Metapost paths and pairs

(and pens and transforms)

Taco Hoekwater

September 21, 2021

☐ We will discuss creating paths first
☐ Followed by creating pairs
☐ Then creating pens

After that, we will discuss the operations on those items, for instance by using transformations

Defining a path

Simple case:

```
path p;
p = (0,0)..(100,100);
```

Defining a path

From a path (sub)expression:

```
path p;
p = ((0,0)..(100,100));
```

Defining a path

From a set of points:

```
pair startp, endp;
startp = (0,0);
endp = (100,100);
path p;
p = startp..endp;
```

Defining a path

From a single point:

```
pair startp;
startp = (0,0);
path p;
p = startp;
```

Defining a path

From another path:

```
path p, q;
q = (0,0)..(100,100);
p = q;
```

Defining a path

From the reverse of another path:

```
path p, q;
q = (0,0)..(100,100);
p = reverse q;
```

Defining a path

From a part of another path:

```
path p, q;
q = (0,0)..(100,100);
p = subpath (0.25,0.5) of q;
```

Defining a path

From a pen:

```
path p;
p = makepath pencircle;
```

Defining a path

From a drawn path:

```
path p,q;
q = (0,0)..(100,100);
p= envelope pencircle of q;
```

Defining a path

- ☐ For `envelope`, the pen needs to be polygonal.
- ☐ To get the ‘other’ side of a cyclic path, reverse the path

Defining a path

Or any combination of all of those:

```
path p, q;
pair endp;
endp = (100,100);
q = (0,0)..(100,100);
p = (0,0) .. reverse (subpath (0.25,0.5) of q)
    .. makepath pencircle .. endp;
```

Path directions and connectors

Simple case of direction:

```
path p;
p = (0,0)..(100,100);
```

Path directions and connectors

With pairs as directions:

```
path p;
pair up,right;
up = (0,1);
right = (1,0);
p = (0,0){up}..{right}(100,100);
```

Path directions and connectors

With explicit direction vectors:

```
path p;
p = (0,0){0,1}..{1,0}(100,100);
```

Path directions and connectors

With `curl`s as directions:

```
path p;
p = (0,0){curl 0}..{curl 1}(100,100);
```

A `curl` specification is a number from 0 to infinity.
What it does:

- ☐ it sets the amount of curliness at that point
- ☐ if the requested amount of curl is high, it will adjust the curliness at adjacent points as well
- ☐ its assumed default value at ending points is `1`
- ☐ an explicit `curl` makes that point an ‘endpoint’ (a.k.a. a corner)

Path directions and connectors

The last item on the previous slide is why this definition works:

```
def -- = {curl 1}..{curl 1} enddef;
```

`curl` demonstration now (does not fit on slide)

Path directions and connectors

Handy to know:

- ☐ explicit vectors are expressions, so you can do calculations
- ☐ explicit incoming or outgoing curl and direction specifications migrate to the implicit side as well

Simple case of connector:

```
path p;
p = (0,0)..(100,100);
```

Path directions and connectors

Using tension as connectors:

```
path p;
p = (0,0)..tension 2 ..(100,100);
```

Path directions and connectors

Using two tensions as connectors:

```
path p;
p = (0,0)..tension 2 and 1 ..(100,100);
```

Path directions and connectors

A *`tension`* specification is a number from 0.75 to infinity. What it does:

- ☐ it controls the amount of ‘wideness’ of the curve segment
- ☐ its default value is `1`
- ☐ you can force a lower boundary with the **`atleast`** keyword

Path directions and connectors

Interesting predefined macros:

```
def --- = .. tension infinity .. enddef;
def ... = .. tension atleast 1 .. enddef;
```

Path directions and connectors

`tension` demonstration now (does not fit on slide)

Path directions and connectors

Using a control point as connector:

```
path p;
p = (0,0)..controls (70,70) ..(100,100);
```

Path directions and connectors

Using two control points as connector:

```
path p;
p = (0,0)..controls (20,70) and (80,100)..(100,100);
```

Path directions and connectors

Handy to know:

- ☐ processed path segments always use control points
- ☐ while METAPOST works out which control points to use, `curl` adjusts the vector angles and `tension` the vector length
- ☐ using explicit control points will therefore overwrite any `curl` specification for the segment

Path directions and connectors

Using concatenation as connector:

```
path p;
p = (0,0)..(50,50) & (50,50)..(100,100);
```

This only works if the left and right points are identical, and it is equivalent to

```
path p;
p = (0,0)..{curl 1}(50,50)..(100,100);
```

Path directions and connectors

Creating a cyclic path:

```
path p;
p = (0,0)..(100,100) .. cycle;
```

The `cycle` is just a reference back to the first point of the path being created

Btw, because of the default `curl 1`, this produces a somewhat circular-looking path

Path directions and connectors

So now you have a path. Here is some ‘handy to know’:

☐ the *`length`* of a path is the number of explicit points, minus 1

☐ the *`subpath`* operator adds points at the beginning and end of the subpath if needed

☐ ‘empty’ curve segments still count, so

```
p = (0,0)..(0,0)..(100,100);
```

defines a path of length 2

Defining a pair

Simple case:

```
pair a;
a = (0,0);
```

Defining a pair

From another pair:

```
pair a,b;
b = (0,0);
a = b;
```

Defining a pair

From an expression:

```
pair a;
a = ((0,0) + (100,100));
```

Defining a pair

From a path point:

```
path p;
pair a;
p = (0,0)..(100,100);
a = point 0.5 of p;
```

Defining a pair

From a path control point:

```
path p;
pair a;
p = (0,0)..(100,100);
a = precontrol 0.5 of p;
```

There is also `postcontrol`

Defining a pair

From a pen offset:

```
path p;
pair a;
p = (0,0)..(100,100);
a = penoffset (1,0.5) of pencircle;
```

Defining a pair

The `penoffset` returns the point along the pen in which the pen travels in the direction given by the offset argument.
For angular pens, different directions may return the same result because corner points are considered to have all directions between incoming and outgoing

Defining a pair

From a path intersection:

```
path p,q;
pair a;
p = (0,0)..(100,100);
q = (0,100)..(100,0);
a = p intersectiontimes q;
```

Defining a pair

The `intersectiontimes` returns two *time* values along the paths, encoded as a pair. If there are no intersections it returns `(-1,-1)`

If there are multiple intersections, it normally returns the first of those along the left-side path

However, if there are multiple intersections within a single curve segment (i.o.w 'between' knots), it will return the 'smallest' combination of times along both paths. (demo)

Defining a pen

For pens, there are a lot less options

Defining a pen

Simple case:

```
pen mypen;
mypen = pencircle;
```

`pencircle` is a built-in pen.

Defining a pen

Clearing a pen:

```
pen mypen;
mypen = nullpen;
```

`nullpen` is a built-in ‘pen’.

Defining a pen

From a path:

```
pen mypen;
path p;
p = (0,0)--(100,100)--(200,0)--cycle;
mypen = makepen p;
```

Defining a pen

Handy to know:

- ☐ `makepen` always converts `..` to `--`
- ☐ `pen`s are always convex; `makepen` will silently enforce this by ignoring concaveness-inducing points
- ☐ elliptical pens are created by transforming `pencircle`

Whenever you use a `path`, `pair` or `pen` in a METAPOST expression, you are allowed to transform it.

The following transformation options apply to all those object types (as well as `picture`s and `transform`s)

I'll use pairs as examples to keep it simple

transformations

```
pair a;
a = (100,100) rotated 90;
```

`rotated` works counter-clockwise around `(0,0)`

transformations

```
pair a;
a = (100,100) scaled 2;
```

transformations

```
pair a;
a = (100,100) shifted (50,50);
```

```
pair a;
a = (100,100) slanted 10;
```

transformations

```
pair a;
transform t;
t := identity scaled 5;
a = (100,100) transformed t;
```

The transform **`identity`** is not actually a primitive, but it is defined a curious way in the plain METAPOST macros:

```
transform identity;
for z=origin,right,up:
  z transformed identity = z;
endfor
```

The three equations in the *`for`* loop together resolve all six parts of the *`transform`* object

transformations

```
pair a;
a = (100,100) xscaled 5;
```

transformations

```
pair a;
a = (100,100) yscaled 2;
```

transformations

```
pair a;
a = (100,10) zscaled (5,2);
```

`zscaled` mimics complex number multiplication

`(100,10) zscaled (5,2)` becomes
`(5*100 - 2*10, 2*100 + 5*10)` = `(480,250)`

visually, `zscaled (a,b)` rotates and scales so that
`(1,0)` becomes `(a,b)`

Handy to know:

- ☐ the right hand sides are `numeric`, `pair`, and `transform` primaries
- ☐ you can chain transformers, they are processed left to right
- ☐ there is no direct assignment syntax for `transform` type definitions
- ☐ do not forget to add grouping if you are mixing `pair` and `path` in the same expression

Now let's look at with other operations you can do on *`path`*s, *`pair`*s, *`pen`*s and **`transforms`**.

other operations on paths

Find the length of a path:

```
path p;
p = (0,0)..(100,100);
d = length p;
```

This returns the number of segments (one less than the number of control points)

Find the drawn length of a path:

```
path p;
p = (0,0)..(100,100);
d = arclength p;
```

This returns the length of the actual curve(s).

Find the drawn time of a path:

```
path p;
p = (0,0)..(100,100);
d = arctime 100 of p;
```

This returns the time along the path at which the *arclength* is the specified value

other operations on paths

Test if a variable is a path:

```
path p;
p = (0,0)..(100,100);
if path p: ... fi
```

Single pairs fail this test, even though they are valid as path declarations

other operations on paths

Test if a variable is a cyclic path:

```
path p;
p = (0,0)..(100,100);
if cycle p: ... fi
```

Only paths created with `cycle` are considered cyclic

Find the time at which a path moves in a certain direction:

```
path p;
p = (0,0)..(100,100)..(200,100);
d = directiontime (1,0) of p;
```

other operations on paths

☐ the `pair` argument is treated as a direction vector

☐ if the path never travels in that direction, the return value is `-1`

☐ if the path travels multiple times in that direction, the first time is returned

☐ corners have all directions between incoming and outgoing angles

Find a bounding box point:

```
path p;
pair a;
p = (0,0)..(100,100)..(200,100);
a = ulcorner p;
```

Also defined are `llcorner`, `lrcorner`, and `urcorner`

other operations on pairs

Test if a variable is a pair:

```
pair a;
a = (100,100);
if pair a: ... fi
```

other operations on pairs

Get the x part:

```
pair a;
a = (10,10);
d = xpart a;
```

other operations on pairs

Get the y part:

```
pair a;
a = (10,10);
d = ypart a;
```

other operations on pairs

multiply or divide by a numeric:

```
pair a;
a = (100,100) * 5;
```

other operations on pairs

add or subtract another pair:

```
pair a,b;
b = (10,10);
a = (100,100) + b;
```

other operations on pairs

negation:

```
pair a;
a = -(100,100);
```

other operations on pairs

compare to another pair:

```
pair a,b;
b = (10,10);
a = (100,100);
if a > b: ... fi
```

Comparison of pairs initially compares the `xpart` value.
If those are equal, next it checks the `ypart`.

other operations on pairs

mediate between two pairs:

```
pair a,b,c;
b = (10,10);
a = (100,100);
c = 0.5[a,b];
```

other operations on pairs

For mediation with negative values, keep in mind that unary minus binds less forcefully than mediation:

```
c = -1[a,b];
```

is `(-10,-10)` because the mediation is processed first, whereas

```
c = (-1)[a,b];
```

is `(190,190)` because `a - (b - a)` = 2`a - b`.

other operations on pairs

Find the angle:

```
pair a;
a = (10,10);
d = angle a;
```

other operations on pens

Test if a variable is a pen:

```
if pen pencircle: ... fi
```

other operations on pens

Find a bounding box point:

```
pair a;
a = ulcorner pencircle;
```

Also defined are `llcorner`, `lrcorner`, and `urcorner`

other operations on transforms

Test if a variable is a transform:

```
if transform identity: ... fi
```

Get the x shift part:

```
d = xpart identity;
```

Get the y shift part:

```
d = ypart identity;
```

other operations on transforms

Get the x scale part:

```
d = xxpart identity;
```

other operations on transforms

Get the xy multiplier part:

```
d = xypart identity;
```

other operations on transforms

Get the yx multiplier part:

```
d = yxpart identity;
```

other operations on transforms

Get the y scale part:

```
d = yypart identity;
```

other operations on transforms

Compare to another transform:

```
transform T,V;
T = identity;
V = T scaled 2;
if T<V: ... fi
```

Comparison of transforms tests *xpart*, *ypart*, *xxpart*, *xypart*, *yxpart*, *yypart* consecutively.

These are the primitive operations.

Of course macro packages tend to define more:

☐ operators
☐ functions
☐ predefined variables
☐ et cetera.

Using `picture`s and the various other drawing primitives will be the topic of next year's talk

That's all!

(this slide only exists so I have exactly 1 slide per minute)