) -- (-b, 2r));
egg = point 4 of base {up} .. cap .. {down} base & cycle;
% more naturally "base {up} .. cap .. {down} & cycle"
% but then point 0 would not be at 3 o'clock
```

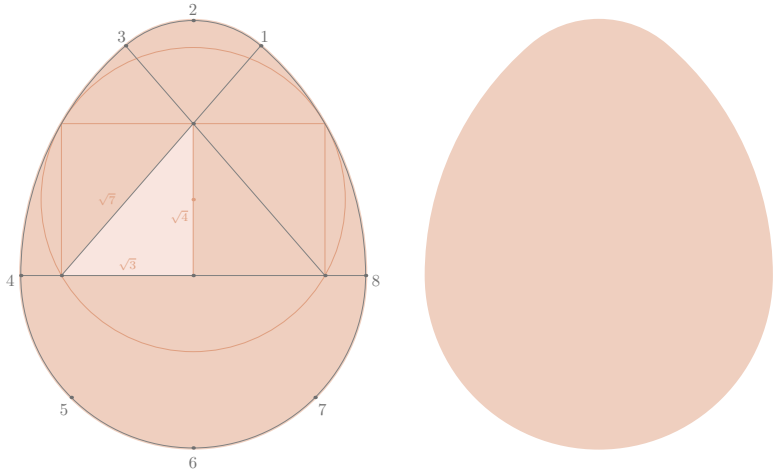


Note that you can use .. to create reasonably large circular arcs. The parts of the drawing for filling the egg, and showing the construction are similar to the first example.

17.3 A taller Pythagorean egg

A slightly different approach using a $\sqrt{3}$-$\sqrt{4}$-$\sqrt{7}$ triangle.

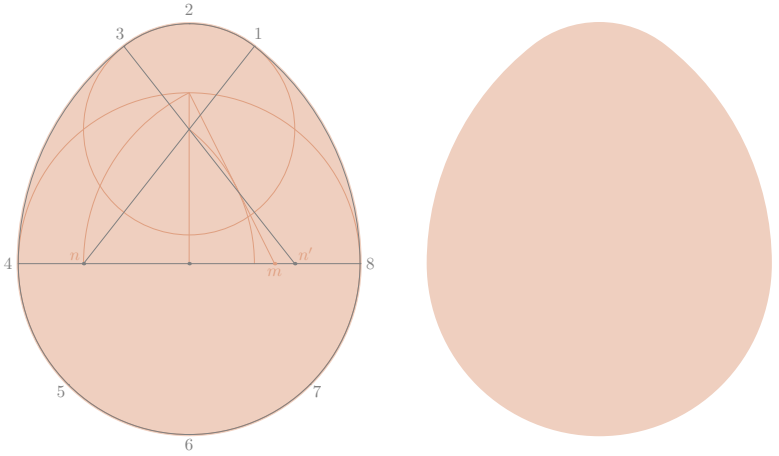
```
path base, cup, cap, egg;
base = fullcircle scaled 180;
z1 = point -2/3 of base;
z2 = point  2/3 of base;
z3 = point 10/3 of base;
z4 = point 14/3 of base;
z5 = 1/2[z2, z3]; z6 = 1/2[z4, z1];
numeric a, b;
a = abs(z3-z1) - abs(z4-z6);
b = abs(z3-z1) - abs(z4-z5);
cup = subpath (4, 8) of fullcircle scaled 2a shifted z6;
cap = fullcircle scaled 2b shifted z5
      cutbefore (z5 -- 2[z4, z5]) cutafter (z5 -- 2[z1, z5]);
egg = point 4 of cup {up} .. cap .. {down} cup & cycle;
```



17.4 Golden section egg

Another alternative construction, for Callicrates.

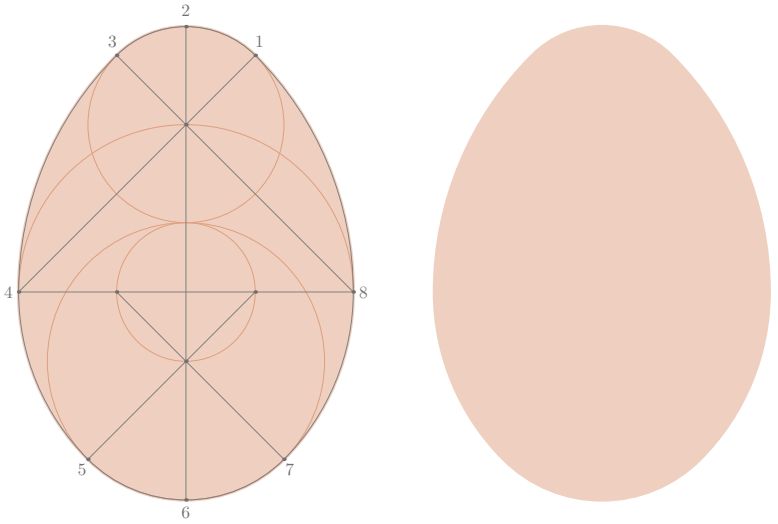
```
path base, aa, bb; pair m, n, n';
base = fullcircle scaled 200; m = 1/2 point 0 of base;
aa = halfcircle scaled 2 abs (point 2 of base - m)
                shifted m cutbefore (origin -- 1000 up);
n = point infinity of aa; n' = n reflectedabout(up, down);
bb = subpath (0, 2) of base shifted n cutafter (origin -- 1000 up);
path dome, cap, cup, egg;
dome = fullcircle
    scaled 2 (abs(n - point 0 of base) - abs(n - point 0 of bb))
    shifted point length bb of bb;
cap = dome
    cutbefore (point 4 of bb -- 2[n, point 4 of bb])
    cutafter  (point 4 of bb -- 2[n', point 4 of bb]);
cup = subpath (4, 8) of base;
egg = point 4 of cup {up} .. cap .. {down} cup & cycle;
```



17.5 Four point egg

So far all the eggs have been drawn with semi-circular big end, but this can be improved. To get a smoother curve, you can use four different sized arcs with four different centres of rotation to make up each side of the egg.

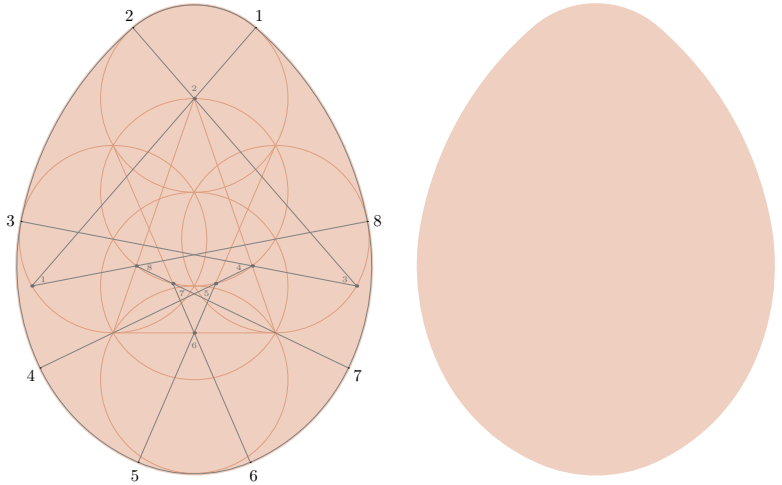
```
path egg, a, b, c, d;
a = fullcircle scaled 80;
b = a scaled 2 shifted point 6 of a;
c = halfcircle
    scaled 2 (abs(point 0 of a - point 5 of b) - abs(point 0 of a));
d = fullcircle
    scaled 2 abs(point 2 of a - point 2 of c) shifted point 2 of c;
egg = point 0 of c {up} .. subpath (1,3) of d .. {down} point 4 of c
      .. subpath (5, 7) of b .. cycle;
```



17.6 Five point egg

The next level of sophistication is to use five different arcs, but this is more complex and you lose the points at exactly E, N, W, and S.

```
numeric a; a = 56;
path r[];  % the rings
r1 = fullcircle scaled 2a shifted (0, -3/2 a);
r2 = fullcircle scaled 2a shifted (0, -1/2 a);
r3 = fullcircle scaled 2a shifted (0, +1/2 a);
r4 = fullcircle scaled 2a shifted (0, +3/2 a);
r5 = r2 rotatedabout(point 2 of r2, -60);
r6 = r2 rotatedabout(point 2 of r2, +60);
pair o[];  % the centres of rotation for each arc
o1 = point 6 of r5;
o2 = point 2 of r3;
o3 = point 6 of r6;
o4 = whatever[o3, point 2+4/3 of r2] = whatever[o2, point 2-4/3 of r1];
o8 = whatever[o1, point 2-4/3 of r2] = whatever[o2, point 2+4/3 of r1];
o6 = 1/2[point 2-4/3 of r1, point 2+4/3 of r1];
o5 = whatever[o6, point 2-4/3 of r3] = whatever[o4, point 2+4/3 of r1];
o7 = whatever[o6, point 2+4/3 of r3] = whatever[o8, point 2-4/3 of r1];
pair u[], t[];  % directions and points for the egg
u0 = (o8 - o1) rotated 90;  t0 = directionpoint u0 of r6;
u1 = (o2 - o1) rotated 90;  t1 = directionpoint u1 of r4;
u2 = (o2 - o3) rotated 90;  t2 = directionpoint u2 of r4;
u3 = (o4 - o3) rotated 90;  t3 = directionpoint u3 of r5;
u4 = (o5 - o4) rotated 90;
u5 = (o6 - o5) rotated 90;
u6 = (o6 - o7) rotated 90;
u7 = (o7 - o8) rotated 90;
t4 = directionpoint u4 of fullcircle scaled 2 abs (t3 - o4) shifted o4;
t5 = directionpoint u5 of fullcircle scaled 2 abs (t4 - o5) shifted o5;
t6 = directionpoint u6 of fullcircle scaled 2 abs (t5 - o6) shifted o6;
t7 = directionpoint u7 of fullcircle scaled 2 abs (t6 - o7) shifted o7;

path egg; egg = for i=0 upto 7: t[i] {u[i]} .. endfor cycle;
```



☞ Instead of defining and joining circular arcs, this construction defines the points for the egg and the desired directions at each point; all the work of making the circular arcs is left to the .. connector. The six symmetrically-arranged rings r_1 to r_6 are used to define eight centres of rotation o_1 to o_8 which are either points on the rings or intersections of lines between them. Then eight directions u_1 to u_8 are defined at right angles to lines between pairs of centre points. Finally the handy `directionpoint` macro is used to find the points where the relevant circle is moving in that direction. To make the egg, these points are joined up with .. constrained by the matching direction to make circular arcs.

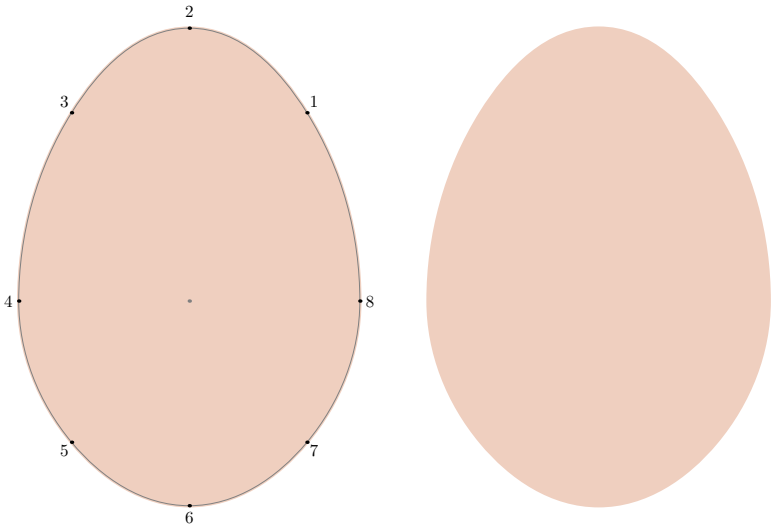
⟵ Like so.

17.7 A superellipse egg

All this construction is a good exercise in ingenuity, but if you just want a simple quick egg path, then the superellipse macro gives you a METAPOST-specific option. All you need is this:

```
path egg;
egg = (superellipse(right, 1.6 up, left, 1.2 down, 0.69));
egg := egg scaled 100;
```

[This space intentionally left blank]



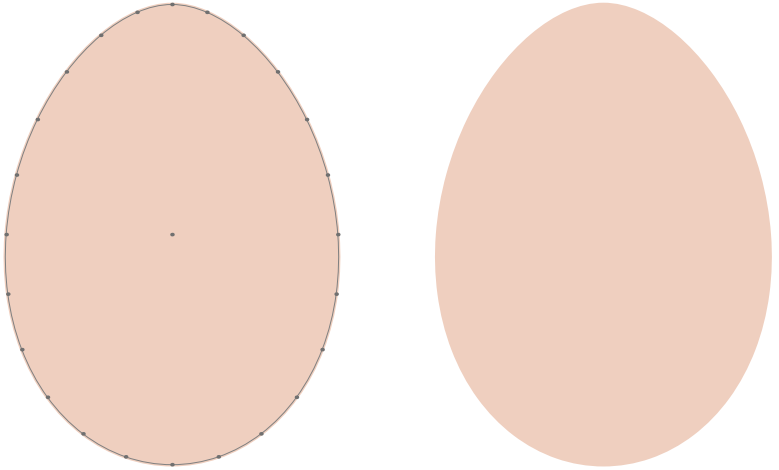
17.8 The perfect egg

The last word in egg curve perfection is the algebraic solution provided by the TDCC Laboratory in Japan [https://nyjp07.com/index_E.html].

```
path egg;
egg = for t=-180 step 15 until 180 - eps:
        (0.78 cosd(1/4 t) * sind(t), -cosd(t)) ..
    endfor cycle;
egg := egg scaled 128;
```

Note that, unlike all the others, this path has 24 points and the path starts at the top. You can draw it starting at 3 o'clock, and with only 8 points but the egg is slightly less perfect. Just replace the loop above with:

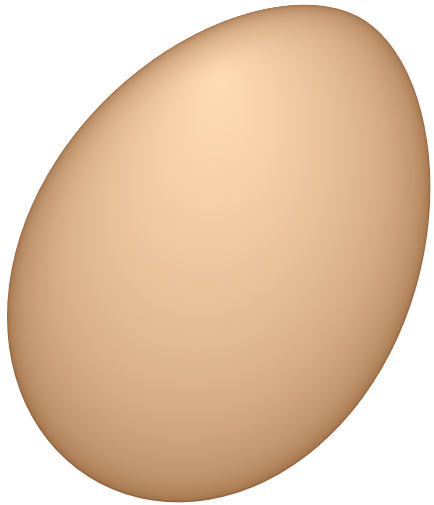
```
egg = for t=90, 135, 180, -135, -90, -45, 0, 45:
        (0.78 cosd(1/4 t) * sind(t), -cosd(t)) ..
    endfor cycle;
```



17.9 Egg kitsch

If you want eggs that look solid, then you can use interpath:

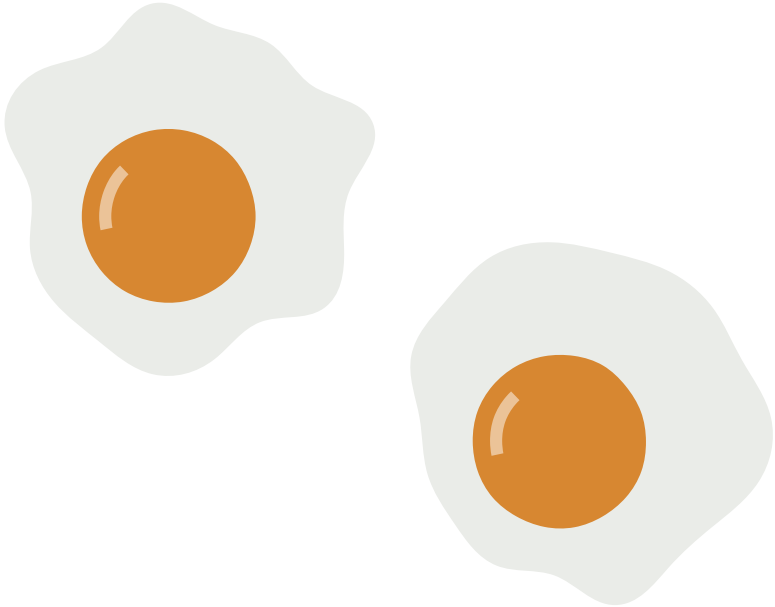
```
path egg, spot;
egg = (for t=-180 step 15 until 180 - eps:
  (0.78 cosd(1/4 t) * sind(t), -cosd(t)) ..
endfor cycle) scaled 100;
spot = fullcircle scaled 4 shifted 3/4 point 3 of egg;
vardef fade_filled(expr egg, spot, dark, light, n) = image(
  for i = 0 upto n:
    fill interpath(i/n, egg, spot) withcolor ((i/n)**1/3)[dark,light];
  endfor)
enddef;
beginfig(1);
  color a; a = 1/256(150, 100, 60);
  color b; b = 1/256(256, 220, 180);
  draw fade_filled(egg, spot, a, b, 256) rotated -30;
endfig;
```



This works nicely with any of the egg paths defined in this section.

And finally, if all that has made you feel peckish, then how about these?

```
path yolk, base;
color cooked_egg_yolk; cooked_egg_yolk = 1/256(216, 136, 49);
color cooked_egg_white; cooked_egg_white = 1/256(235, 237, 233);
vardef fried_egg(expr r) = image(
  save base, yolk; path base, yolk;
  yolk = for i=0 upto 17: (r + 1/8 normaldeviate) * dir 20i .. endfor cycle;
  base = (for i=0 upto 17: (2r + 1/8r * normaldeviate) * dir 20i .. endfor cycle)
         shifted (uniformdeviate r/2, uniformdeviate r/2);
  fill base withcolor cooked_egg_white;
  fill yolk withcolor cooked_egg_yolk;
  fill subpath (6.7, 9.6) of yolk scaled 0.8 --
       subpath (9.6, 6.7) of yolk scaled 0.66 -- cycle
       withcolor 1/2[cooked_egg_yolk, white]
) enddef;
for i=0 upto 1: draw fried_egg(40) shifted 120 dir 120i; endfor
```



18 Data visualizations

Graphs and other displays that show data, rather than a mathematical function, are presented in this section, with a focus on illustrations for books and technical papers. Most of the examples here follow those developed in Edward Tufte's books on graphics.

- *The Visual Display of Quantitative Information*
- *Visual Explanations*
- *Beautiful Evidence*
- … and others, at `https://edwardtufte.com`

Most of the drawings in this section have a pale manila background, which was added using the technique from §13.8. At the beginning, the background colour is changed like this:

```
background := (1, 1, 31/32);  % change at start so "unfill" works
```

Then at the end of the figure, the background is applied like this:

```
picture p; p = currentpicture; clearit;
bboxmargin := 12; unfill bbox p; draw p;
```

But there are lots of figures in this section, so this "wrapping" is actually applied using the begin- and end-figure "hooks". If you examine the source code, you will see that each figure inputs a file called `tufte-manila-paper` which contains this:

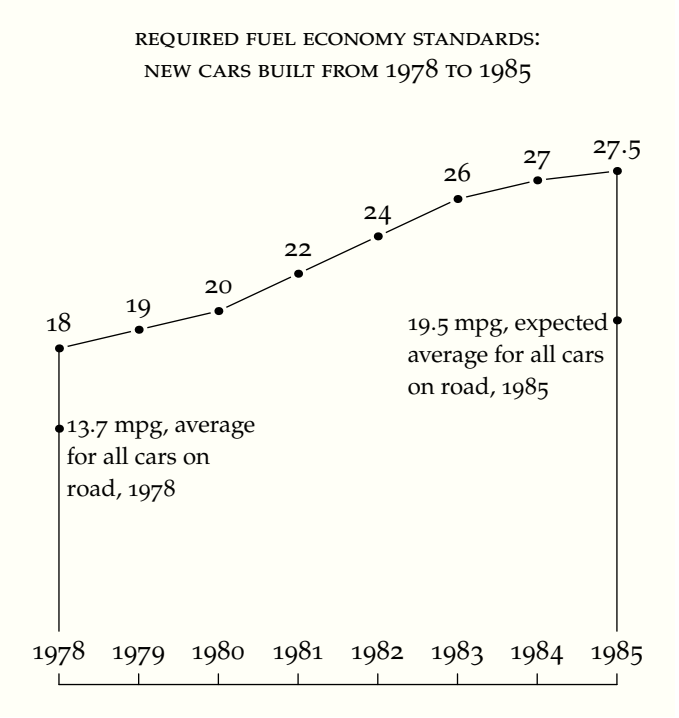
```
extra_beginfig := "background := (1,1,31/32);";
extra_endfig := "picture p; p = currentpicture; clearit;"
              & "bboxmargin := 12; unfill bbox p; draw p;";
```

The `beginfig` and `endfig` macros provided by plain METAPOST automatically apply anything that is set in the two `extra` strings.

18.1 Simple time lines

```
\documentclass{standalone}
\usepackage{luamplib}
\mplibtextextlabel{enable}
\usepackage{fontspec}
\setmainfont[Numbers=OldStyle]{TeX Gyre Pagella}
\begin{document}
\begin{mplibcode}
input tufte-manila-paper
beginfig(1);
path data, p; numeric n, u, v;
data = (1978, 18) -- (1979, 19) -- (1980, 20) -- (1981, 22) --
       (1982, 24) -- (1983, 26) -- (1984, 27) -- (1985, 27.5);
u = xpart urcorner textext("1980\quad"); v = 7;
p = data shifted -(1978, 0) xscaled u yscaled v;

draw p;
for i=0 upto length p:
  z[i] = point i of p;
  undraw z[i] withpen pencircle scaled 2 dotlabeldiam;
  dotlabel.top("\strut" & decimal ypart point i of data, z[i]);
  label("\strut" & decimal xpart point i of data, (x[i], 12));
  draw (x[i], 0) -- (x[i], 4);
endfor
draw (x0, 20) -- z0; draw (x7, 20) -- z7; draw (x0, 0) -- (x7, 0);

dotlabel.rt(btex \vbox to 6pt{\halign{\small #\hss\cr
  13.7 mpg, average\cr for all cars on\cr road, 1978\cr
  }\vss} etex, (x0, 13.7v));
dotlabel.lft(btex \vbox to 6pt{\halign{\small #\hss\cr
  19.5 mpg, expected\cr average for all cars\cr on road, 1985\cr
  }\vss} etex, (x7, 19.5v));
label.top(btex \vbox{\halign{\hss\textsc{#}\hss\cr
  required fuel economy standards:\cr
  new cars built from 1978 to 1985\cr}} etex,
  point 5/2 of bbox currentpicture shifted 13 up);
endfig;
\end{mplibcode}
\end{document}
```

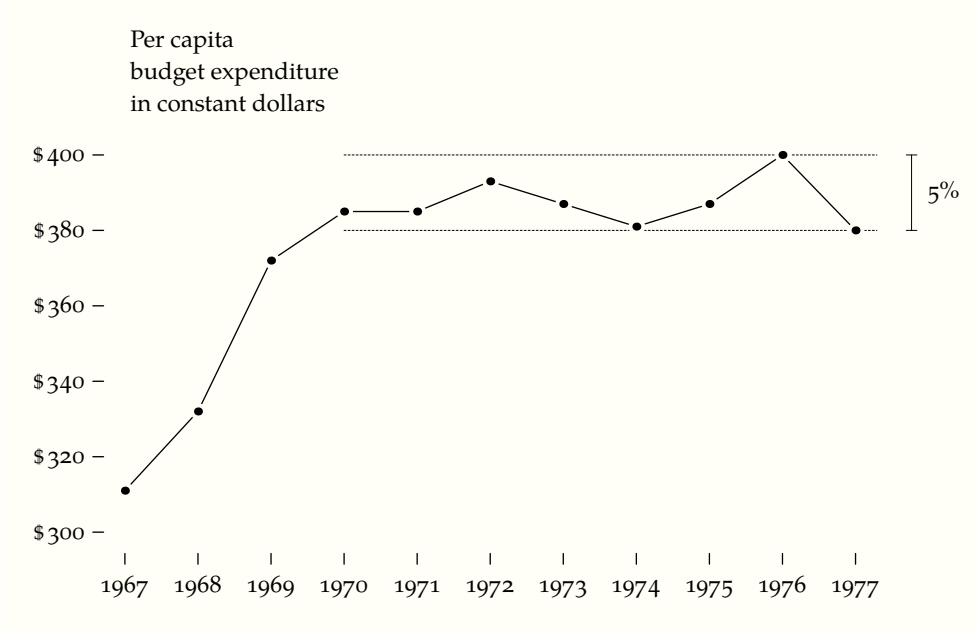
- The complete LuaTEX document shows the use of `fontspec` to get the Palatino-like font, used in E. Tufte's books.
- The data is stored as ⟨pair⟩ values in a ⟨path⟩ variable. This only works if both data values are numeric. If the values are larger than 4096 you either need to scale them or use the `double` number system.
- Notice how the horizontal unit u is set by measuring the width of a label using the technique discussed in §11.1.6. And that you have to shift the data path left before applying the x-scaling to avoid overflow.



18.2 Time line with minimal annotation

```
beginfig(1);
  path data; numeric n;
  data = (1967, 311) -- (1968, 332) -- (1969, 372) -- (1970, 385)
      -- (1971, 385) -- (1972, 393) -- (1973, 387) -- (1974, 381)
      -- (1975, 387) -- (1976, 400) -- (1977, 380);
  n = length data;
  numeric u, v; path p;  % make data --> p
  u = xpart urcorner textext("1980\kern 0.75em"); v = 1.414;
  p = data shifted -(xpart point 0 of data, 300) xscaled u yscaled v;
  for i=0 upto n: z[i] = point i of p; endfor
  for d = 300 step 20 until 400: % y-axis
    numeric y; y =  (d - 300) * v; draw (-8, y) -- (-12, y);
    label.lft("\strut\scriptsize\$\,\small" & decimal d, (-12, y));
  endfor
  path a, b, c;  % annotating lines
  a = (x3, y[n-1]) -- (x[n] + 8, y[n-1]);
  b = (x3, y[n])   -- (x[n] + 8, y[n]);
  c = (x[n] + 21, y[n]) -- (x[n] + 21, y[n-1]);
  drawoptions(withpen pencircle scaled 1/4);
    draw a dashed evenly scaled 1/4;
    draw b dashed evenly scaled 1/4;
    draw c;
    for i=0,1: draw (left--right) scaled 2 shifted point i of c; endfor
    label("\small 5\%", point 1/2 of c shifted 12 right);
  drawoptions();
  draw p;  % the data points
  for i = 0 upto n:
    undraw z[i] withpen pencircle scaled 2 dotlabeldiam;
    draw z[i] withpen pencircle scaled dotlabeldiam;
    draw (x[i], -8) -- (x[i], -12);
    label("\strut\small" & decimal xpart point i of data, (x[i], -20));
  endfor
  label.urt(btex \vbox{\halign{\small #\hfill\cr Per capita\cr
    budget expenditure\cr in constant dollars\cr}} etex, (x0,y[n-1]+10v));
endfig;
```

- This one has an even sparser frame, and dispenses with the labels on each data point.
- It is suggested that you resist the temptation to make very many special macros to do charts like this; the ideas here are mainly to show that METAPOST makes a good environment for following Tufte's advice about charts: to maximize data ink and minimize chart junk; albeit at the cost of some pains-taking.



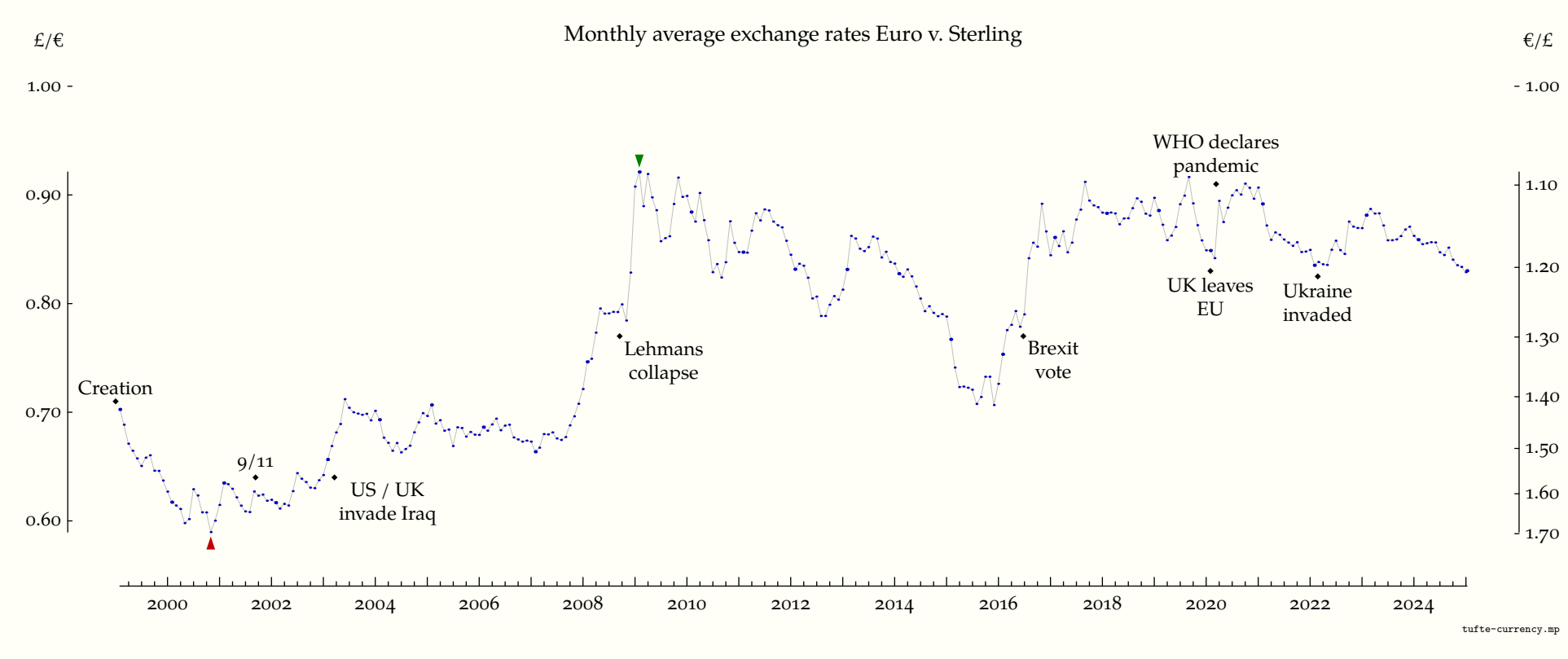
18.3 Time line with more complex dates

If the dates in your time line are more granular, then you need a better way to deal with them. This chart below shows the £/€ exchange rate by month. The dates on the horizontal axis were transformed from calendar dates to a serial number using the routine shown on the right, which allows you to do this:

```
dotlabel.bot("Brexit vote", (base(2016, 6, 24) * u, 78 v));
```

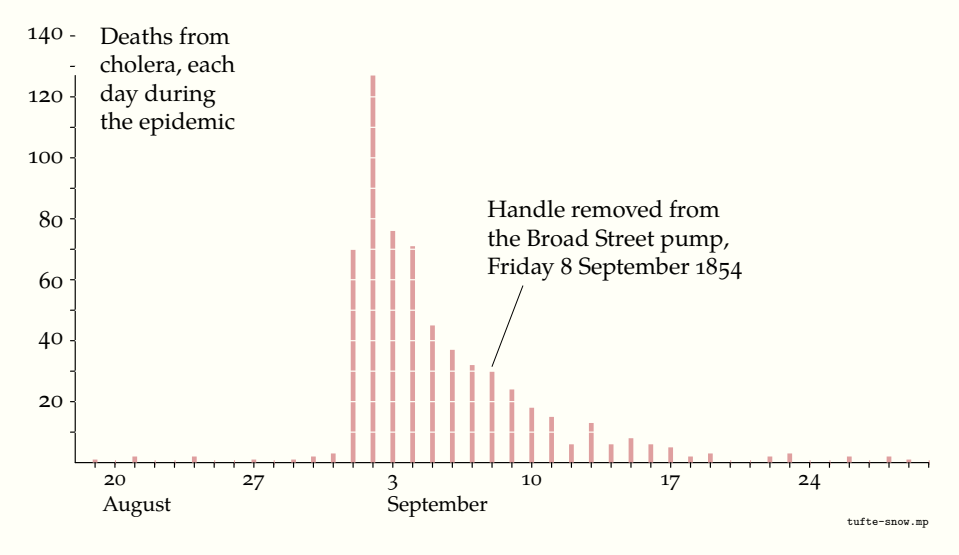
for suitable values of u and v.

```
vardef base(expr Y, M, d) =
  save m, y;
  if M < 3: m = M + 9; y = Y - 1;
  else: m = M - 3; y = Y; fi
  365/1024 y + (floor(y/4) - floor(y/100) + floor(y/400)
    + floor((2+3m)/5) + 30m + d - 307) / 1024
enddef;
```



18.4 Daily events with annotation

THIS CHART is adapted from Edward Tufte's *Visual explanations*, p.33; it presents the data from John Snow's table of daily deaths in the London cholera epidemic of 1854 in a simple bar chart form, with annotation pointing out that the epidemic was already declining when the supposedly decisive intervention of removing the handle from the water pump at the centre of the outbreak was made.



The data for the chart was edited directly into the METAPOST source, using a text editor to copy the data from the source table, and format them into lists so that you can use loops to capture the values in two parallel indexed numeric variables `deaths[]` and `day_number[]`. It is useful to be able to group the data visually into weeks to reduce the chance of errors, but there is no other significance.

Note that the bars are drawn with a wide pen and `cutdraw` in order to get nice square ends on the bars; then `undraw` is used to make implicit grid lines.

```
numeric i, deaths[], ymax, dmax; i = 0; dmax = 0;
for n =                 1,
    0, 2, 0, 0, 2, 0, 0,
    1, 0, 1, 2, 3,70,127,
   76,71,45,37,32,30,24,
   18,15, 6,13, 6, 8, 6,
    5, 2, 3, 0, 0, 2, 3,
    0, 0, 2, 0, 2, 1, 0:
  deaths[incr i] = n; if n > dmax: dmax := n; fi
endfor
numeric ymax; ymax = 20 * ceiling (dmax / 20);
numeric days, day_number[]; days = 0;
for n =                 19,
    20,21,22,23,24,25,26,
    27,28,29,30,31, 1, 2,
     3, 4, 5, 6, 7, 8, 9,
    10,11,12,13,14,15,16,
    17,18,19,20,21,22,23,
    24,25,26,27,28,29,30:
  day_number[incr days] = n;
endfor
numeric u, v; days * u = 5in = ymax * 2v;
path xx; xx = origin -- (days * u, 0); draw xx;
path yy; yy = origin -- (0, dmax * v); draw yy;
for x = 1 upto days:
  cutdraw (x * u, 0) -- (x * u, deaths[x] * v)
    withpen pencircle scaled 2 withcolor 5/8[2/3 red, white];
  draw (origin -- 2 down) shifted (x * u, 0);
  if x mod 7 = 2: label.bot("\small" & decimal day_number[x], (x*u, -2)); fi
  if      x = 1: label.lrt("\small August", (x * u + 1, -12));
  elseif x = 15: label.lrt("\small September", (x * u + 1/2 u, -12));
  fi
endfor
for y = 10 step 10 until ymax:
  draw (origin -- 2 left) shifted (0, y * v);
  undraw xx shifted (0, y * v) withpen pencircle scaled 1/2;
  if y mod 20 = 0: label.lft(decimal y, (-2, y * v)); fi
endfor
```

☞ *The code for the annotation and the title was omitted to save space. See the source file for details.*

18.5 Scatter plot with annotations and simple jitter

THIS CHART is from Tufte's analysis of the Challenger space shuttle disaster. His objective here was to show all the available data relating launch temperature and damage to the booster rockets, and to point out that the forecast for 28 January 1986 was very much colder than any other launch.

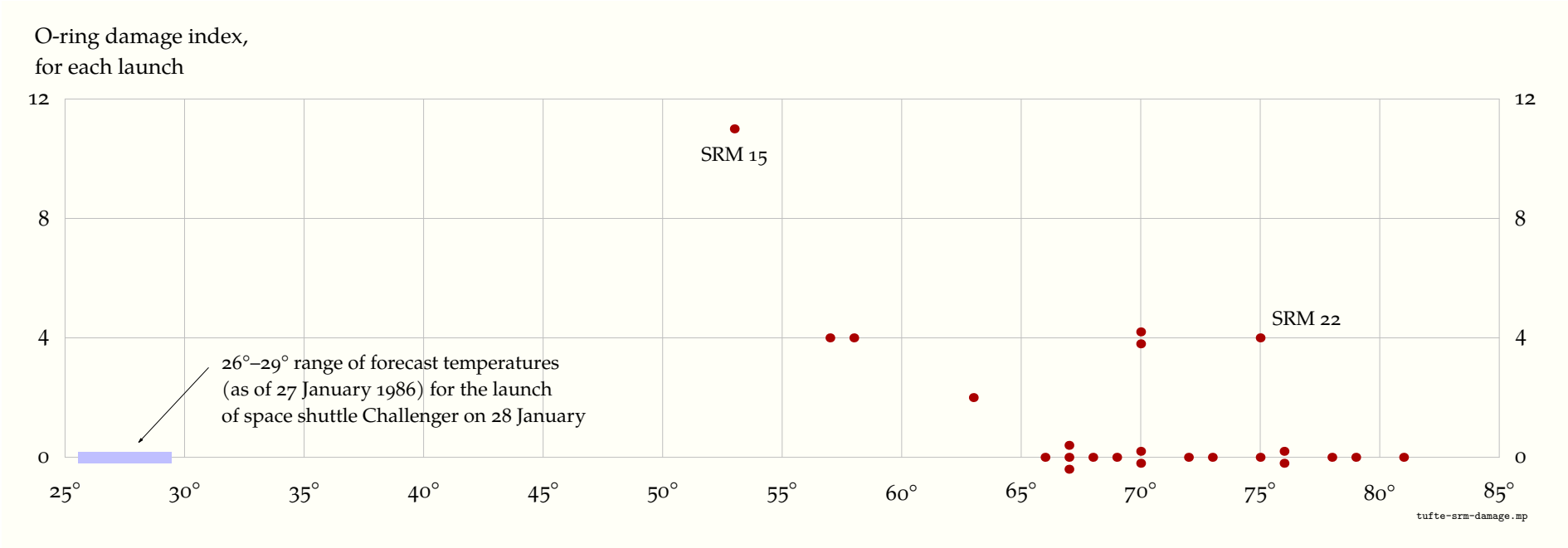
The data was again edited directly into the METAPOST source, but this time captured as a ⟨path⟩ variable. One advantage of this is that it allows you to process all the values in parallel using `scaled` or `shifted` on the whole path.

```
numeric u, v; u = 10.8; 5u = 4v;
damage := damage xscaled u yscaled v;
```

The individual points are accessed using the `point n of p` syntax:

```
for i = 1 upto length damage:
  draw point i of damage withpen pencircle scaled (3/2r * v) withcolor 2/3 red;
endfor
```

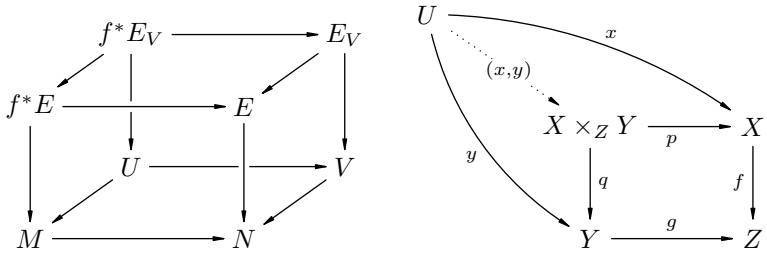
```
numeric r; r = 1/5;  % adjustment of the marks where required...
path damage; damage = origin .. (53, 11) .. (57, 4) .. (58, 4) ..
  (63, 2)..
  (66, 0)..
  (67, 0 + 2r) ..  % because the data set is quite small
  (67, 0) ..       % the simplest way to show multiple points
  (67, 0 - 2r) ..  % at the same (x, y) location is to add
  (68, 0)..        % this little vertical shift by hand
  (69, 0)..
  (70, 4 + r)..    % the data is captured in a path, but only the
  (70, 4 - r)..    % points are drawn, not the lines between them
  (70, 0 + r)..
  (70, 0 - r)..
  (72, 0)..
  % and so on
```



19 Commutative diagrams

If you want lots of complex bells and whistles on your commutative diagrams, then you probably want to use specialist tools like `tikz-cd` or `xypic`, but if your needs are simpler, plain METAPOST is more than capable, and if you are already using it for other illustrations, there is one less thing to learn.

Here are two examples to illustrate some general techniques. They may be familiar to readers of the manuals for the tools referred to above.



The complete code used to generate the right-hand picture is shown on the right.→ The approach taken for both examples is first to define the node labels as pictures placed as needed, and then write a special-purpose `connect` macro to make consistent arrows between the nodes, using `center`, `bbox`, `cutbefore`, and `cutafter` as appropriate.

The $X \times_Z Y$ example needs two variations. The `curved_connect` macro takes a string for the label, the two nodes to be connected, and the initial direction for the curved path to connect them. You can't have optional macro arguments in METAPOST but you can call macros from inside another macro, so a second simpler macro `connect` is defined to do straight connections. To avoid repetition, this macro simply calls `curved_connect` with the required direction from a to b.

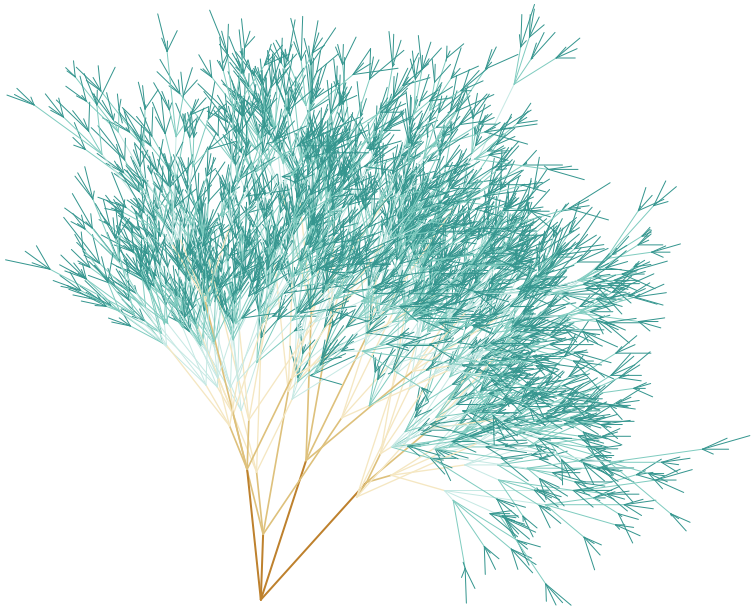
For the f^*E example, `connect` is simpler, because all the arrows are straight and there are no labels, but needs to allow for arrows crossing:

```
vardef connect(expr a, b) =
  save line; path line; interim bboxmargin := 3;
  line = center a .. center b cutbefore bbox a cutafter bbox b;
  cutdraw line withpen pencircle scaled 4 withcolor background;
  drawarrow line
enddef;
```

```
beginfig(1);
  picture U, XY, X, Y, Z;
  z1 = -z2 = (-61, 42);
  U  = thelabel("$U$", z1);
  XY = thelabel("$X\times_ZY$", origin);
  X  = thelabel("$X$", (x2, 0));
  Y  = thelabel("$Y$", (0, y2));
  Z =  thelabel("$Z$", z2);

  forsuffixes @=U, XY, X, Y, Z: draw @; endfor

  ahangle := 20;
  vardef connect@#(expr s, a, b) =
    curved_connect@#(s, a, b, center b - center a)
  enddef;
  vardef curved_connect@#(expr s, a, b, d) =
    save line, mark; path line; picture mark;
    line = center a {d} .. center b;
    interim bboxmargin := 4;
    drawarrow line cutbefore bbox a cutafter bbox b;
    mark = thelabel@#("$\scriptstyle " & s & "$",
                                        point 1/2 of line);
    interim bboxmargin := 1;
    unfill bbox mark; draw mark;
  enddef;

  connect.bot("p", XY, X);
  connect.rt ("q", XY, Y);
  connect.top("g", Y, Z);
  connect.lft("f", X, Z);
  curved_connect.urt("x", U, X, right);
  curved_connect.llft("y", U, Y, dir -80);

  drawoptions(dashed withdots scaled 1/2);
  connect("(x,y)", U, XY);
  drawoptions();
endfig;
```

This example assumes you are using `mplibtextextlabel` – *§12.1*

20 Recursion and iteration



This chapter is not a tutorial on recursion or iteration *per se*, but rather more of an exploration of the METAPOST techniques you can use to create some particular types of drawing such as trees, plane-filling curves, fractals, and some types of tiling pattern [§22].

Even so, it is perhaps useful to review some of the basic programming ideas involved. Consider for example the greatest common divisor algorithm presented in §4.6.

```
vardef gcd(expr a, b) = if b = 0: a else: gcd(b, a mod b) fi enddef;
```

The reason this works is that we know (mathematically) that $0 \le a \bmod b < b$, and therefore that the arguments to the recursive call must get smaller each time and eventually we must get to $b = 0$ when the answer will be a. This is the basic recursive approach: ensure at least one argument gets smaller each time and stop when you get to a given limit. The examples in this chapter use one of two simple approaches:

- Explicitly pass a `level` argument that is decremented on each recursive call, and stop the recursion when the level gets to zero
- Pass a ⟨path⟩ or two ⟨pair⟩ arguments, and stop the recursion when the path is too short or the pairs are too close together.

If you find recursion confusing, you can nearly always use an iterative approach instead. For example, you can implement the `gcd` function like this:

```
vardef gcd(expr A, B) = save r, a, b; numeric a, b, r; a := A; b := B;
  forever:
    r := a mod b; exitif r = 0;
    a := b; b := r;
  endfor b
enddef;
```

Notice that you have to use assignment in the loop to update the variables, and that you cannot assign to the arguments of a macro. Notice also that this version requires both arguments to be positive integers. You need to use your judgement to decide which is the better approach for a given problem.

20.1 The Koch curve

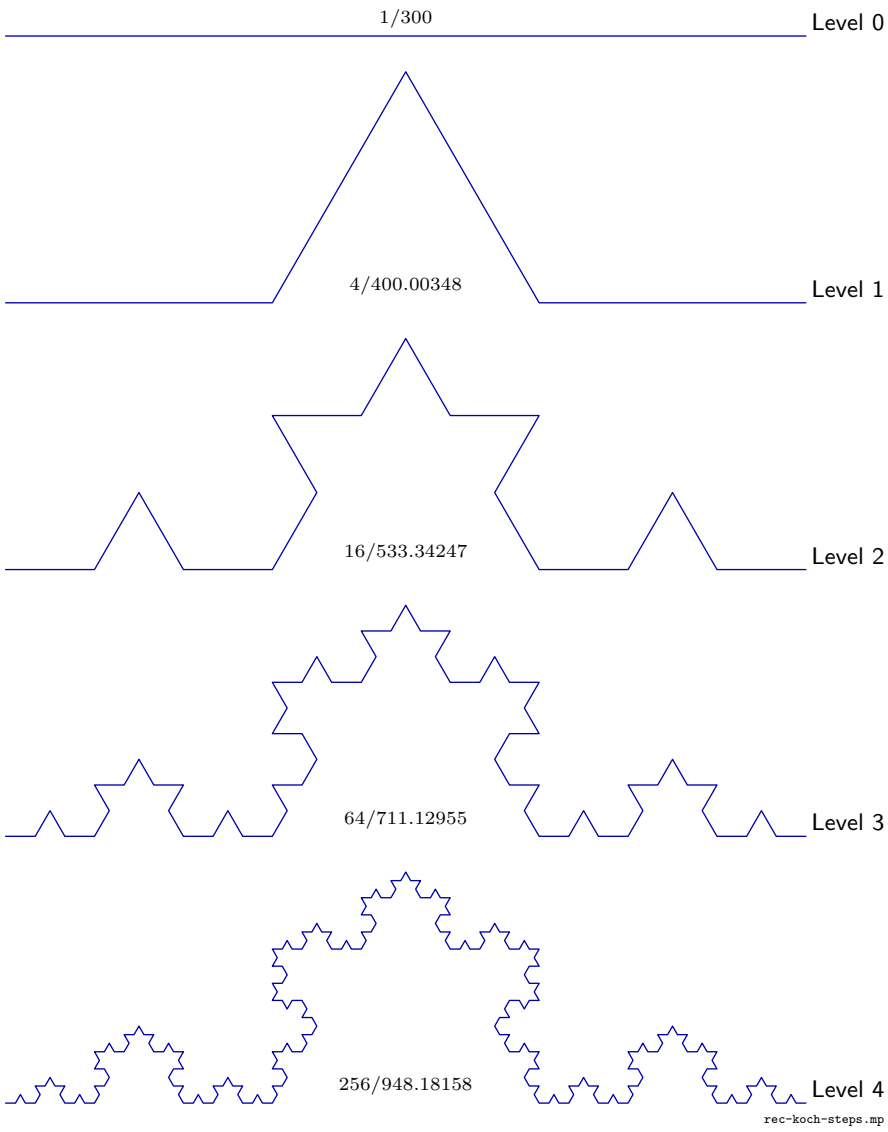
The Swedish mathematician Helge von Koch originally devised the Koch curve as an example of a non-differentiable curve that could be constructed with elementary geometry. It makes a good introduction to recursive paths with METAPOST. The construction is recursive: each straight line segment in the path is replaced with four copies of itself, scaled down $\frac{1}{3}$ and arranged as shown at Level 1 ⟶ At each level of the construction, the number of points in the path increases four-fold and the `arclength` of the path gets $\frac{4}{3}$ longer.

```
vardef koch(expr level, a, b) =
  if level = 0:
    a -- b
  else:
    save p, q, r; pair p, q, r;
    p = 1/3[a,b]; r = 2/3[a,b]; q = r rotatedabout(p, 60);
    koch(level-1, a, p) &
    koch(level-1, p, q) &
    koch(level-1, q, r) &
    koch(level-1, r, b)
  fi
enddef;
```

The five levels were drawing using this function in a loop like this:

```
for n=0 upto 4:
    draw koch(n, origin, 300 right) shifted (0, -100n)
endfor
```

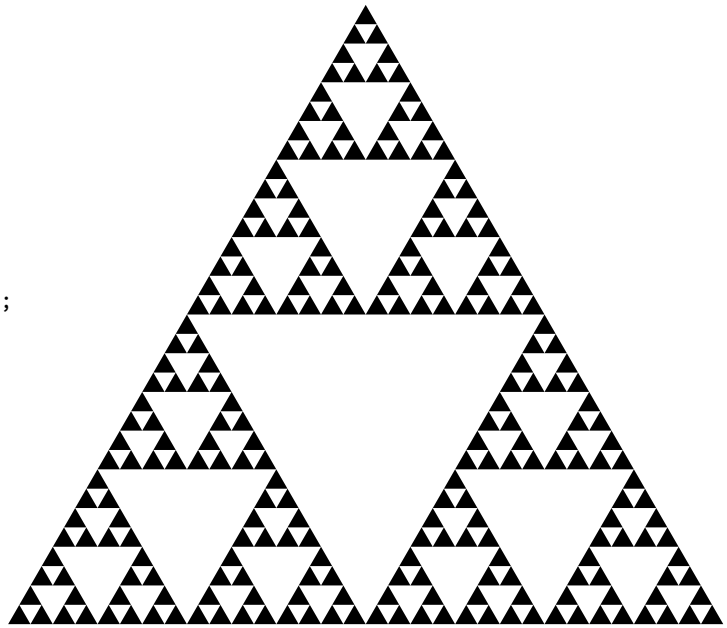
The annotations in the middle show the `length` of the path and the `arclength`. The curve will always fit in the same bounding box but in theory it can become infinitely long. In practice it looks much the same past level 6, and you will start running into METAPOST limitations. The length of a path is not limited by the default scaled number system, but the numbers actually returned by `length` and `arclength` may not be reliable for very long paths. Also the call stack may get very large and the single-threaded processing time can get very slow, and when the length of each segment shrinks to the size of your pen, you will start losing detail.



20.2 Sierpinski's gaskets

The second example of recursive construction also dates from the early 20th century, but unlike the infinite length of von Koch's curve, the area of Sierpinski's gasket tends to zero. In the original specification, you are supposed to remove the central quarter of each triangle, but this program does it the other way round and delays drawing the triangles until they are small enough.

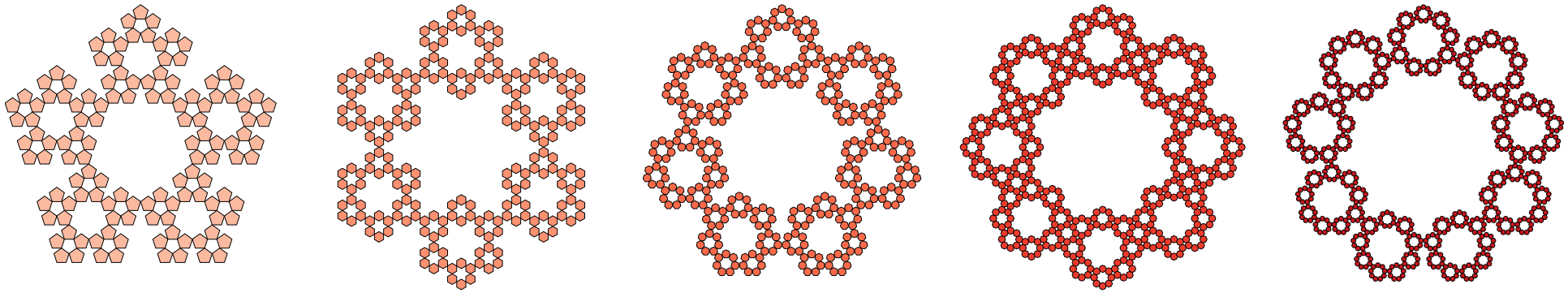
```
vardef gasket(expr t, s, limit) =
  if length (point 1 of t - point 0 of t) < limit:
    fill t;
  else:
    save little_t; path little_t; little_t = t scaled s;
    for i=1 upto length t:
      gasket(little_t shifted (point i of t - point i of little_t), s, limit);
    endfor
  fi
enddef;
```



Note the useful idiom `shifted (point i of t - point i of little_t)` – this neatly tucks a copy of the small triangle into the appropriate corner of the big triangle. You can make this even simpler by coding the scaling parameter s and *limit* as constants in the recursive routine, so that you do not have to pass them down each time. The triangular gasket was generated using the macro like this:

```
beginfig(1);
  path T; T = for i = 1 upto 3: 220 up rotated (120i) -- endfor cycle;
  gasket(T, 1/2, 20);
endfig;
```

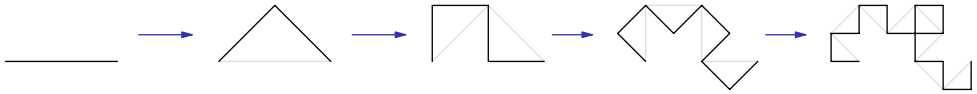
You can generalize this idea, for example by making the path T into a pentagon with scaling factor $s = (3 - \sqrt{5})/2$, or a hexagon with $s = 1/3$, and so on:



rec-sierpinski-garlands.mp

20.3 The Heighway dragon

The Heighway dragon curve dates from the 1960s, and is created in a similar way to the Koch curve: each straight line segment is recursively replaced with two copies of itself scaled down and arranged as shown.



Note that every other segment is flipped left and right. At each stage, the number of points increases two-fold and the path gets $\sqrt{2}$ times longer. Here is a recursive routine to generate the curve as a single path.

```
numeric r, theta; r = sqrt 1/2; theta = 45;
vardef dragon(expr level, a, b) =
  if level > 0:
    save p; pair p;
    p = r[a, b] rotatedabout(a, theta);
    dragon(level - 1, a, p) & reverse dragon(level - 1, b, p)
  else:
    a .. b
  fi
enddef;
```

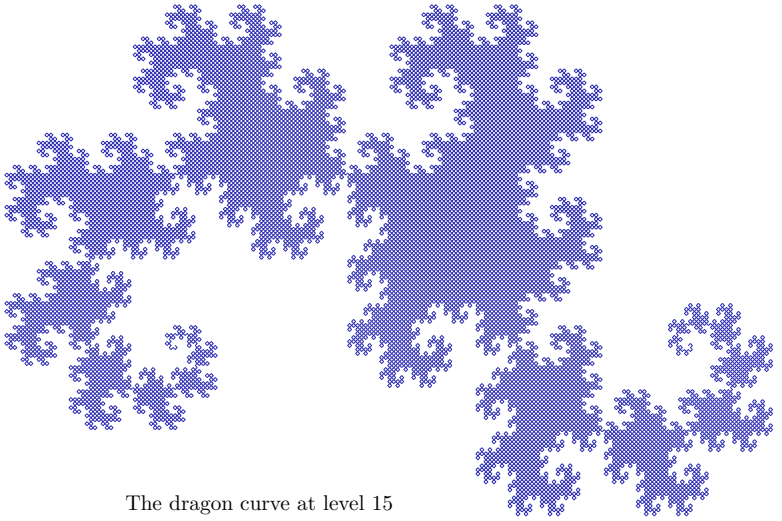
The blue dragon was created with: `draw dragon(15, origin, 240 right)` ↗

After the fourth level the corners of the curve start to touch each other, but they never cross. You can see this if you draw the curve with rounded corners.

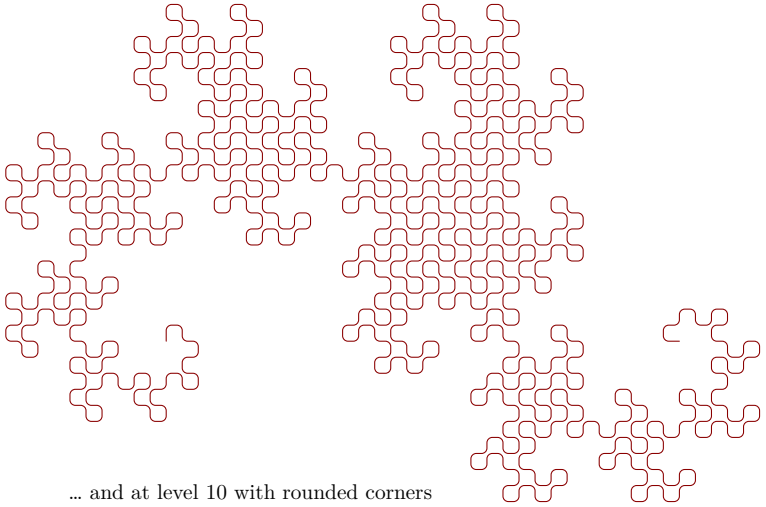
```
vardef rounded_corners expr p =
  save r, n; numeric r, n; r = 1/3; n = length p;
  subpath (0, 1-r) of p
  for t=1 upto n-1:
    .. subpath (t+r, t+1-r) of p
  endfor .. subpath (n-r, n) of p
enddef;
```

The red dragon was: `draw rounded_corners dragon(10, origin, 240 right)` ⟶

☞ The length of the dragon path at level n is 2^n, so if you want to do arithmetic with the path length you need to use a big number system when $n > 11$.



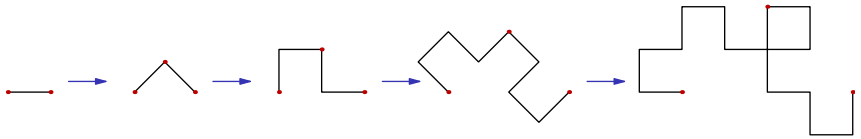
The dragon curve at level 15



… and at level 10 with rounded corners

20.4 Iterative dragons

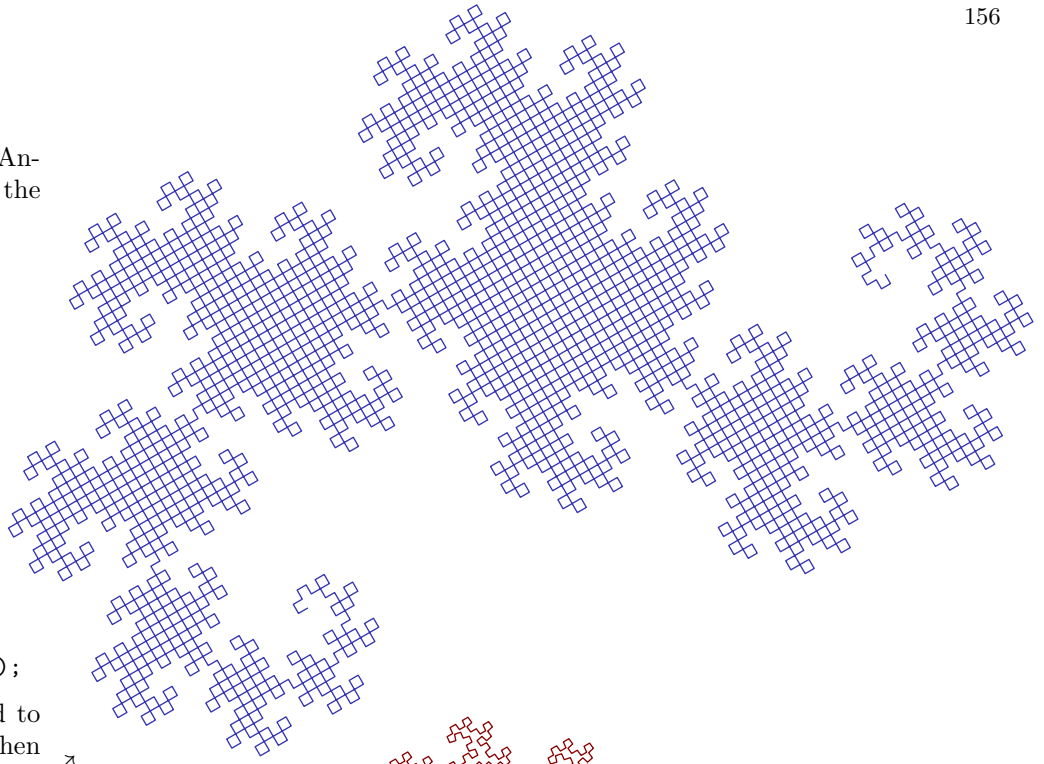
THE DRAGON CURVE can also be created with iteration instead of recursion. Another way of viewing the stages of developing the curve is that at each stage the whole curve is replaced by two copies of itself arranged as shown:



We can do this in two steps in a loop, like this:

```
path p; p = origin -- dir 30;
numeric n;
for i=1 upto 12:
  n := length p;
  p := p rotated 45;
  p := p & reverse p rotatedabout(point n of p, 90);
endfor
draw p scaled (384 / xpart (urcorner p - llcorner p)) withcolor (.2,.2,.7);
```

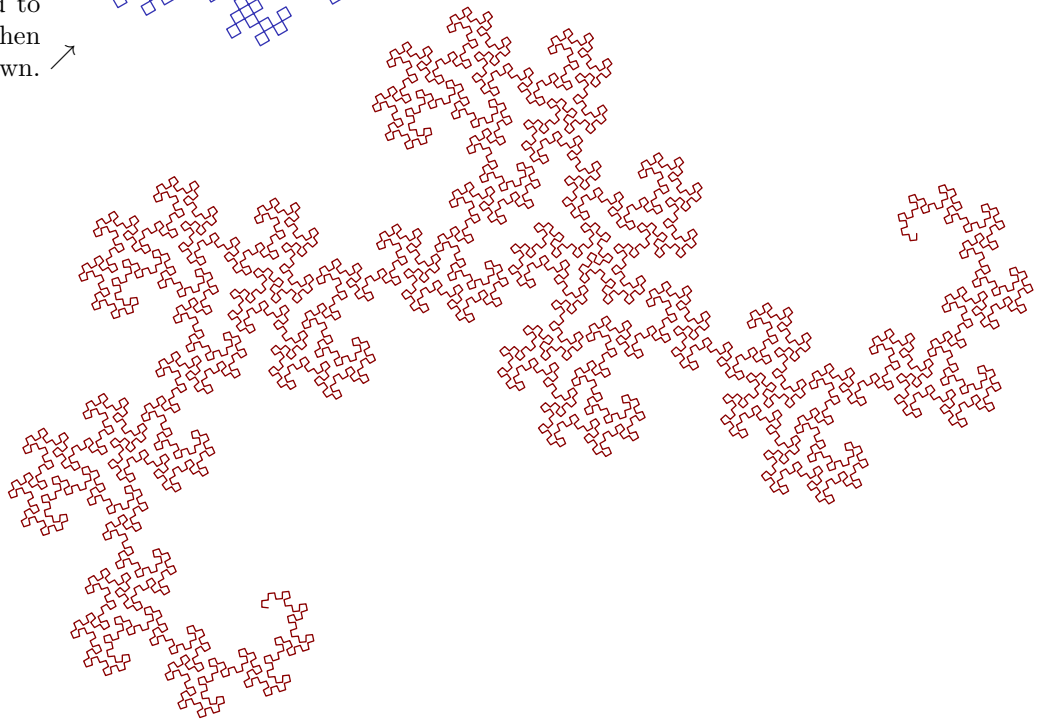
In this approach, instead of scaling down each time, the curve is just allowed to grow $\sqrt{2}$-times bigger in each loop, then scaled to the desired width (384 pt) when it is complete. The 12 stages in the loop produce the rotated blue dragon as shown.



With a small adaptation, you can also use this to explore variations.

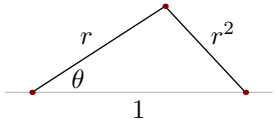
```
path p; p = origin -- dir 30;
numeric n, r; r = 3;
for i=1 upto 12:
  n := length p;
  p := p rotated (45 - r);
  p := p & reverse p rotatedabout(point n of p, 90 + 2r);
endfor
draw p scaled (384 / xpart (urcorner p - llcorner p)) withcolor .54 red;
```

The extra parameter r is used to open up the folds, making it more “organic”.



20.5 The golden dragon

You can explore variations with the recursive approach by altering the scaling and rotation. Here $r = 1/\phi^{1/\phi} \simeq 0.74274$ and $\theta \simeq 32.893°$ in the initial triangle.

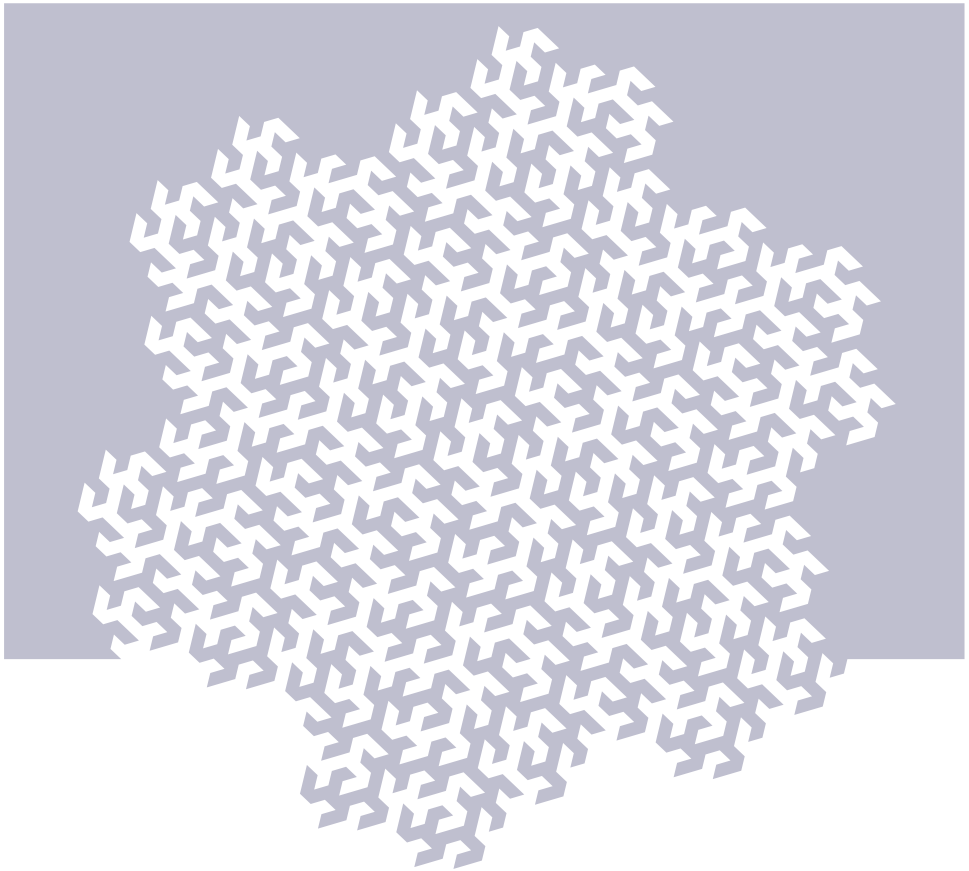


Because the initial triangle shape is shorter on one side, it is better to adapt the `dragon` routine to measure the gap between points instead of using a fixed level parameter:

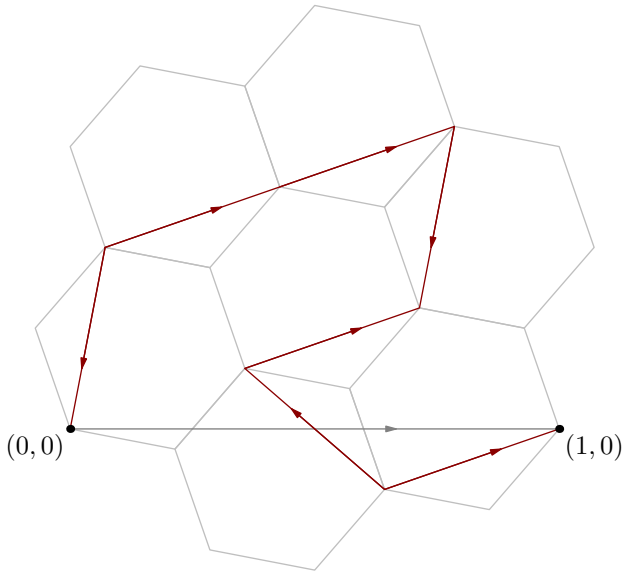
```
vardef dragon(expr a, b) =
    if abs(a-b) > 1.618:
        save p; pair p;
        p = r[a, b] rotatedabout(a, theta);
        dragon(a, p) & reverse dragon(b, p)
    else:
        a .. b
    fi
enddef;
```

20.6 The Peano-Gosper curve, or flow-snake

The Peano-Gosper curve is a space-filling curve. It is constructed in the same way as the dragons, but the generating shape has seven sections instead of two, and they are cunningly arranged to fill the space. The arrows indicate the reversals so that the scaled-down copies fill the larger part of the hexagons that contain them. →



This fourth-level curve is the boundary between the filled and unfilled branches.



Given the *snake* path scaled to unit length as shown above in red, the flow-snake on the left was created with

```
vardef rattle(expr level, a, b) =
  if level > 0:
    save s; path s; s = snake zscaled (b-a) shifted a;
    reverse rattle(level - 1, point 1 of s, a) &
            rattle(level - 1, point 1 of s, point 2 of s) &
            rattle(level - 1, point 2 of s, point 3 of s) &
            rattle(level - 1, point 3 of s, point 4 of s) &
    reverse rattle(level - 1, point 5 of s, point 4 of s) &
    reverse rattle(level - 1, point 6 of s, point 5 of s) &
            rattle(level - 1, point 6 of s, b)
  else:
    a -- b
  fi
enddef;
beginfig(1);
  pair a, b; path s;
  a = 124 left; b = 124 right; s = rattle(4, a, b);
  fill s -- b + (40, 0) -- b + (40, 224)
         -- a - (40, -224) -- a - (40,0) -- cycle
         withcolor 3/4[1/4 blue, white];
endfig;
```

20.7 Fractal trees

You can make a tree with a vertical line segment, a scaling factor $0 < r < 1$, and an angle $0° < \theta < 180°$. Start with the vertical line as the trunk, then make the first branch by scaling the trunk by r, rotating it $+\theta$, and moving it to the top of the trunk; make the second in the same way but rotate it by $-\theta$. Then repeat using each branch as a new trunk. And stop after enough levels.

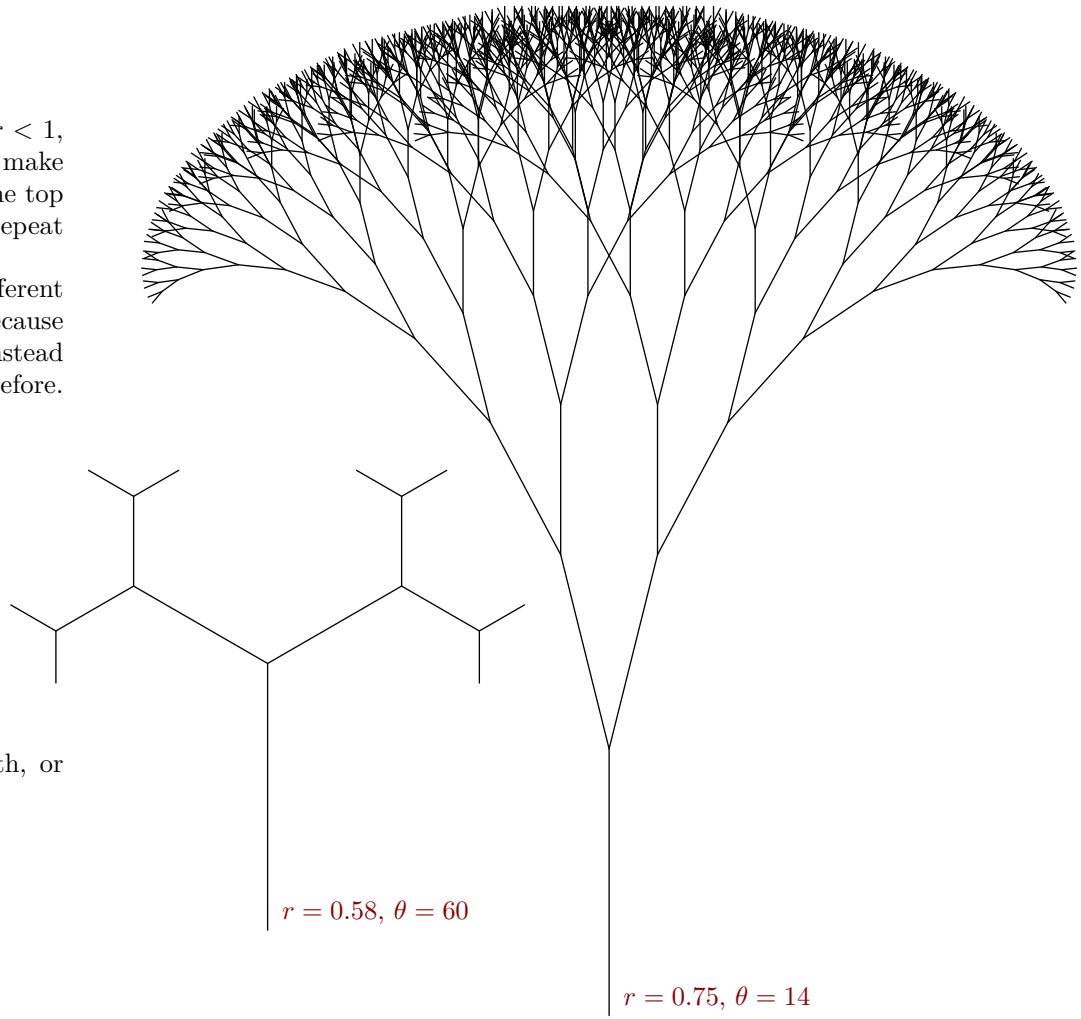
This can be implemented easily in METAPOST, but requires a slightly different technique. There is no simple way to represent the tree as a single ⟨path⟩ because a path cannot have branches, so you need to draw each segment separately instead of trying to join them up. Compare this to the `rattle` routine on the page before.

```
vardef make_tree(expr level, bar) =
  draw bar;
  if level > 0:
    for t=-theta, theta:
      make_tree(level - 1, bar shifted - point 0 of bar
        scaled r rotated t shifted point 1 of bar
      );
    endfor
  fi
enddef;
```

To combine several such trees in a drawing either move the initial *bar* path, or capture the tree as a ⟨picture⟩ using `image(make_tree(...))`, like so:

```
picture T[]; numeric r, theta;
r := 0.58; theta := 60; T1 = image(
  make_tree(3, origin -- 100 up);
);
r := 0.75; theta := 14; T2 = image(
  make_tree(10, origin -- 100 up);
);
draw T1 shifted 32 up;
draw T2 shifted 128 right;
```

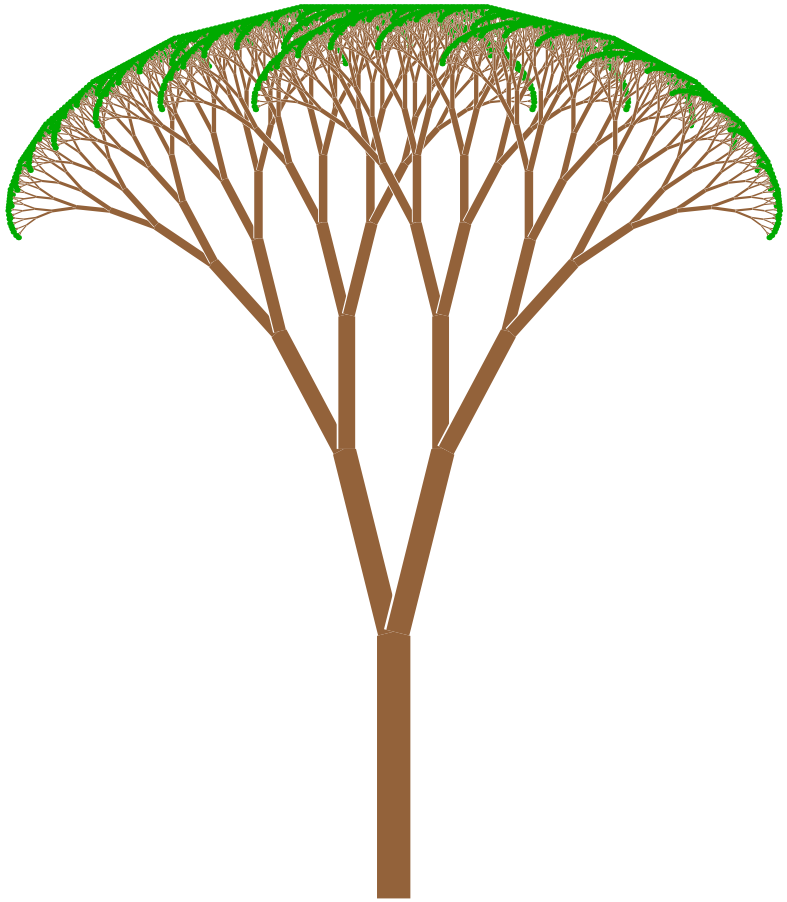
Notice that r and θ are treated as global constants. You could pass them to the `make_tree` routine instead. Notice also that the smaller tree only has three levels, but the larger one has 10.



More fractal vegetation

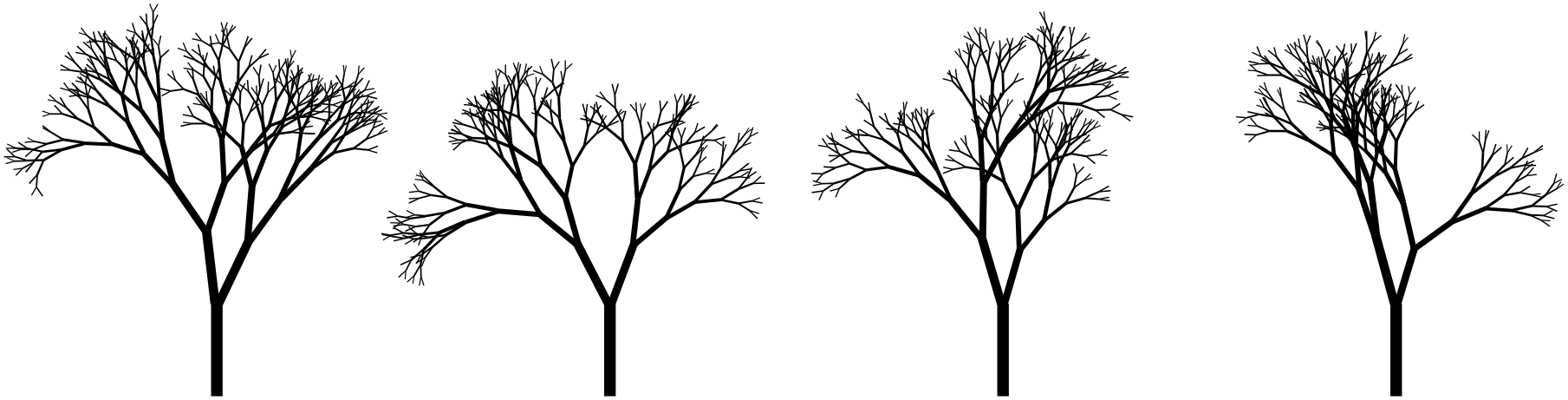
If you are more interested in the visual aspect of your tree than the mathematical, you can tweak the `make_tree` routine a bit.

```
vardef make_tree(expr bar) =
  save a; numeric a; a = abs(point 1 of bar - point 0 of bar);
  cutdraw bar withpen pencircle scaled 1.2(1/8 a) withcolor background;
  cutdraw bar withpen pencircle scaled (1/8 a) withcolor 1/256(148,98,58);
  if a > leaf:
    save s; pair s;
    s = 1/32 a * r * unitvector(direction 1 of bar) rotated 90;
    make_tree(
      bar shifted - point 0 of bar
      shifted s scaled r rotated theta
      shifted point 1 of bar
    );
    make_tree(
      bar shifted - point 0 of bar
      shifted -s scaled r rotated -theta
      shifted point 1 of bar
    );
  else:
    draw point 1 of bar withpen pencircle scaled 2 withcolor 2/3 green;
  fi
enddef;
beginfig(1);
  numeric leaf, r, theta;
  leaf = 3; r = 0.71; theta = 14;
  make_tree(origin -- 100 up);
endfig;
```



- The recursion is controlled by measuring the length of the branches instead of using a *level* parameter.
- Each branch is drawn with a pen scaled to $\frac{1}{8}$ of its length using `cutdraw`, and outlined so you can see the crossings.
- Each branch is moved slightly left or right with s to make the joins neater.
- Tiny green leaves are added on the ends of each branch.

Randomized recursive plants

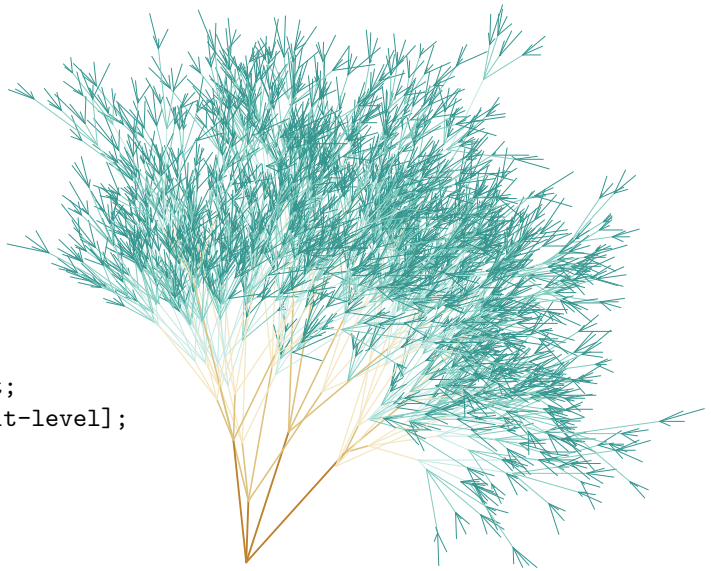


rec-general-tree-deviate.mp

To get plants that look more natural you can introduce some random factors. Strictly these are not fractal because they are not self-similar. The only change from the previous `make_tree` was to replace `scaled r rotated theta` with

```
scaled (r + 1/16 normaldeviate) rotated (theta + 8 normaldeviate)
```

The bush is similar, except that it splits into four branches at each step instead of two and only the lengths are randomized. The colouring was a happy accident.



```
input colorbrewer-rgb
vardef bush(expr start, aim, level, limit) =
  save s, target; pair target; numeric s; s = level / limit;
  for a = -32, -8, 8, 16:
    target := aim scaled ((64 + 32 normaldeviate) * s) rotated a shifted start;
    draw start -- target withpen pencircle scaled s withcolor BrBG[limit][limit-level];
    if level > 1: bush(target, aim rotated a, level - 1, limit); fi
  endfor
enddef;
beginfig(1); bush(origin, dir 80, 6, 8); endfig;
```

21 Periodic tilings

In mathematical terms, a "tiling" is a countable set of tiles that cover the plane without gaps or overlaps.[1] A *periodic* tiling is one that consists of a unit shape or pattern that is repeated by translation in two dimensions. This section loosely follows that idea, and presents some ideas and general techniques for creating tilings and other patterns or textures. —— You can make an effective grid by drawing repeated lines and then clipping to the size you want:

```
for i = -10 upto 10:
  draw (left--right) scaled 200 shifted (0, 20i);
  draw (down--up) scaled 200 shifted (20i, 0);
endfor
path c; c = fullcircle scaled 180; clip currentpicture to c; draw c;
```

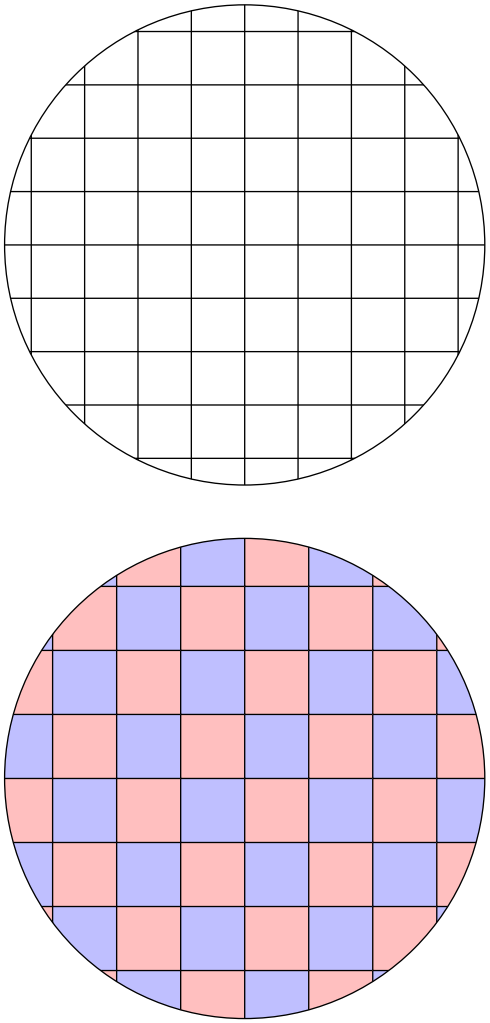
but this is a bit limited. If you want to produce more interesting periodic tilings, you need to define a unit shape or picture, and a pair of vectors to repeat it.

```
path unit; pair u, v; color a, b;
unit = unitsquare scaled 24;
u = point 1 of unit - point 0 of unit;
v = point 3 of unit - point 0 of unit;
a = 3/4[red, white]; b = 3/4[blue, white];
for i=-5 upto 5:
  for j=-5 upto 5:
    fill unit shifted (i*u + j*v)
      withcolor if odd (i+j): a else: b fi;
    draw subpath (-1,1) of unit shifted (i*u + j*v);
  endfor
endfor
path c; c = fullcircle scaled 180; clip currentpicture to c; draw c;
```

In tilings with more complex shapes you may find that using `fill` and `draw` in the same loop causes uneven lines because the fill overlaps part of the line. In these cases it is a good idea to duplicate the loops; use the first set for filling, the second for drawing.

[1] Adapted from *Tilings and Patterns*, Branko Grünbaum & G. C. Shephard, Freeman, 1987

For notes on drawing *aperiodic* tilings, see below in §22



21.1 Tiling with regular polygons

After tiling with squares, the next two simplest tilings use triangles and hexagons (made using the routines from §4.3). The basic loop is the same as the previous page except that the vectors u and v are now at 60° to each other (as shown in blue and red in the examples to the right). All of these examples were drawn with the same basic loop as before:

```
for i = -n upto n:
  for j = -n upto n:
    draw P shifted (i * u + j * v);
  endfor
endfor;
```

In the first row, P was set to a simple polygon path:

```
triangle = for i=0 upto 2: (0, 16) rotated 120i -- endfor cycle;
hexagon  = for i=0 upto 5: (0, 16) rotated  60i -- endfor cycle;
```

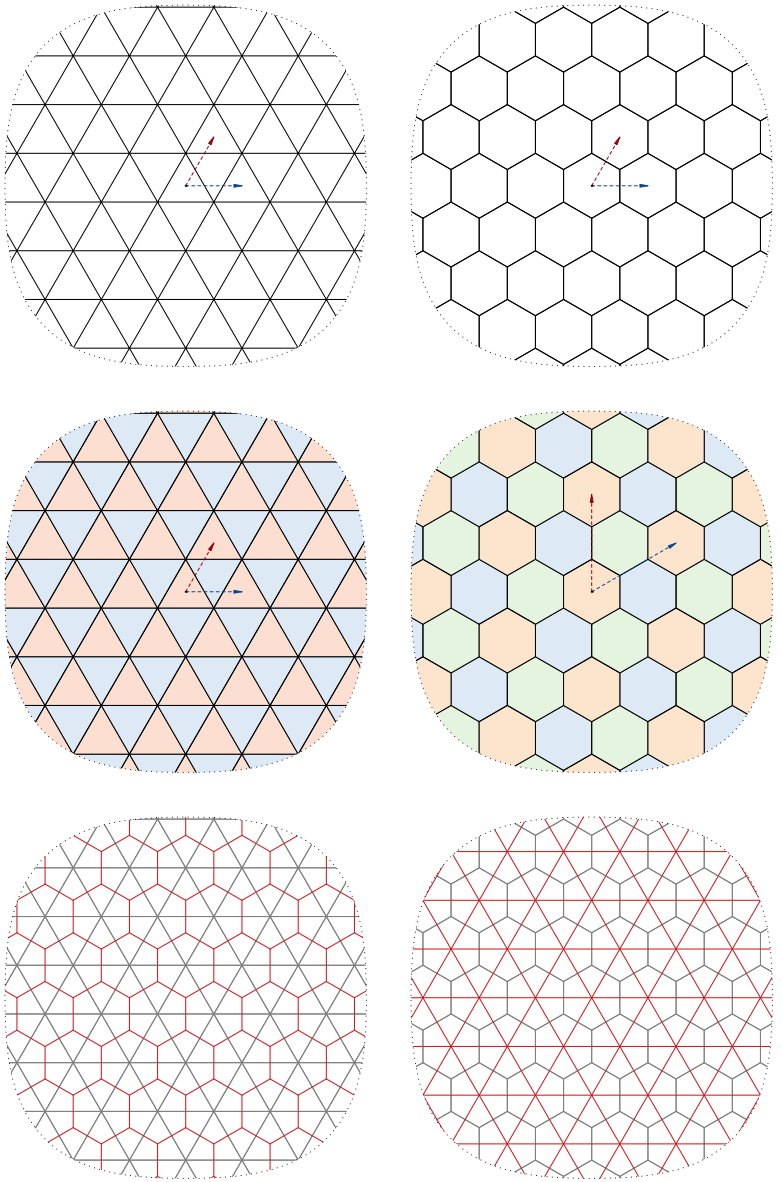
The vectors u (in red) and v (in blue) were defined as (for both tilings):

```
u = point 0 of triangle - point 1 of triangle;
v = u rotated -60;
```

To make the coloured versions, P was defined as an appropriate ⟨picture⟩. For the triangular tiling, it looked like this:  so that the tiling actually filled the plane. In the hexagonal tiling there are no gaps to fill, but in order to get a non-adjacent colouring, the unit picture was defined as three shifted copies of the hexagon each filled with a different color. The unit vectors were therefore scaled by $\sqrt{3}$ and rotated by 30° (as shown).

The bottom row the unit pictures were replaced with drawings that connect the centre of each shape to the midpoint of each side (in red), like this:

This has the effect of connecting the centres of adjacent shapes in the tiling, which reveals that each tiling is the dual of the other.

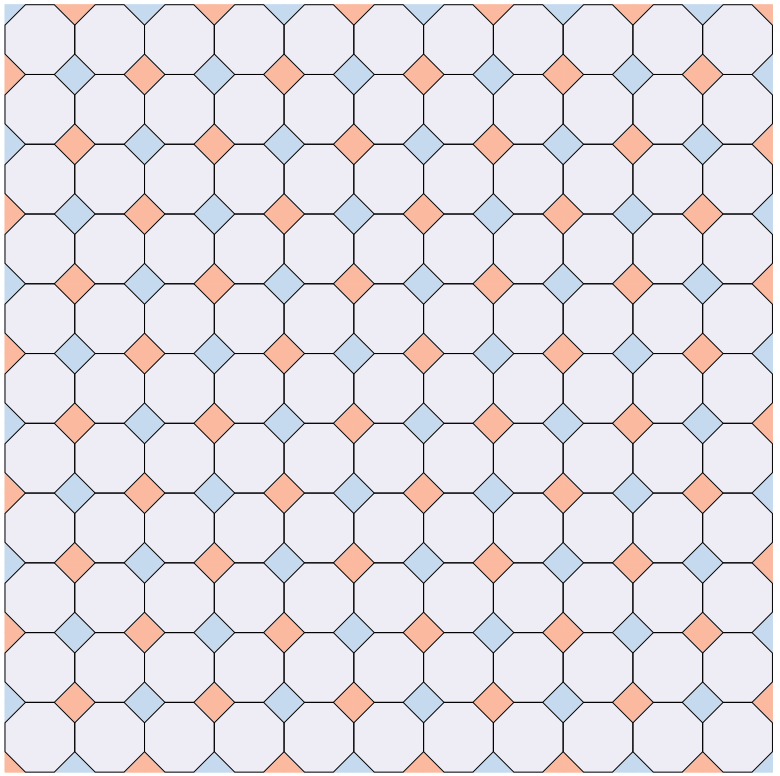


21.2 Separating filling and drawing

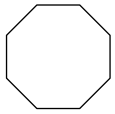
Repeating a unit image can sometimes cause unwanted overlaps, so as noted above, the solution is to make a filler unit and a drawing unit and do the filling first and the drawing second. In this example the drawing unit (the octagon) is simple so you can just draw that path instead of making another ⟨picture⟩ for it.

```
input colorbrewer-rgb
path o, r[];
o = (for i=0 upto 7: 21 dir 45i -- endfor cycle) rotated -90/4;
pair t; t = whatever[point 0 of o, point 1 of o]
= whatever[point 2 of o, point 3 of o];
r1 = subpath (1,2) of o -- t -- cycle;
r2 = r1 rotated 90;
r3 = r2 rotated 90;
r4 = r3 rotated 90;
picture filler; filler = image(
  filldraw r1 withcolor Reds 8 3;
  filldraw r2 withcolor Blues 8 3;
  filldraw r3 withcolor Reds 8 3;
  filldraw r4 withcolor Blues 8 3;
  filldraw o withcolor Purples 8 2;
);
pair u, v;
u = point 0 of o - point 5 of o; v = u rotated 90;
beginfig(1);
  numeric n;  n = 5;
  for i=-n upto n: for j=-n upto n:
    draw filler rotated ((i+j) mod 2 * 90) shifted (i*u + j*v);
  endfor endfor
  for i=-n upto n: for j=-n upto n:
    draw o shifted (i*u + j*v);
  endfor endfor
endfig;
```

Rotating every other `filler` allows you to get the alternate colours in the squares. Using `filldraw` ensures that there are no gaps between adjacent segments.



The `filler` picture (unrotated)

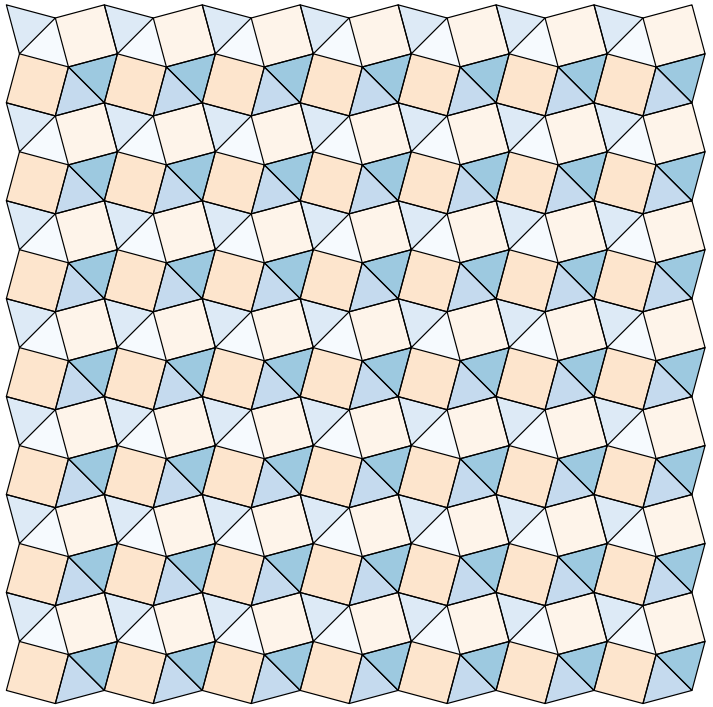


The octagon path `o`

21.3 Tilings with more complex patterns

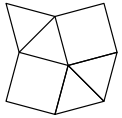
The next example also uses the square lattice, but the unit is more complicated, so the drawing needs two ⟨picture⟩ variables, one for the colour fill and a second for the grid.

```
input colorbrewer-rgb
path s[], t[];
s1 = unitsquare scaled 21 rotated 15; s2 = s1 rotated 150;
t1 = subpath (4, 3) of s1 -- point 1 of s2 -- cycle;
t2 = t1 reflectedabout(point 1 of t1, point 2 of t1);
t3 = t1 rotated 150; t4 = t1 rotated 210;
picture color_unit, grid_unit;
color_unit = image(
  fill s1 withcolor Oranges 8 1;
  fill s2 withcolor Oranges 8 2;
  fill t1 withcolor Blues 8 1;
  fill t2 withcolor Blues 8 2;
  fill t3 withcolor Blues 8 3;
  fill t4 withcolor Blues 8 4;
);
grid_unit = image(
  draw s1; draw s2;
  draw t2; draw t3; draw t4;
);
pair u, v;
u = point 1 of s1 - point 1 of s2;
v = u rotated 90;
numeric n; n = 3;
forsuffixes $=color_unit, grid_unit:
  for i=-n upto n:
    for j=-n upto n:
      draw $ shifted (i * u + j * v);
    endfor
  endfor
endfor
```



The arrangement of polygons in the units was carefully chosen to give the tiling neat edges.

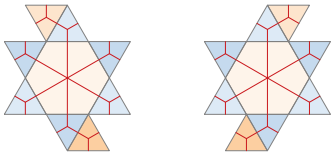
color_unit



grid_unit

21.4 Showing the dual tiling

THESE TILINGS CAN BE CLASSIFIED by the configuration of the polygons that meet at each vertex. This one is $(3^4, 6)$ because each vertex has four triangles and one hexagon. It exists in two enantiomorph forms. The unit pictures look like this:

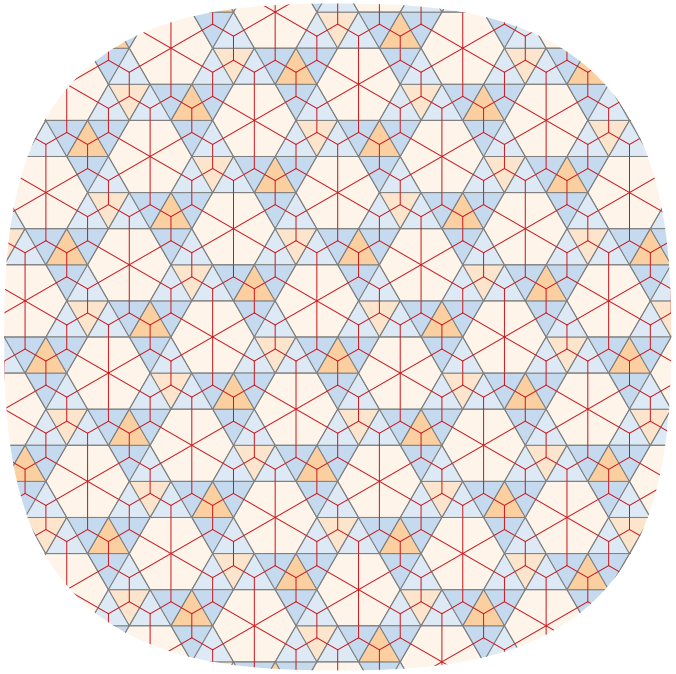


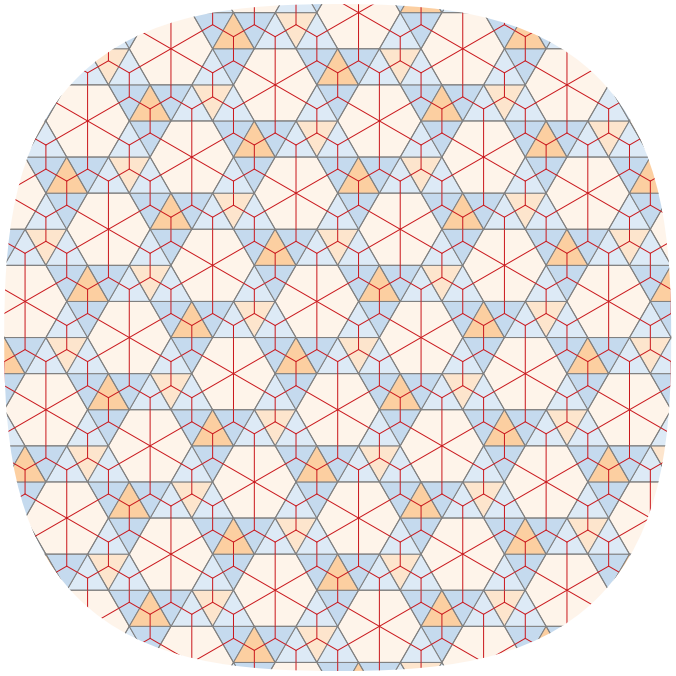
☞ *To reveal the dual of the tiling, you can draw a line from the centroid of each polygon to the midpoint of each edge.*

They are drawn like this, where h is the hexagon, $t_1 \ldots t_6$ are the blue triangles surrounding it, and t_7 & t_8 are the two "connecting" triangles, which swap sides to make the enantiomorphs.

```
unit[k] = image(
  for i=1 upto 6:
    fill t[i] withcolor Blues 8 if odd i: 2 else: 3 fi;
  endfor
  for i=7 upto 8:
    fill t[i] withcolor Oranges 8 if odd i: 3 else: 2 fi;
  endfor
  fill h withcolor Oranges 8 1;
  forsuffixes $=h, t1, t2, t3, t4, t5, t6, t7, t8:
    draw $ withpen pencircle scaled 1/4 withcolor 1/2;
    for i=1 upto length $:
      draw centroid($) -- point i - 1/2 of $ withcolor Reds 7 6;
    endfor
  endfor
);
```

The *centroid*() routine is from §4.3 and the colours are from §7.4. This tiling is generated using the loop-with-triangular-grid-vectors from §21.1.



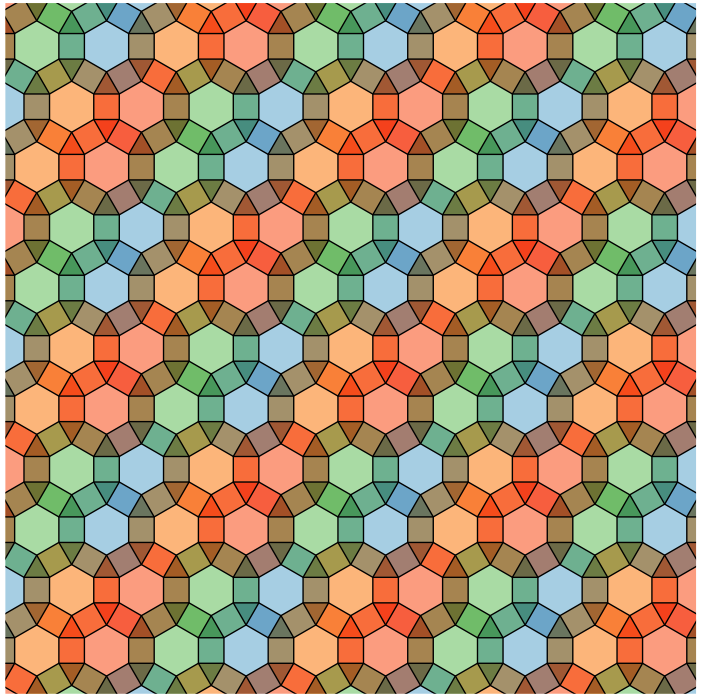


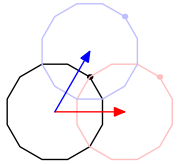
21.5 Tiling with a dynamic unit

In order to reveal patterns in a tiling, you might want to vary the colours or line styles used in each repeated drawing unit. In this case, you can write a macro that takes a parameter and returns a picture to draw.

```
def withalpha expr a =  % <-- this requires "luamplib"
  withprescript "tr_alternative=2"
  withprescript "tr_transparency=" & decimal a
enddef;
input colorbrewer-rgb
path d; d = for i=1 upto 12: 18 dir (15+30i) -- endfor cycle;

color shade[];
shade0 = Oranges 8 4; shade1 = Blues 8 4;
shade2 = Greens 8 4; shade3 = Reds 8 4;

vardef unit(expr n) = image(
  fill d withalpha 0.9 withcolor shade[n mod 4];
  draw d;
) enddef;
pair u, v;
u = point 0 of d - point 3 of d;
v = u rotated 60;
numeric n; n = 6;
for i=-n upto n:
  for j=-n upto n:
    draw unit(3i-7j) shifted ((i-floor(j/2)) * u + j * v);
  endfor
endfor
clip currentpicture to unitsquare shifted -(1/2, 1/2) scaled 42n;
```

Instead of defining six triangles, six squares, and a hexagon, you can just define the dodecagon and overlap each one. Using a macro to create the unit, allows you to choose a different colour for each filler. Using the transparency support from `luamplib` automatically mixes the colours for the overlaps. But the edges don't look so good, so you need to clip the whole picture to a neat square.





The vectors are chosen so that the dodecagons overlap to make the required triangles, squares, and hexagons.

21.6 Colouring with a set number of colours

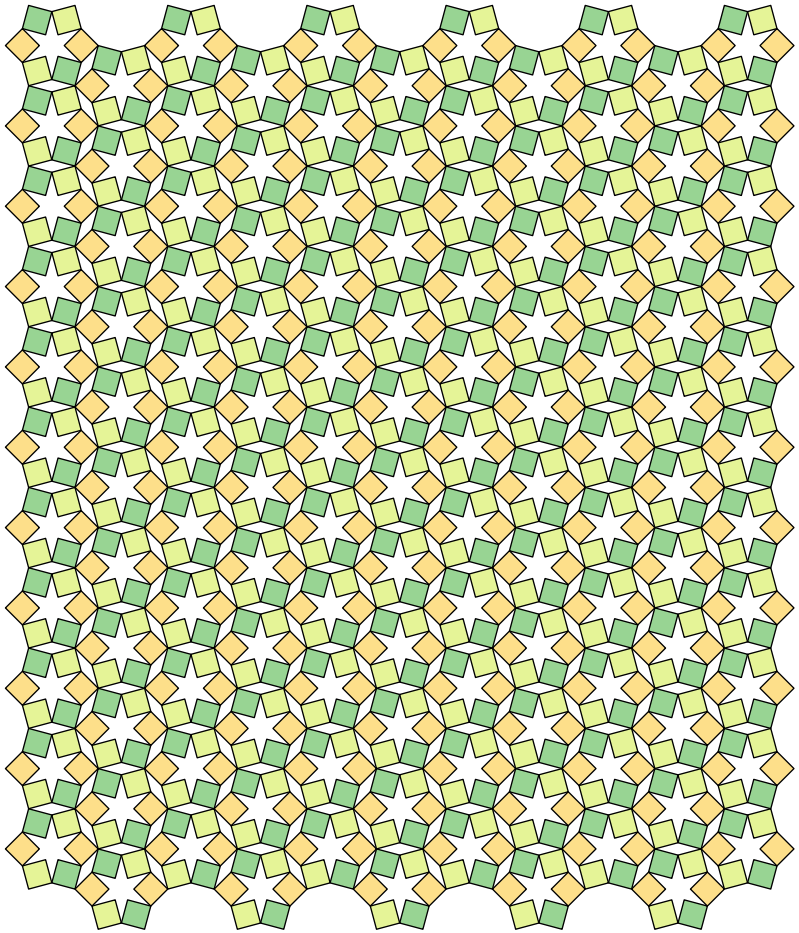
This "attractive and ingenious" pattern of squares is given in Grünbaum & Shephard, chapter 8, and is shown here coloured with three colours so that no two squares with the same colour touch each other.

```
input colorbrewer-rgb
beginfig(1);
  numeric r, s; s = 9; r = 2s * (sind(75)-sind(45));
  path a; a = unitsquare rotated -45 scaled s shifted (r, 0);
  picture unit; unit = image(
    for t=0 upto 5:
      fill a rotated 60t withcolor Spectral[6][t mod 3 + 3];
      draw a rotated 60t;
    endfor
  );
  pair u, v;
  u = (2s * sind(75), 0); u := u + u rotated 60; u := u rotated 60;
  v = u rotated 60;
  numeric n; n = 5;
  for i = -n upto n:
    for j = -n upto n:
      draw unit shifted (i*u + j*v - floor(j/2)*u);
    endfor
  endfor
endfig;
```

The points of interest here are

- The use of the `sind` function to position the square in the unit so that it makes the six-pointed star when rotated.
- The expression "`t mod 3 + 3`" which cycles through 3, 4, and 5 for different values of t.

Many different colourings are possible but you might need to make a more complex unit drawing for some of them.



22 Aperiodic tilings

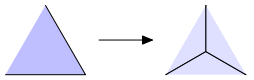
Mathematical research in the 1960s and 1970s established that it is also possible to tile an infinite plane with polygons that do not have any periodic repeating pattern. There are various mathematical approaches to creating these tilings; this section explores substitution tilings, which lend themselves to recursive programming. The rules for a substitution define a dissection for each shape that makes up the tiling.

The idea is to define a recursive macro for each shape in the tiling, with parameters for the recursion level and points that define the shape. If $level = 0$, draw or decorate that particular shape, otherwise work out the points for the dissection and call the appropriate macros for each shape with $level - 1$. Here is an example:

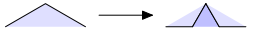
```
vardef tall(expr level, a, b, c) =
  if level = 0:
    fill a--b--c--cycle  withcolor 3/4[blue, white];
    draw a--c--b;
  else:
    save m; pair m; m = 1/3 (a + b + c);  % the centroid
    wide(level - 1, a, b, m);
    wide(level - 1, b, c, m);
    wide(level - 1, c, a, m);
  fi
enddef;
vardef wide(expr level, a, b, c) =
  if level = 0:
    fill a--b--c--cycle withcolor 7/8[blue, white];
    draw a--c--b;
  else:
    save p, q; pair p, q; p = 1/3[a,b]; q = 1/3[b,a];
    wide(level - 1, c, a, p);
    tall(level - 1, p, q, c);
    wide(level - 1, b, c, q);
  fi
enddef;
```

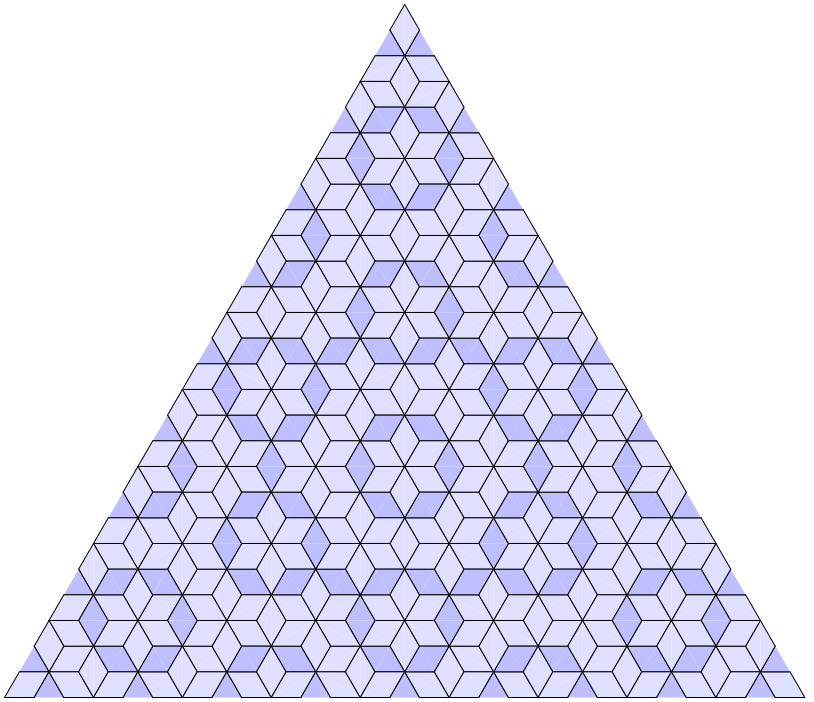
These procedures are illustrated on the right. The completed tiling was drawn with `tall(6, 173 dir 210, 173 dir 330, 173 dir 90)`.

The *tall* macro dissects a tall triangle into 3 wide ones



The *wide* macro dissects a wide triangle into 2 wides, and 1 tall.

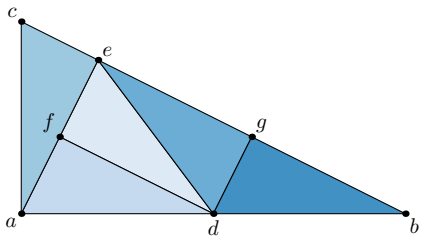




Note that only part of the perimeter of each triangle is drawn, to give the illusion that the completed tiling is made up of identical rhombs.

22.1 Pinwheel tiling

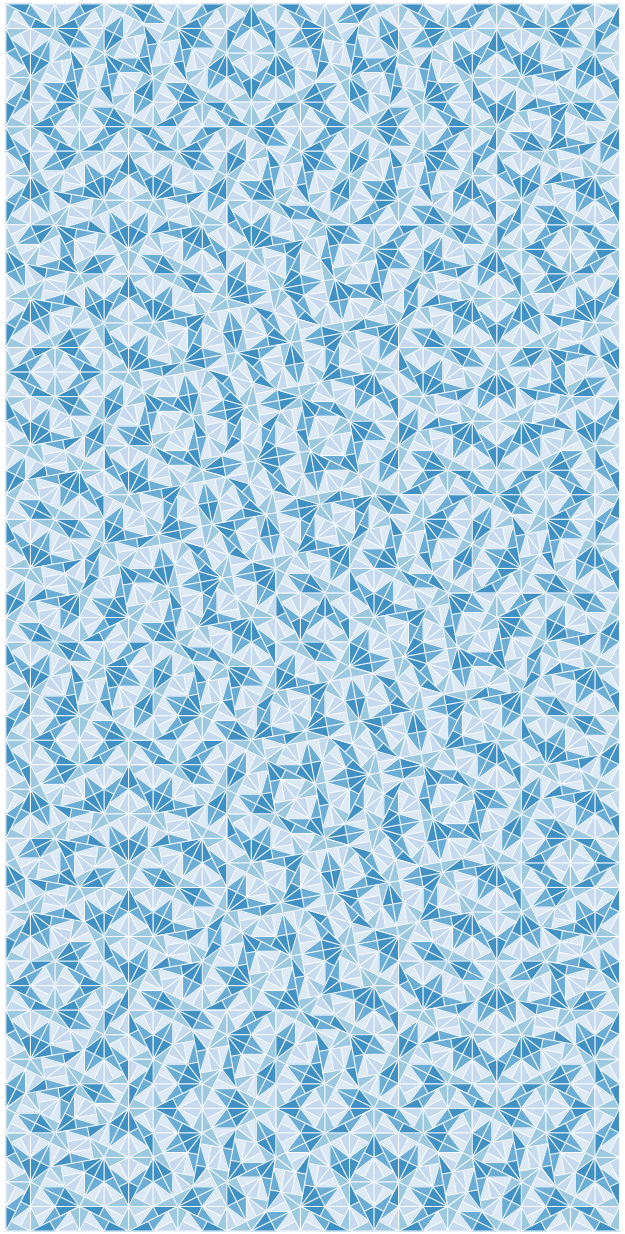
The second aperiodic tiling example is the so-called pinwheel tiling, devised in 1994 by Charles Radin, based on this dissection by John Conway.



Starting with a triangle of this shape, the tiling recursively divides into five smaller copies of itself. The colouring is passed down each level but only used on the lowest.

```
input colorbrewer-rgb
vardef pinwheel(expr level, a, b, c, s) =
  if level = 0:
    fill a--b--c--cycle withcolor s;
    draw a--b--c--cycle;
  else:
    save d, e, f, g; pair d, e, f, g;
    d = 1/2[a,b]; e = 1/5[c,b]; f = 1/2[a,e]; g = 1/2[b,e];
    pinwheel(level - 1, f, d, e, Blues 9 2);
    pinwheel(level - 1, f, d, a, Blues 9 3);
    pinwheel(level - 1, e, a, c, Blues 9 4);
    pinwheel(level - 1, g, e, d, Blues 9 5);
    pinwheel(level - 1, g, b, d, Blues 9 6);
  fi
enddef;
beginfig(1);
  pair a, b, c; a = origin; b = 460 right; c = 1/2 b rotated 90;
  drawoptions(withpen pencircle scaled 1/8 withcolor white);
    pinwheel(5, a, b, c, "");
  drawoptions();
  draw currentpicture rotatedabout(1/2[b,c], 180);
  currentpicture := currentpicture rotated 90;
endfig;
```

☞ To make the dissection work, it is important to pass the three ⟨pair⟩ arguments in the right order. Note also the manipulation of `currentpicture` at the end.

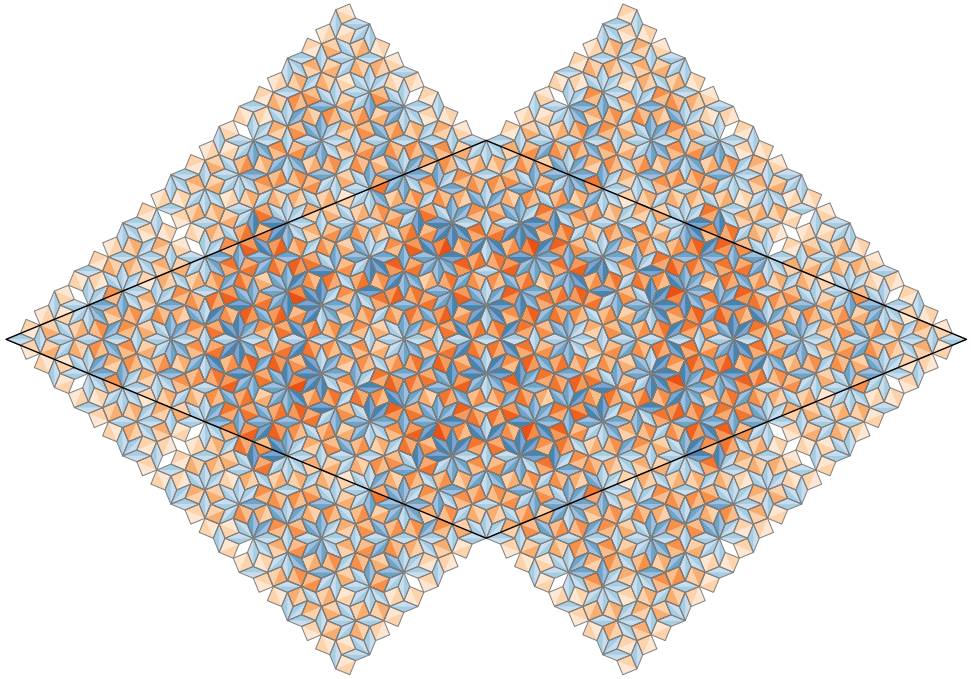


22.2 Ammann A5 tiling

Of the various sets of aperiodic tiles discovered by Robert Ammann in 1977, the best known and probably most attractive is the set known as A5. The recursive substitutions for the tiling are usually given as follows,

with the square shapes overlapping the edges of the parent shapes. This can be implemented with two co-operating recursive macros ⟶

The overlaps mean that much of the tiling is overlaid by other expansions which makes the resulting PDF rather large, and that you need to clip the final picture to the original shape. The uneven texture is apparent if you colour the tiles with transparent colours (although this is not unattractive):

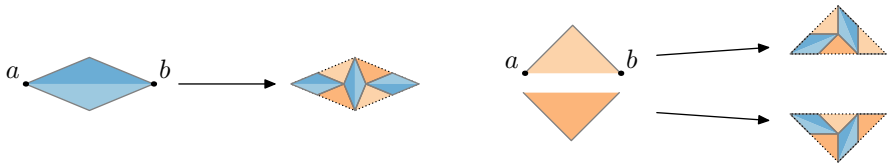


This needs the `withalpha` *macro given in §12.7.*

```
path r, s;
r = origin -- (1/2, 1/2-sqrt(1/2)) -- right -- (1/2, sqrt(1/2)-1/2) -- cycle;
s = origin -- (1/2, -1/2) -- right -- (1/2, 1/2) -- cycle;
numeric p, q;
p = sqrt(2) - 1; q = 1 - sqrt(1/2);
vardef rhomb(expr level, a, b) =
  save R; path R; R = r zscaled (b-a) shifted a;
  if level = 0:
    fill subpath (0, 2) of R -- cycle withalpha 0.9 withcolor Blues 9 3;
    fill subpath (2, 4) of R -- cycle withalpha 0.9 withcolor Blues 9 4;
    draw R withcolor 1/2;
  else:
    rhomb(level-1, a, p[a,b]);
    rhomb(level-1, b, p[b,a]);
    rhomb(level-1, point 1 of R, point 3 of R);
    square(level-1, point 1 of R, point +p of R);
    square(level-1, point 3 of R, point -p of R);
    square(level-1, point 1 of R, point 2-p of R);
    square(level-1, point 3 of R, point 2+p of R);
  fi
enddef;
vardef square(expr level, a, b) =
  save S; path S; S = s zscaled (b-a) shifted a;
  if level = 0:
    fill subpath (0, 2) of S -- cycle withalpha 0.9 withcolor Oranges 8 3;
    fill subpath (2, 4) of S -- cycle withalpha 0.9 withcolor Oranges 8 2;
    draw S withcolor 1/2;
  else:
    rhomb(level-1, a, q[point 1 of S, point 3 of S]);
    rhomb(level-1, a, q[point 3 of S, point 1 of S]);
    rhomb(level-1, point 1 of S, q[b,a]);
    rhomb(level-1, point 3 of S, q[b,a]);
    square(level-1, q[b,a], q[a,b]);
    square(level-1, point 1 of S, point +p of S);
    square(level-1, point 3 of S, point -p of S);
    square(level-1, b, point +1+p of S);
    square(level-1, b, point -1-p of S);
  fi
enddef;
```

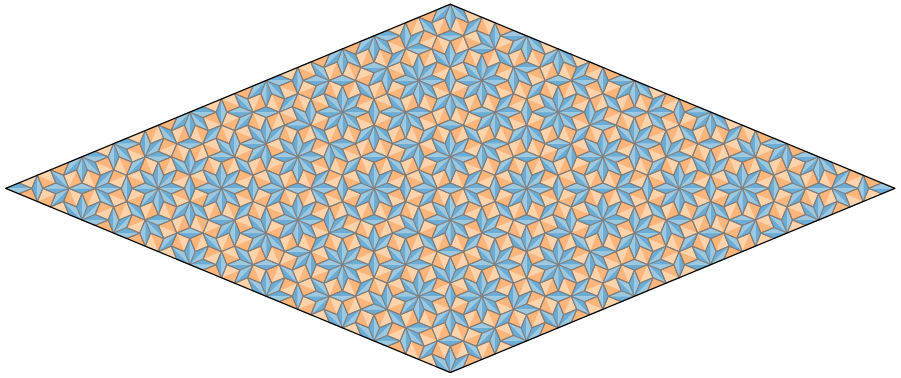
22.3 Ammann A5 tiling improved

With some extra thought, you can devise a substitution pattern that avoids the overlaps and makes a smaller tiling. The idea is to split the square in half, and make a macro that can do the top half or the bottom half as required.



This requires an extra parameter for the new `half_square` macro to tell it which half to do, and two triangular paths for the upper and lower halves. Note that the second of these paths is reversed, which simplifies the macro.

Now there is no overlapping of any of the tiles and the tiling does not overflow outside the original shape, and the colouring is uniform.



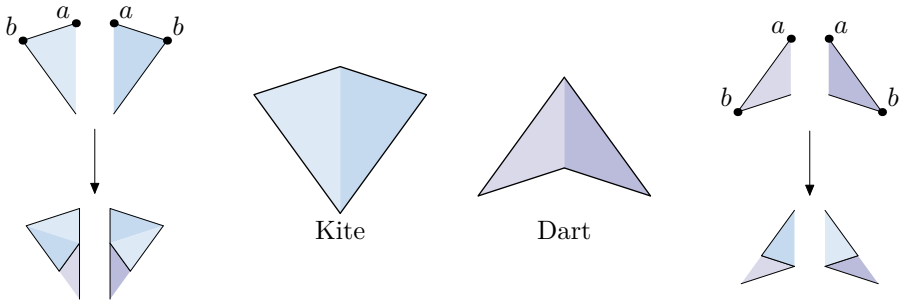
In general, aperiodic tilings work best if you can find a set of recursive substitutions that are bounded within each parent shape.

☞ Notice also the useful idiom `zscaled (b-a) shifted a` used with a "unit" shape.

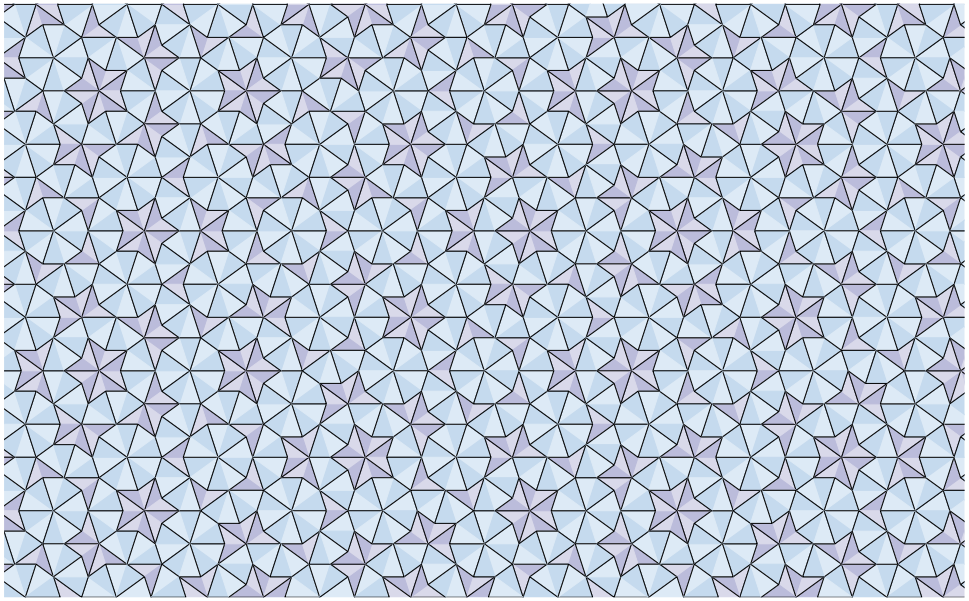
```
path r, s;
r = origin -- (1/2, 1/2-sqrt(1/2)) -- right -- (1/2, sqrt(1/2)-1/2) -- cycle;
s = origin -- (1/2, -1/2) -- right -- (1/2, 1/2) -- cycle;
numeric p, q;
p = sqrt(2) - 1; q = 1 - sqrt(1/2);

path t[];
t0 = origin -- right -- point 3 of s -- cycle;
t1 = origin -- right -- point 1 of s -- cycle;
vardef bounded_rhomb(expr level, a, b) =
  save R; path R; R = r zscaled (b-a) shifted a;
  if level = 0:
    fill subpath (0, 2) of R -- cycle withcolor Blues 9 4;
    fill subpath (2, 4) of R -- cycle withcolor Blues 9 5;
    draw R withcolor 1/2;
  else:
    bounded_rhomb(level-1, a, p[a,b]);
    bounded_rhomb(level-1, b, p[b,a]);
    bounded_rhomb(level-1, point 1 of R, point 3 of R);
    half_square(level-1, point 1 of R, point +p of R, 1);
    half_square(level-1, point 3 of R, point -p of R, 0);
    half_square(level-1, point 3 of R, point 2+p of R, 1);
    half_square(level-1, point 1 of R, point 2-p of R, 0);
  fi
enddef;
vardef half_square(expr level, a, b, side) =
  save T; path T; T = t[side] zscaled (b-a) shifted a;
  if level = 0:
    fill T withalpha 0.9 withcolor Oranges[8][3+side];
    draw subpath (1, 3) of T withcolor 1/2;
  else:
    bounded_rhomb(level-1, point 0 of T, p[point 1/2 of T, point 2 of T]);
    bounded_rhomb(level-1, point 2 of T, point sqrt(1/2) of T);
    half_square(level-1, point 1 of T, point 2-p of T, side);
    half_square(level-1, point 2 of T, point 3-p of T, side);
    half_square(level-1, point 1-q of T, point q of T, 1-side);
  fi
enddef;
```

22.4 Penrose P2 tiling

THE SAME TECHNIQUES ARE USEFUL for generating the well-known aperiodic tilings discovered in the 1970s by the British polymath Sir Roger Penrose. This P2 tiling is also known as the "kite and dart" tiling. The expansion rules used here are those documented by Simon Tatham.



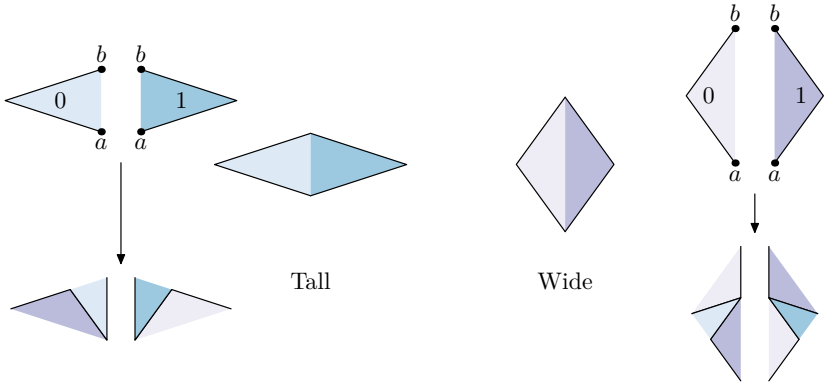
Each of the shapes is split into two opposite triangles and the expansion rules applied separately. Only part of the edge of each triangle is drawn, so that the proper shapes appear in the final result. The patch below was clipped to a rectangle.



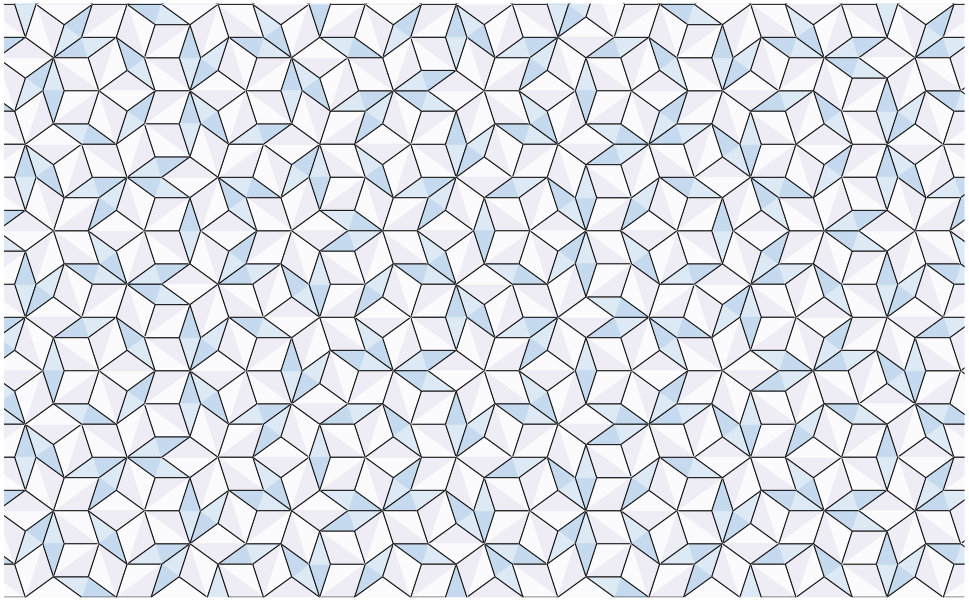
```
input colorbrewer-rgb

numeric phi; phi = 0.61803398875;

path wide[], tall[];
wide0 = origin -- right -- phi * dir 36 -- cycle;
wide1 = origin -- right -- phi * dir -36 -- cycle;
tall0 = origin -- right -- (1+phi) * dir 72 -- cycle;
tall1 = origin -- right -- (1+phi) * dir -72 -- cycle;

vardef half_dart(expr level, a, b, side) =
  save T; path T; T = wide[side] zscaled (b-a) shifted a;
  if level = 0:
    fill T withcolor Purples[9][3+side];
    draw subpath (0, 2) of T;
  else:
    half_dart(level - 1, point 1 of T, point 2   of T, side);
    half_kite(level - 1, point 2 of T, point phi of T, 1-side);
  fi
enddef;

vardef half_kite(expr level, a, b, side) =
  save T; path T; T = tall[side] zscaled (b-a) shifted a;
  if level = 0:
    fill T withcolor Blues[9][2+side];
    draw subpath (0, 2) of T;
  else:
    half_dart(level - 1, point 2     of T, point 2+phi of T, side);
    half_kite(level - 1, point 2+phi of T, point 0     of T, side);
    half_kite(level - 1, point 2+phi of T, point 1+phi of T, 1-side);
  fi
enddef;

beginfig(1);
  numeric wd; wd = 5in;
  drawoptions(withpen pencircle scaled 1/8);
    half_dart(9, 1.375 wd * left, 1.375 wd * right, 0);
  drawoptions();
  clip currentpicture to unitsquare shifted 1/2 left scaled wd yscaled phi;
endfig;
```

22.5 Penrose P3 tiling

The final example is Penrose's P3 tiling, composed of thick and thin rhombus shapes. There is no new METAPOST technique, but it is perhaps of mathematical interest that the code for P3 is so similar to P2 on the previous page.



Each of the shapes is again split into two opposite triangles and the expansion rules applied separately, and only part of the edge of each triangle is drawn, so that the proper shapes appear in the final result.



```
input colorbrewer-rgb

numeric phi; phi = 0.61803398875;

path wide[], tall[];
wide0 = origin -- right -- phi * dir 36 -- cycle;
wide1 = origin -- right -- phi * dir -36 -- cycle;
tall0 = origin -- right -- (1+phi) * dir 72 -- cycle;
tall1 = origin -- right -- (1+phi) * dir -72 -- cycle;

vardef half_tall(expr level, a, b, side) =
  save T; path T; T = tall[side] zscaled (b-a) shifted a;
  if level = 0:
    fill T withcolor Blues[9][2+side];
    draw subpath (1, 3) of T;
  else:
    half_tall(level - 1, point 1 of T, point 2-phi of T, side);
    half_wide(level - 1, point 0 of T, point 2     of T, 1-side);
  fi
enddef;

vardef half_wide(expr level, a, b, side) =
  save T; path T; T = wide[side] zscaled (b-a) shifted a;
  if level = 0:
    fill T withcolor Purples[9][1+side];
    draw subpath (1, 3) of T;
  else:
    half_tall(level - 1, point 2   of T, point -phi of T, side);
    half_wide(level - 1, point phi of T, point  0   of T, 1-side);
    half_wide(level - 1, point 1   of T, point  2   of T, side);
  fi
enddef;

beginfig(1);
  numeric wd; wd = 5in;
  drawoptions(withpen pencircle scaled 1/8);
    half_wide(8, 1.375 wd * left, 1.375 wd * right, 0);
  drawoptions();
  clip currentpicture to unitsquare shifted 1/2 left scaled wd yscaled phi;
endfig;
```

Contents