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 work-flow 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.

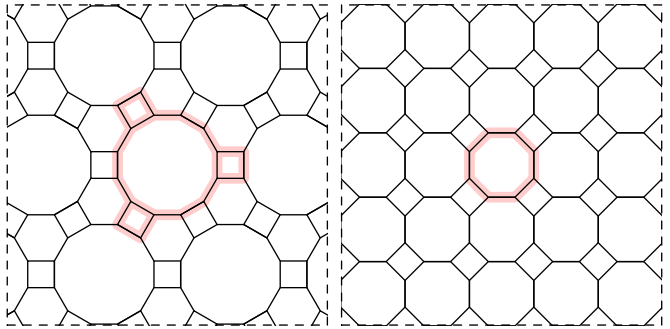
These notes are based on the many examples I have developed as answers to questions about technical drawing on the wonderful TEX StackExchange site. In accordance with their terms and conditions, I've only included material here that I've written myself — if you want other people's code then visit the 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.

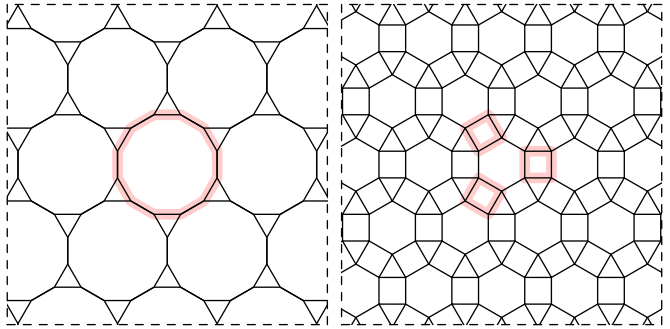
And when I say *plain* METAPOST I mean METAPOST with the default format (as defined in the file `plain.mp`) loaded and no other external packages (apart from `boxes.mp` very occasionally). Nearly all of the examples here are supposed to be self contained, and any macros are defined locally so you can 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; in contrast I've tried to be as clear as possible in my examples.

Drawing with Metapost

Toby Thurston— September 2015



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



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

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 equations like `a=-b;`. After this, as soon as you give a value to `a`, METAPOST immediately works out the value of `b`.

- Variable types:

  - numeric
  - pair
  - path
  - transform
  - color
  - string
  - 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.

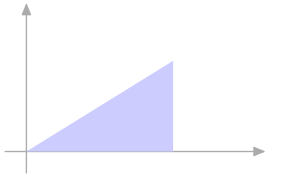
- Implicit multiplication: METAPOST inherits a rich set of rules about numerical expressions from METAFONT, and of special interest is the scalar multiplication operator. Any simple number, like `42`, `3.1415`, or `.6931`, or any simple fraction like `1/2` or `355/113` standing on it's own (technically 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 additictive after a while.

3 Making and using closed paths

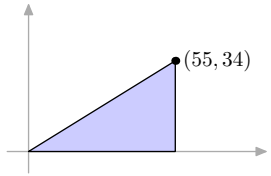
In METAPOST there are two sorts of paths: open and closed. A closed path is called a cycle, and is created with the `cycle` primitive like this:

```
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'. 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'. You can also use `cycle` in a boolean context to test whether a path `p` is cyclic: `if cycle p: ... fi`.



fill t **withcolor** $0.8[blue, white]$;



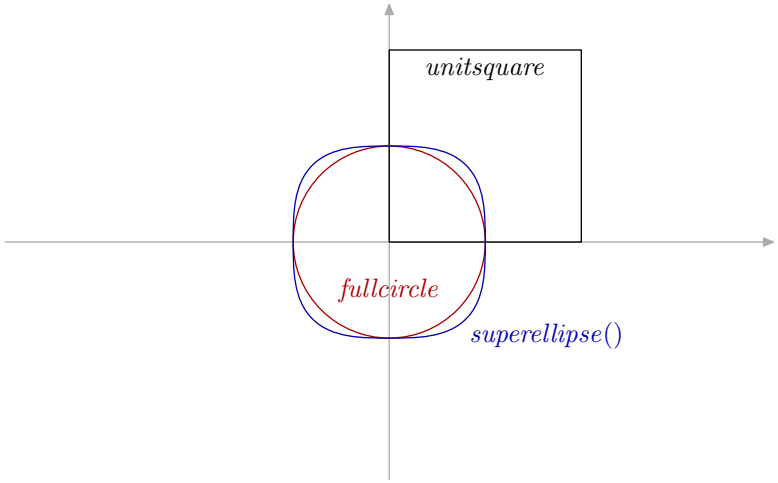
... **draw** t;

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

- `unitsquare` which you can use to draw any rectangle with appropriate use of `xscaled` and `yscaled` — it's defined so that the bottom left corner is point 0 of the shape. This point is also defined as `(0,0)`, so the `unitsquare` is centred on point `(1/2,1/2)`. If you want a square centered on the origin, then shift it by `-(1/2,1/2)` before you scale it.
- `fullcircle` which you can use to draw any circle or ellipse with appropriate use of `xscaled` and `yscaled`. Defined so that it is centered at the origin and has unit *diameter* and point 0 is `(1/2,0)`.
- `superellipse()` which creates the shape beloved of the Danish designer Piet Hein. Unlike the other two, this one is a function rather than a path, so you need to call it like this:

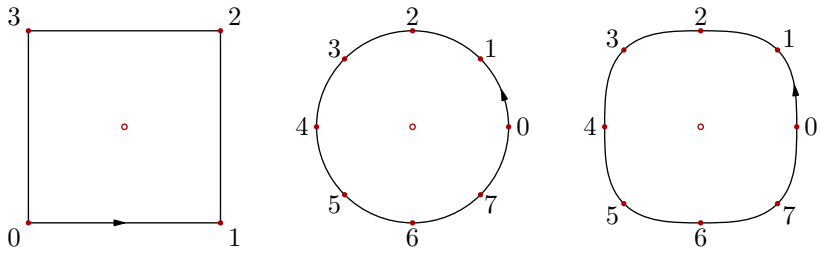
  ```
  path s; s = superellipse(right,up,left,down,.8);
  ```

  to create a 'unit' shape. The fifth parameter is the 'superness': the value 1 makes it look almost square, 0.8 is about right, 0.5 gives you a diamond, and values outside the range $(0.5, 1)$ give you rather weird propeller shapes.



3.1 Points on the standard closed paths

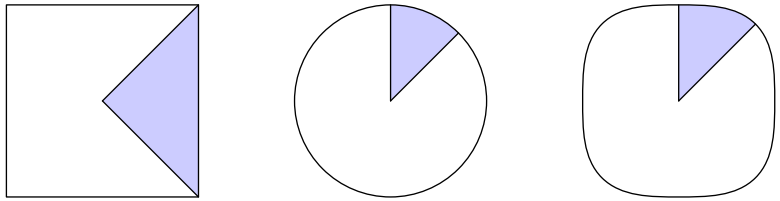
Here are the three shapes centered on the origin and labelled to show the points along them. **Note** that the *unitsquare* shape has been shifted so that it is centered on the origin in all of these examples. The small red circle marks the *origin*, and the labelled red dots are the points of each path. 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:

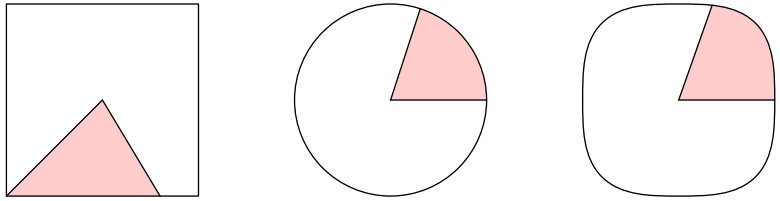
```
center p -- subpath(1,2) of p -- cycle
```

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

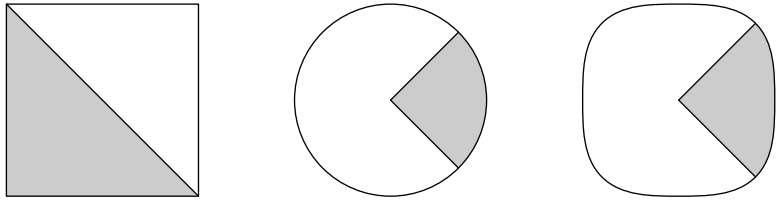
```
center p -- subpath(0,1/5 length p) of p -- cycle
```



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 a circle, then `point 1 of c` is 45° round and 1 radian is `point 1.27324 of c`.

In a cyclic 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 $n +$ **length** p. This works with negative numbers too, so we could use

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



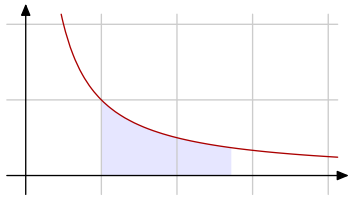
to get wedge centered on point 0.

3.2 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 cyclic 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 blue area was defined with

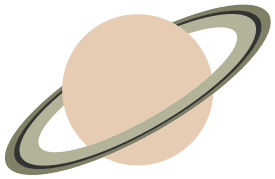
```
buildcycle(yy shifted (1u,0), f, yy shifted (2.71828u,0), xx)
```

Note the use of the y-axis 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 illlusion of depth to a simple graphic of the planet Saturn.

```
prologues:=3; outputtemplate := "%j%c.eps";
beginfig(1);
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;
endfig;
end
```



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` is shorthand for `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.

3.3 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.

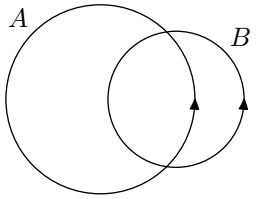
If all has gone well, then `ta` will hold all the start points of the desired subpaths, and `tb` all the corresponding end points. The third and final `for` loop assumes that this is indeed the case, and tries to connect them all together. Note that it uses `..` rather than `&` just in case the points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

3.4 Strange behaviour of `buildcycle` with two cyclic paths

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

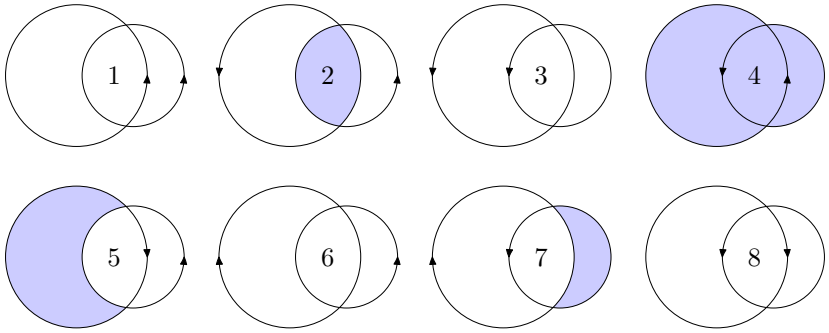
```
beginfig(1);
  path A, B;
  A = fullcircle scaled 2.5cm;
  B = fullcircle scaled 1.8cm shifted (1cm,0);
  fill buildcycle(A,B) withcolor .8[blue,white];
  drawarrow A; drawarrow B;
endfig;
```



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 cyclic 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 it's 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, use: `p rotatedabout(center p, 180)`

3.5 Find the overlap of two cyclic paths

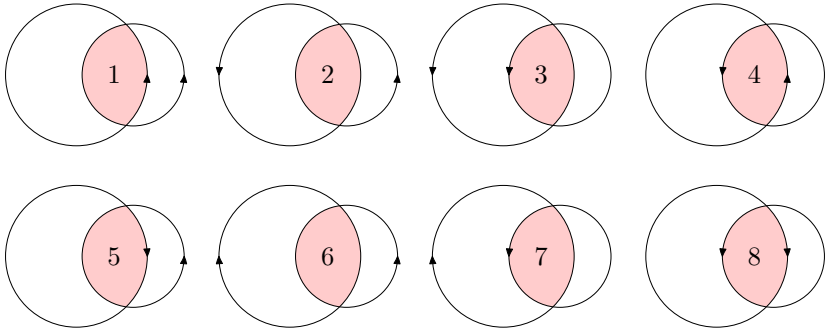
As we have seen, in order to get the overlap of two cyclic paths from `buildcycle`, we need both paths to be running in the same direction, and neither path should start inside the other one. It's not hard to create an `overlap` macro that does this automatically for us. The first element we need is 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 cyclic 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 cyclic 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.

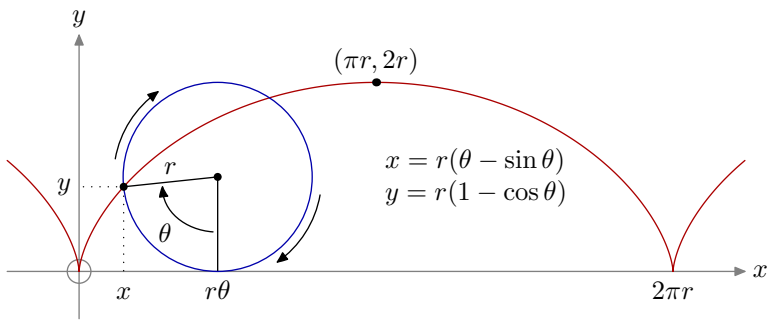
```
vardef inside(expr p, ring) =
  save t, count, test_line;
  count := 0;
  path test_line;
  test_line = p -- (infinity, ypart p);
  for i = 1 upto length ring:
     t := xpart(subpath(i-1,i) of ring
                  intersectiontimes test_line);
     if ((0<=t) and (t<1)): count := count + 1; fi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



4 Cycloid and other spiral graphs

This section examines cycloids; 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.

```
prologues := 3; outputtemplate := "%j%c.eps";
beginfig(1);
r = 1.25cm; % radius of the wheel
pi = 3.14159265; % constant

% define the cycloid
path c;
c = (0,-r) rotated 100 shifted (r*-100/180*pi,r)
    for t=-99 upto 460:
      -- (0,-r) rotated -t shifted (r*t/180*pi,r)
    endfor;

% axes
drawoptions(withcolor .5 white);
path xx, yy;
yy = (down -- 5 up) scaled 1/2 r;
xx = (xpart point 0 of c, 0) -- (xpart point infinity of c,0);
draw fullcircle scaled 1/4r; drawarrow xx; drawarrow yy;
drawoptions();
label.rt (btex $x$ etex, point 1 of xx);
label.top(btex $y$ etex, point 1 of yy);

% draw the cycloid on top of the axes
draw c withcolor .67 red;

% define a couple of related points:
% z1 center of the blue wheel
% z2 intersection of rim and cycloid
t = 84; % if you change t then the wheel will "roll" along...
```



- Near the beginning we define $\pi = 3.14159265$, as there's no such constant built in, but it makes the source more understandable to write `pi/180` instead of `0.017453`. It would be nice to use the Greek letters themselves in the source, but METAPOST only lets you use plain ASCII characters, to you have to write `pi` instead. Later on `t` is used instead of θ.

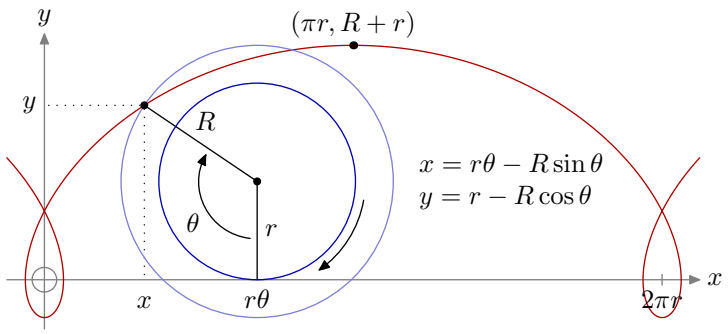
- The cycloid path `c` is defined using an inline `for` loop. There's a slight awkwardness to doing this as you have to repeat yourself either at the beginning or the end, because you can't have a dangling `--` or `..` at the end of the path. With a cyclic path it's easier because you can just put `--cycle` after the `endfor`. The strange numbers here are because we are going from a rotation of $-100°$ to $+460°$; $360°$ 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 fill a circle scaled to `dotlabeldiam`; this internal parameter is the current size to be used for the dots in `dotlabel`.

```
z1 = (r*t/180*pi,r);
z2 = (0,-r) rotated -t shifted z1;

% draw the auxiliary lines
draw (0,y2) -- z2 -- (x2,0) dashed withdots scaled .6;
draw z2 -- z1 -- (x1,0);

% draw the rolling circle and mark the centre and intersection with cycloid
draw fullcircle scaled 2r           shifted z1 withcolor .67 blue;
fill fullcircle scaled dotlabeldiam shifted z1;
fill fullcircle scaled dotlabeldiam shifted z2;

% some arc arrows and labels
path a[];
z3 = (x1,5/12y1);
a1 = z3 {left} .. {left rotatedabout(z1,-t)} z3 rotatedabout(z1,-t);
drawarrow subpath (.05,.95) of a1;
label.llft(btex $\theta$ etex, point .5 of a1);

a2 = subpath (0,1) of reverse quartercircle scaled 2.2r shifted z1;
drawarrow a2 rotatedabout(z1,-100);
drawarrow a2 rotatedabout(z1,80);

% finally all the other labels
label.top(btex $r$ etex, .5[z1,z2]);
label.lft(btex $y$ etex, (0,y2));
% give all the x-axis labels a common baseline with mathstrut
label.bot(btex $\mathstrut x$ etex, (x2,0));
label.bot(btex $\mathstrut r\theta$ etex, (x1,0));
label.bot(btex $\mathstrut 2\pi r$ etex, (r*2pi,0));
% notice how nicely the coordinates work...
dotlabel.top(btex $(\pi r,2r)$ etex, (pi*r,2r));
% and a little alignment to finish
label(btex $\vcenter{\halign{&$#$\hfil\cr
x=r(\theta-\sin\theta)\cr
y=r(1-\cos\theta)\cr}}$ etex,(4.2r,r));

endfig;
end.
```

You can generalize the picture to make cycloids where the point tracing the cycloid is not on the circumference doing the rolling; the classic example is the wheel of the train with a flange. Here I have added R to define the radius of an outer rim, while the wheel still rolls along a circle of radius r. You might like to experiment with making $R < r$. Note also that variable names are case sensitive in METAPOST.




```
prologues := 3; outputtemplate := "%j%c.eps";
beginfig(1);
R = 1.8cm; % radius of the outer part
r = 1.3cm; % radius of the inner part
pi = 3.14159265; % constant

% define the cycloid
path c;
c = (0,-R) rotated 100 shifted (r*-100/180*pi,r)
    for t=-99 upto 460:
      -- (0,-R) rotated -t shifted (r*t/180*pi,r)
    endfor;

% axes
drawoptions(withcolor .5 white);
path xx, yy;
yy = (down -- 5 up) scaled 1/2 r;
xx = (xpart point 0 of c, 0) -- (xpart point infinity of c,0);
draw fullcircle scaled 1/4r; drawarrow xx; drawarrow yy;
drawoptions();
label.rt (btex $x$ etex, point 1 of xx);
label.top(btex $y$ etex, point 1 of yy);

% draw the cycloid on top of the axes
draw c withcolor .67 red;

% define a couple of related points:
% z1 center of the blue wheel
% z2 intersection of rim and cycloid
t = 124; % if you change t then the wheel will "roll" along...
z1 = (r*t/180*pi,r);
z2 = (0,-R) rotated -t shifted z1;
```

```
% draw the auxiliary lines
draw (0,y2) -- z2 -- (x2,0) dashed withdots scaled .6;
draw z2 -- z1 -- (x1,0);

% draw the rolling circle and mark the centre and intersection with cycloid
draw fullcircle scaled 2r           shifted z1 withcolor .67 blue;
draw fullcircle scaled 2R           shifted z1 withcolor .5[.67 blue,white];
fill fullcircle scaled dotlabeldiam shifted z1;
fill fullcircle scaled dotlabeldiam shifted z2;

% some arc arrows and labels
path a[];
z3 = (x1,5/12y1);
a1 = z3 {left} .. {left rotatedabout(z1,-t)} z3 rotatedabout(z1,-t);
drawarrow subpath (.05,.95) of a1;
label.llft(btex $\theta$ etex, point .5 of a1);

a2 = subpath (0,1) of reverse quartercircle scaled 2.2r shifted z1;
drawarrow a2 rotatedabout(z1,-100);

% finally all the other labels
label.rt (btex $r$ etex, (x1,.5y1));
label.urt(btex $R$ etex, .6[z1,z2]);
label.lft(btex $y$ etex, (0,y2));
% give all the x-axis labels a common baseline with mathstrut
label.bot(btex $\mathstrut x$ etex, (x2,0));
label.bot(btex $\mathstrut r\theta$ etex, (x1,0));
label.bot(btex $\mathstrut 2\pi r$ etex, (r*2pi,0));
draw (down--up) scaled 2 shifted (r*2pi,0) withcolor .5 white;
% notice how nicely the coordinates work...
dotlabel.top(btex $(\pi r,R+r)$ etex, (pi*r,R+r));
% and a little alignment to finish
label(btex $\vcenter{\halign{&$#$\hfil\cr
x=r\theta-R\sin\theta\cr
y=r-R\cos\theta\cr}}$ etex,(4.75r,r));

endfig;
end.
```

5 Mathematics, trigonometry and transformations

This section discusses plain METAPOST's concepts of numeric variables, pair variables, and path variables; and what you can do with them.

First **numerics**: METAPOST inherits it's 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 ones 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 are chosing fractional step-values in a `for` loop, where 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 upto 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 for filling or drawing.
- You can turn a number upto 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`

You get 11 iterations in the second but only 10 with the first.

5.1 Numeric constants

Plain METAPOST 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 that 150,000 dots per square inch. If you are designing fonts in METAFONT it's useful to help 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: $\mathbf{abs}(a-b) < eps$.
- *epsilon* is defined to be $\frac{1}{65536}$, the smallest postive scaled number.
- *infinity* is defined to be $4096 - epsilon$, 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 3.5.

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 $4096 = infinity + epsilon$ 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"
>> 0.00048999999999999998
>> 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.

Using the 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: ⟶

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 more accurate definitions. You can safely use these definitions with `scaled`, as they are equivalent to the decimals currently given in `plain.mp` (with the exception of `cc` which is 0.00003 smaller!).

5.3 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.

Unlike numerics, pair 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 one, you can equate it to a pair value using $=$ as normal, and overwrite it using the assignment operator $:=$.

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: '$A := (42, \mathbf{ypart}\ A)$;'.

In addition there is a neat macro defintion in plain METAPOST that allows you do deal with the x- and y-parts of pairs rather more succintly. The deceptivly 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`.

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

Plain METAPOST defines five useful pair variables: *origin*, *right*, *up*, *left*, and *down*. As so often, the Knuthian definitions in `plain.mp` are quite illuminating $\longrightarrow$

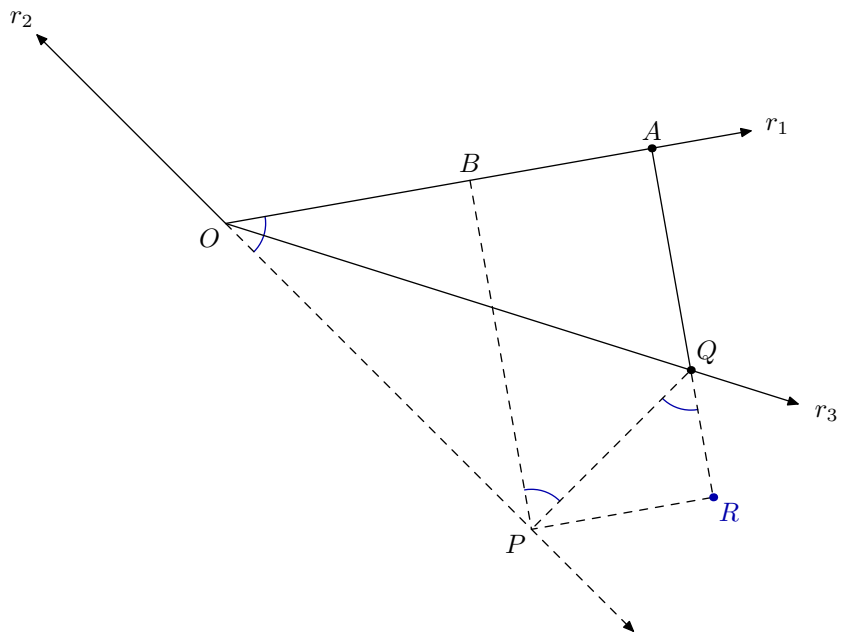
```
% pair constants
pair right,left,up,down,origin;
origin=(0,0); up=-down=(0,1); right=-left=(1,0);
```

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*;' applied to a path or an image works well.

Unfortunately this convenient notation does not work well with units of measure, 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 '$(144, 0)$'.

5.4 Coordinate geometry example

```
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);
draw fullcircle scaled 30 rotated angle (Q-P) shifted P cutafter (P--B);
draw fullcircle scaled 30 rotated angle (P-Q) shifted Q cutafter (Q--R);
draw fullcircle scaled 30 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
draw B--P--R--Q--P dashed evenly;
drawarrow origin -- P scaled 4/3 dashed evenly;
% label the points
dotlabel.urt(btex $Q$ etex, Q);
dotlabel.top(btex $A$ etex, A);
dotlabel.lrt(btex $R$ etex, R) withcolor .67 blue;
label.top (btex $B$ etex, B);
label.llft(btex $P$ etex, P);
label.llft(btex $O$ etex, origin);
endfig;
```



5.5 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 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` and `binary` number systems, but the `decimal` number system will give you an error if you supply more digits that the default precision (which is 34). So it's best to use no more that 34 digits. It's also possible, but not really worth the trouble, to define a routine to calculate π for you 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 $\tan 30^\circ = y/x$. 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 tangent function shown here does not check whether $x = 0$; if this is an issue you can say something like '**if** $x = 0$: *infinity* **else**: y/x **fi**' at the appropriate point.

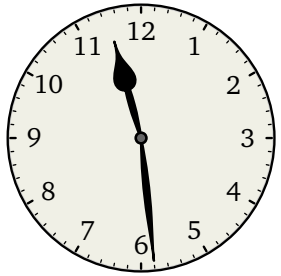
```
numeric pi;
% approximate value
pi := 3.14159;

% up to 34 digits of precision
pi := 3.14159265358979323846264338327950 3;

% as many digits as are needed...
vardef getpi =
  numeric 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
vardef degrees(expr theta) = 180 / pi * theta enddef;
vardef radians(expr theta) = pi / 180 * theta enddef;

% trig functions that expect radians
vardef sin(expr theta) = sind(degrees(theta)) enddef;
vardef cos(expr theta) = cosd(degrees(theta)) enddef;
% inverse trig functions
vardef acosd(expr a) = angle (a,1+-+a) enddef;
vardef asind(expr a) = angle (1+-+a,a) enddef;
vardef acos(expr a) = radians(acosd(a)) enddef;
vardef asin(expr a) = radians(asind(a)) enddef;

% tangents
vardef tand(expr theta) = save x,y; (x,y)=dir theta; y/x enddef;
vardef atand(expr a) = angle (1,a) enddef;
```

5.6 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 $\leq x$. You can use `x=floor x` to check that x is an integer.
- '**ceiling** x' returns $\lceil x \rceil$, the smallest integer $\geq x$.
- '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 (240/255, 240/255, 230/255);
  draw fullcircle scaled  99 withcolor .8 white;
  draw fullcircle scaled 100 withpen pencircle scaled 7/8;
  % numerals
  for h=1 upto 12:
    label( decimal h infont "bchr8r", (40,0) rotated htod(h));
  endfor
  % hour and minute marks
  for t=0 step  6 until 359: draw ((48,0)--(49,0)) rotated t; endfor
  drawoptions(withpen pencircle scaled 7/8);
  for t=0 step 30 until 359: draw ((47,0)--(49,0)) rotated t; endfor
  % hands rotated to the given time
  filldraw hand1 rotated htod(hours+minutes/60);
  filldraw hand2 rotated mtod(minutes);
  % draw the center on top
  fill fullcircle scaled 5;
  fill fullcircle scaled 3 withcolor .4 white;
) enddef;
```

6 Random numbers

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

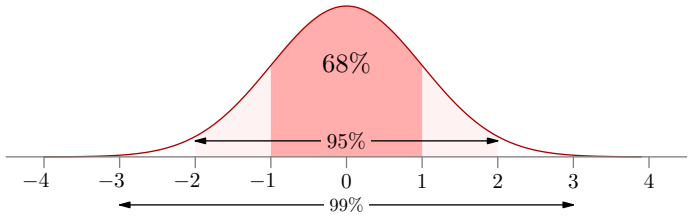
- '**uniformdeviate** n' generates a random real number between 0 and n.

  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'.

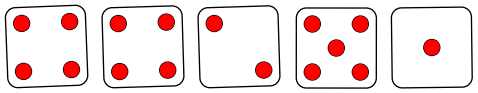
  If you use the new number systems, you should beware that the numbers generated will all be multiples of $\frac{1}{4096}$, so **uniformdeviate** 8092 (for example) will generate even integers instead of random real numbers. This 'feature' is an accident of the way that the original rather complicated arithmetic routines have been 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%; between −2 and 2, about 95%.



  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; picture p;
  d = (for i=0 upto 3:
         subpath (r+i,1+i-r) of unitsquare ..
       endfor cycle) scaled 30;
  p = image(draw fullcircle scaled 6;
            fill fullcircle scaled 6 withcolor pip_color);
  pair ul, ur, ll, lr;
  ul = 1/5[ulcorner d, lrcorner d];
  lr = 4/5[ulcorner d, lrcorner d];
  ur = 1/5[urcorner d, llcorner d];
  ll = 4/5[urcorner d, llcorner d];
  image(fill d withcolor background; 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,ur];
     draw p shifted 1/2[ll,lr];
  fi)
enddef;

beginfig(1);
for i=0 upto 4:
  draw dice(1+floor uniformdeviate 6, red)
       rotated (2 normaldeviate)
       shifted (36i,0);
endfor
endfig;
```



6.1 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 →

There are a couple points here. First, the inner loop around the assignment to u is designed to avoid very small values that would cause v/u to be larger than 64, and hence make `xa**2` 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 it's 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.

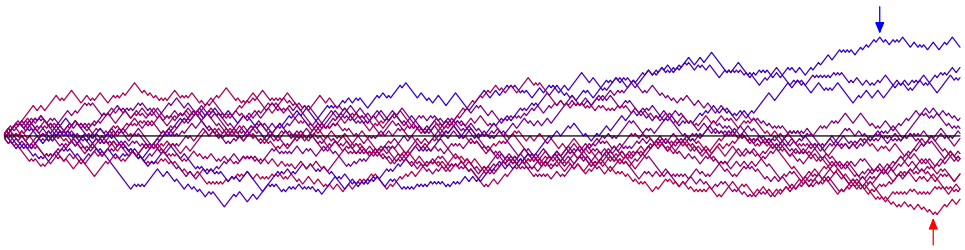
You can also see the special nature of METAPOST's **mexp** and **mlog** functions. They are defined so that $\mathbf{mexp}\, x = \exp(x/256)$ and $\mathbf{mlog}\, x = 256 \log(x)$. This is another artefact of the scaled number system. METAPOST computes x^y using the formula `mexp(y*mlog(x))`, and the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

At the start of each job, METAPOST automatically sets a new seed for the random number generator, so that the sequence of numbers is different each time. But you can set this yourself if you need the same sequence each time. At the start of your program you should put `randomseed:=3.14;` (or whatever value you prefer). According to *The Metafont Book*, the default value is *day*+ *time*∗ *epsilon*.

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

vardef exponentialdeviate =
  save u;
  forever:
    u := uniformdeviate 1;
    exitif (u>0);
  endfor
  -mlog(u)/256
enddef;

vardef gammadeviate(expr a,b) =
  save y, x, v, s, accept; boolean accept;
  s = sqrt(2a-1);
  forever:
    forever:
      y := tand(uniformdeviate 180);
      exitif y<64;
    endfor
    x := s * y + a - 1;
    accept := false;
    if x>0:
       v := uniformdeviate 1;
       if (v <= (1+y**2)*mexp((a-1)*mlog(x/(a-1))-(256*s*y))):
         accept := true;
       fi
    fi
    exitif accept;
  endfor
  x/b
 enddef;
```

6.2 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 sets `d` to -1 or $+1$. You can (of course) create different kinds of random walks, by changing the way you set this delta value, for example by removing the **floor** instruction, or scaling the value, or changing the odds in favour of one direction or the other. For example:

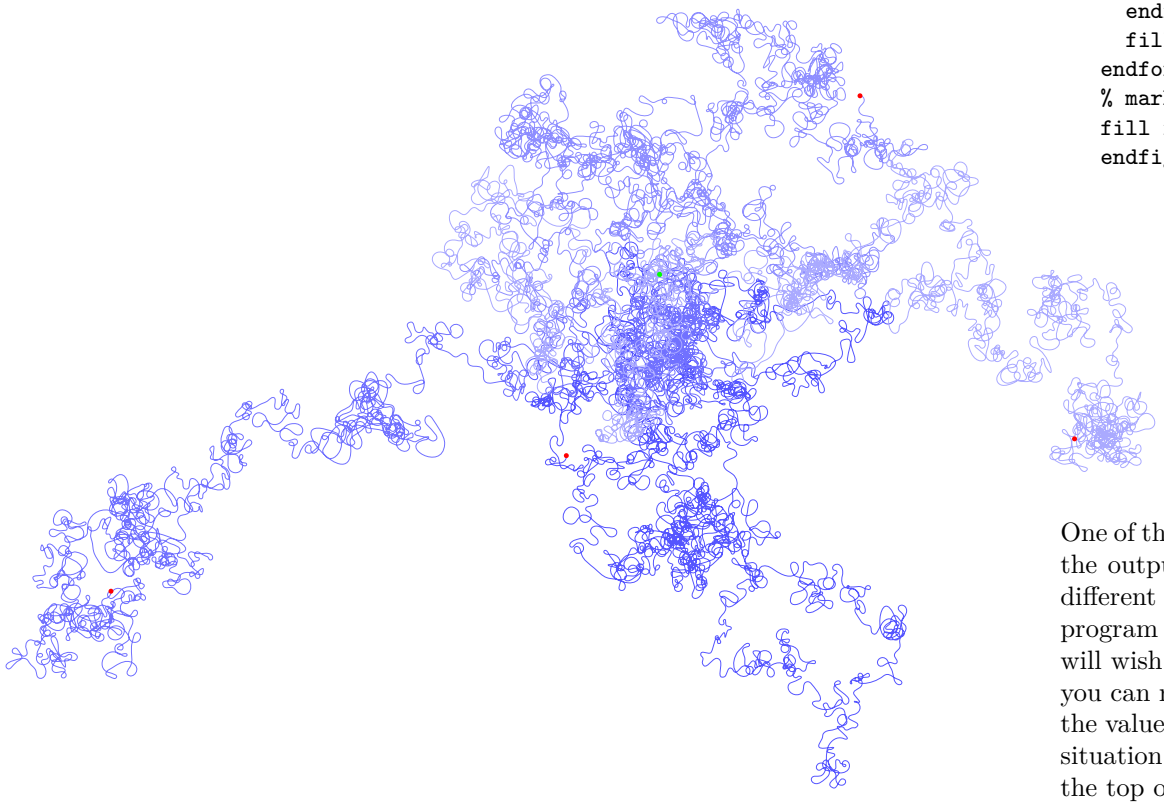
$$d \leftarrow \textbf{if } p > \textbf{uniformdeviate } 1 : +1 \textbf{ else} : -1 \textbf{ fi};$$

will set d positive with probability p and negative with probability $1 - p$.

```
beginfig(1);
w = 377; h = 233; n = 500;
pair zenith, nadir; zenith = nadir = origin;
path walk[];
for i=1 upto 16:
  y := 0;
  walk[i] = origin for x=0 step w/n until w:
     hide(
       d := floor uniformdeviate 2 * 2 - 1;
       y := y + d;
       if y > ypart zenith: zenith := (x,y) ; fi
       if y < ypart nadir:  nadir  := (x,y) ; fi
     )
     -- (x,y)
  endfor;
  draw walk[i] withcolor ((y+h/4)/(h/2))[red,blue];
  endfor

draw (origin--right) scaled w;
draw (origin--right) scaled w shifted (0,+h/4) withcolor .8 white;
draw (origin--right) scaled w shifted (0,-h/4) withcolor .8 white;

drawarrow (12 up   -- 2 up  ) shifted zenith withcolor blue;
drawarrow (12 down -- 2 down) shifted nadir  withcolor red;

endfig;
```

6.3 Brownian motion

A random walk is normally constrained to move one unit at a time, but if you relax that constraint and use '**normaldeviate**' in place of '**uniformdeviate**' you can get rather 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, as well as vary the colour each time you draw a new curve, then the result is almost artistic.



```
beginfig(2);
for n=1 upto 4:
  x:=y:=0;
  draw (x,y) for i=1 upto 2000:
    hide(x:=x+4normaldeviate; y:=y+4normaldeviate;)
    .. (x,y)
  endfor withcolor ((n+2)/9)[blue,white];
  fill fullcircle scaled 3 shifted(x,y) withcolor red;
endfor
% mark the origin
fill fullcircle scaled 3 withcolor green;
endfig;
```

One of the features of using these random number generators is 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 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.

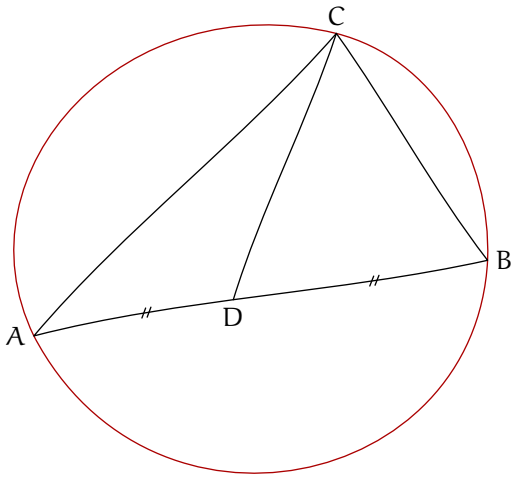
```
randomseed := uniformdeviate infinity;
```

Now you will find METAPOST writes the value it used in the log.

6.4 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.

6.4.1 Making curves and straight lines look hand drawn

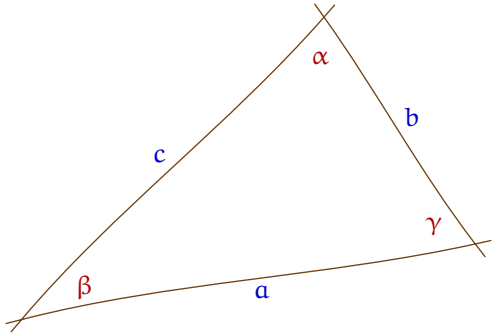


A small amount of random wiggle makes the drawing come out charmingly wonky. Notice that the `freehand_path` macro will transform a path whether it is straight or curved, and open or cyclic. Notice also that to find the mid-point of a line, you find the point along the freehand path; if you simply put `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 font complements the hand-drawn look; you might find that a little of this type of decoration goes a long way.

```
defaultfont := "eurm10";

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;

picture mark[];
mark1 = image(draw (left--right) scaled 2 rotated 60);
mark2 = image(draw mark1 shifted left; draw mark1 shifted right);

def moved_along expr x of p = rotated angle direction x of p
                              shifted point x of p enddef;

z0 = (0,-1cm); z1 = (6cm,0);  z2 = (4cm,3cm);
path t, c;
t = freehand_path(z0--z1--z2--cycle);
c = freehand_path(z0..z1..z2..cycle);

z3 = point 1/2 + 1/20 normaldeviate of subpath (0,1) of t;

beginfig(1);
draw c withcolor .67 red;
draw t;
draw freehand_segment(point 2 of t--z3);
draw mark2 moved_along 1/4 of t;
draw mark2 moved_along 3/4 of t;

label.lft("A", z0);
label.rt ("B", z1);
label.top("C", z2);
label.bot("D", z3);
endfig;
```

6.4.2 Extending straight lines slightly

This second freehand figure uses the same macros as the one on the previous page, but now the ink colour is set to sepia, and the lines are given a slightly more hand drawn look at the corners.



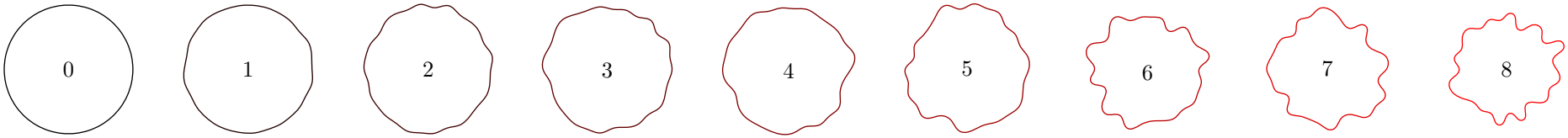
❖ The AMS Euler font available to METAPOST as `eurm10` is encoded as a subset of the TEX math italic layout — essentially it has all the Greek letters but none of the arrows.

	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	℘		

```
color sepia; sepia = (0.44, 0.26, 0.08);

def draw_out(expr p, o) =
  draw p;
  for i=1 upto length(p):
     draw (unitvector(direction i-eps of p) scaled +o
           -- origin --
           unitvector(direction i+eps of p) scaled -o)
           shifted point i of p;
  endfor
enddef;

def angle_label_pos(expr p, i, s) =
  ( unitvector(point i-1 of p-point i of p)
  + unitvector(point i+1 of p-point i of p)
  ) scaled s shifted point i of p
enddef;

beginfig(2);
  drawoptions(withcolor sepia);
  draw_out(t,6);
  drawoptions(withcolor .78 blue);
  label.lrt ("a", point 1/2 of t);
  label.urt ("b", point 3/2 of t);
  label.ulft("c", point 5/2 of t);

  drawoptions(withcolor .67 red);
  label(char 11, angle_label_pos(t,2,10));
  label(char 12, angle_label_pos(t,0,14));
  label(char 13, angle_label_pos(t,1,10));
endfig;
```

6.5 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. The basic path was a circle with radius `s` drawn by a loop with `n` steps like this:

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

The random noise is then added to the x-coordinate 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.

Note that since we assign to `shape` each time round the loop, we have to use `:=` to update the value instead of `=`.

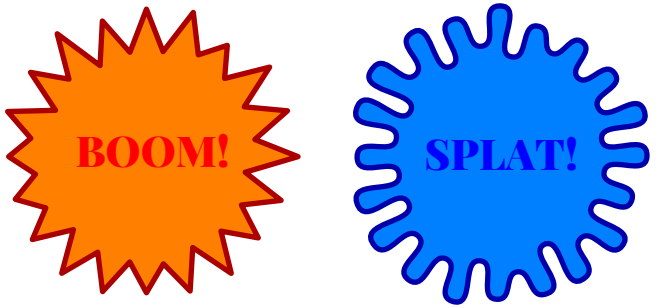
```
beginfig(1);
desired_length := 180; n := 30; s := 80;
path shape;
for r=0 upto 8:
  shape := for i=0 upto n-1:
             (s + r * normaldeviate, 0) rotated (360/n*i) ..
           endfor cycle;
  shape := shape scaled (desired_length/arclength shape);
  shape := shape shifted (r*s,0);
  draw shape withcolor (r/8)[black,red];
  label(decimal r, center shape);
endfor
endfig;
```

6.6 Explosions and splashes

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

```
string heavy_font;
heavy_font = "PlayfairDisplay-Black-osf-t1--base";

randomseed:=2128.5073;

beginfig(1);

n = 40; r = 10; s = 50;

path explosion, splash;
explosion = for i=1 upto n:
  (s if odd(i): - else: + fi r + uniformdeviate r,0) rotated (i*360/n) --
endfor cycle;

splash = for i=1 upto n:
  (s if odd(i): - else: + fi r + uniformdeviate r,0) rotated (i*360/n) ..
endfor cycle;
splash := splash shifted (3s,0);

fill explosion withcolor 1/2 green + red;
draw explosion withpen pencircle scaled 2 withcolor 2/3 red;
label("BOOM!" infont heavy_font scaled 2, center explosion) withcolor red;

fill splash withcolor 1/2 green + blue;
draw splash withpen pencircle scaled 2 withcolor 2/3 blue;
label("SPLAT!" infont heavy_font scaled 2, center splash) withcolor blue;

endfig;
```

In this figure `n` is the number of points in the shape, `r` is the amount of randomness, and `:s` is the radius used. 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.

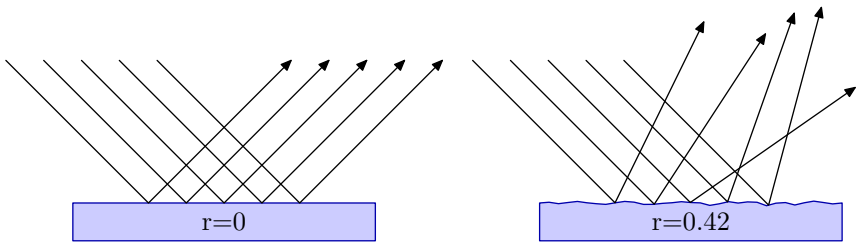


The display font used here is one of the gems hidden away in `psfonts.map`. If you run METAPOST with the `-recorder` option, it will create a list of all the files used, with the current job name and an extension of `.fls`. This file will include a line which tells you exactly which version of `psfonts.map` 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. However beware that just because a name is defined in your map file, does not necessarily mean that you actually have the required PostScript font files installed as well. But if you have a full TexLive installation you will find that very many of them are already installed.

6.7 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.

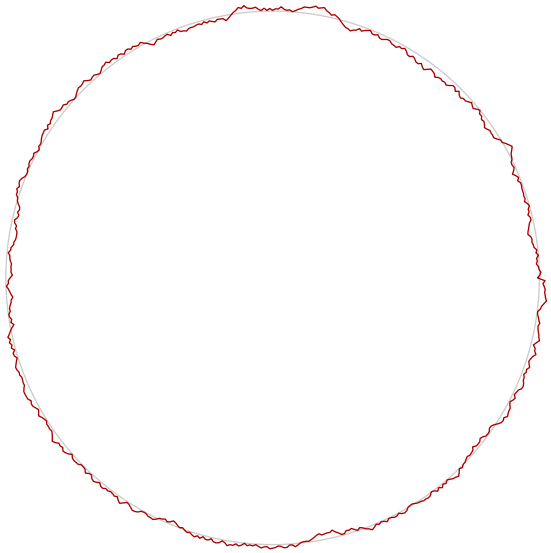
```
def perpendicular expr t of p =
  direction t of p rotated 90 shifted point t of p
  enddef;

beginfig(1);
u = 5mm;  r = 0.42; n = 32; s = 8u; theta = -45;

path base;
base = origin
       for i=1 upto n-1: -- (i/n*s,r*normaldeviate) endfor
       -- (s,0) -- (s,-u) -- (0,-u) -- cycle;
fill base withcolor .8[blue,white];
draw base withcolor .67 blue;

path ray[];
for i=2 upto 6:
  ray[i] = (left--right) scaled 2/3 s rotated theta shifted (i*u,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);
endfig;
```

6.7.1 Walking along a torn edge

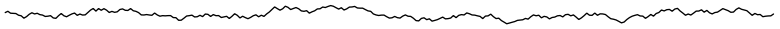
It's also possible to use a random walk approach so that each random step takes account of the previous one to avoid any big jumps. Here's one way to do that.

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

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. 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.



```
beginfig(1);
  y=0;
  path e;
  e = (0,y) for i=1 upto 288: -- (i,walkr y) endfor ;
  draw e;
endfig;
```



```
beginfig(2);
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;
end.
```

7 Labels and annotations

- dealing with text - plain, psfonts, stacks, latex, OTF fonts,
  - dynamic labels, rotating labels
  - annotations - overbrace, underbrace, length arrows etc, ahangle trick
  - paragraphs of text

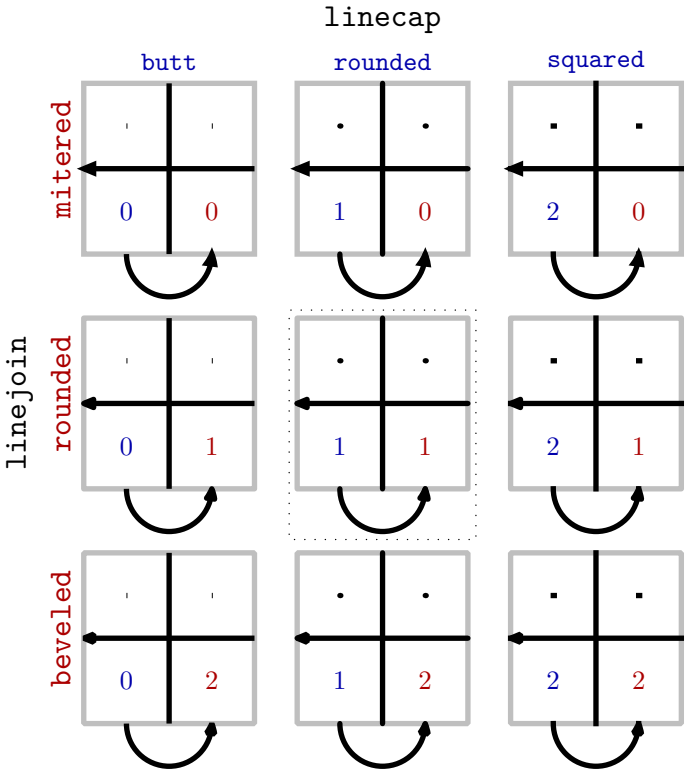
8 Line caps and line joins

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 affect 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** produces 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.
- METAPOST's arrow head routines don't work very well when *linecap* = *squared*; 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 of the grey box 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.
- The default mode, with rounded caps and rounded joins, looks rather sharper with the normal pen (`pencircle scaled 1/2`). But the dots drawn with the default pen are easy to miss. This drawing was done with `pencircle scaled 2`.

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 is correct for nearly all drawings. If you set `miterlimit:=0;` then the mitered line join mode becomes more or less the same as the beveled mode.



9 To do...

filling shapes gradients - faking transparency, overlaying colours, clipping to shapes, filling with patterns - making waves

- paths, points, directions, subpaths, reversed paths
- boxen, fitting, corners and centres, cutbefore cutafter
- inline if and for and range, loops exitif upto downto
- dots, dotlabeldiam, coloured dots, hollow dots
- graphs, axes, grids, number labels, jitter, sketch graphs
- geometry - constructions, perpendiculars, bisection, tangents, segments and lines, incircle, circumcircle
- angle marks, including curved angle marks
- intersections, buildcycle, eggs, ellipses
- drawing knots, double lines, ropes
- HSV colours, brewer colours, rainbows, 50% white etc
- decorating lines, Meccano
- finding supremum
- decorated tables
- numberlines
- a pulse
- reuleaux polygons
- the eye
- feynman diagrams the easy way
- physics diagrams, light rays, pendulum, indicating movement and vibration
- parametric equations, folium of Descartes
- examining a glyph
- tesellations and tiling (reference title page)
- all sorts of arrow, arrows between arrows, arrows next to a path (handles)
- faking 3d
- recursive drawings, trees
- four box model charts - Tufte charts - venn diagrams
- my workflow - tex process flow diagrams
- triangle of polygons

10 A tour of the plain format

Contents

1 Start here **1**

2 Some features of the syntax **2**

3 Making and using closed paths **3**
- 3.1 Points on the standard closed paths . . . 4
- 3.2 Building cycles from parts of other paths . . . 5
- 3.3 The implementation of `buildcycle` . . . 6
- 3.4 Strange behaviour of `buildcycle` with two cyclic paths . . . 7
- 3.5 Find the overlap of two cyclic paths . . . 8

4 Cycloid and other spiral graphs **9**

5 Mathematics, trigonometry and transformations **13**
- 5.1 Numeric constants . . . 14
- 5.2 Units of measure . . . 15
- 5.3 Pairs and coordinates . . . 16
- 5.4 Coordinate geometry example . . . 17
- 5.5 Trigonometry functions . . . 18
- 5.6 Integer arithmetic, clocks, and rounding . . . 19

6 Random numbers **20**
- 6.1 Random numbers from other distributions . . . 21
- 6.2 Random walks . . . 22
- 6.3 Brownian motion . . . 23
- 6.4 Drawing freehand . . . 24
  - 6.4.1 Making curves and straight lines look hand drawn . . . 24
  - 6.4.2 Extending straight lines slightly . . . 25
- 6.5 Increasingly random shapes of the same size . . . 26
- 6.6 Explosions and splashes . . . 27
- 6.7 Simulating jagged edges or rough surfaces . . . 28
  - 6.7.1 Walking along a torn edge . . . 29

7 Labels and annotations **30**

8 Line caps and line joins **31**

9 To do... **32**

10 A tour of the plain format **33**