

Practical introduction to METAPOST*

Clément Hurlin[†]

December 10, 2007

Abstract

This document aims at providing you an introduction to METAPOST. METAPOST is a programming language dedicated to the generation of high-quality graphics. Produced images are vectorials, which means they can be rendered with great quality, no matter of the level of zoom. Plus, the different constructs inherited from classical programming languages allow the user to generate a large number of geometrical figures in an elegant and concise way.

You're invited to look at the sources of this document to see how figures included have been generated. To see all features of METAPOST in details, you can take a look at [1].

1 Compilation

A metapost file (extension .mp) has the following form:

```
beginfig(1)
...
endfig;
end.
```

Compiling this file can be done by typing:

```
mpost test.mp
```

This is going to generate a file named `test.1`¹ which contains the image generated in eps format. You can visualize this file with `ghostview` or you can include it in a L^AT_EX document such as this one:

```
\documentclass{article}
\usepackage{graphics}
\begin{document}

\includegraphics{test.1}

\end{document}
```

The `graphics` package is mandatory to include eps files with the `includegraphics` command. Once the dvi file has been obtained, you can generate a ps file with `dvips` or a pdf file thanks to `dvipdf`.

If you wish to generate ps with `latex` and pdf directly with `pdflatex`, insert the following piece of code in your preamble²:

```
\usepackage{ifpdf}
\ifpdf
\DeclareGraphicsRule{*}{mps}{*}{}
\fi
```

In the following, all figures included have been generated with this simple set of commands.

METAPOST is included in the modern distribution of L^AT_EX, thus if you have L^AT_EX installed, you have METAPOST as well.

2 Drawing

2.1 Points

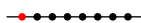
METAPOST features multiple types, the simplest one is the point (type `pair`). Let's begin by an example:

²For further information about this code see [2, 3]

*version of the document: 0.17

[†]clement.hurlin@sophia.inria.fr

¹test.1 because of the `beginfig(1) ... endfig` bloc



```

beginfig(1)
  pair p;
  pair q;

  p := (0, 0);
  q := (2cm, 0);

  draw p;
  draw q;
endfig;
end.

```

This code produces:

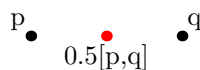


The `:=` assigns its coordinates to a point and you must note that it is different from `=`³. Note also that the syntax `pair p := (0, 0);` isn't allowed: you have to separate the assignation from declaration.

The `draw` command draws elements given as parameters. Multiple operations on points are predefined (see section 8.3 of [1]), as for example:

$$\begin{aligned}
 (x, y) \text{ shifted } (a, b) &= (x + a, y + b) \\
 (x, y) \text{ rotated } \theta &= (x \cos\theta - y \sin\theta, \\
 &\quad x \sin\theta + y \cos\theta)
 \end{aligned}$$

In the previous example, `q` could have been obtained with `q := p shifted (2cm, 0);`. It's also possible to compute a precise point of a segment. For example, the middle of the segment `[p, q]` could be obtained by `pair r; r := 0/5[p,q];`, as shown in red in the next figure:



The `xpart` function (respectively `ypart`) returns the x coordinate (respectively the y one) of a point.

³`=` is different from `:=` because it tries to solve the mathematical equation "left = right". It means that, from `a + b = 1;` and `a - b = 0;` `METAPOST` deduces `a = 0.5` and `b = 0.5` (see chapter 4 of [1] for further details). All examples of this document use `:=` because it seems more natural for the author however using `=` would not have something.

Note that, in `METAPOST`, the variables `z`, `x` and `y` are predefined: for all `i`, `z[i]` is a point and `x[i]` (respectively `y[i]`) is equal to `xpart z[i]` (respectively `ypart z[i]`).

2.2 Paths

From 2 points, you can define the `path` between them. To connect two points by a line you have to use the `--` operator `,` to connect them by a curve, you have to use the `..` operator.

```

beginfig(1)
  pair p [];

  p[1] := (0, 0);
  p[2] := p[1] shifted (1cm, 0);
  p[3] := p[2] shifted (0, 1cm);

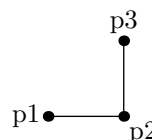
  draw p[1]; draw p[2]; draw p[3];

  path pt;
  pt := p[1] -- p[2] -- p[3]; % segments

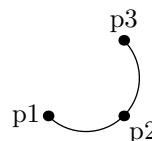
  draw pt;
endfig;
end.

```

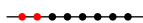
This piece of code produces:



By changing `pt := p[1] -- p[2] -- p[3];` to `pt := p[1] .. p[2] .. p[3];`, one obtains:

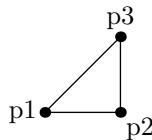


The curve generated can be modified by multiple parameters, see section 3 of [1]. You may also have noted that `draw` can be applied to paths as well (and as we will see later, to figures too).

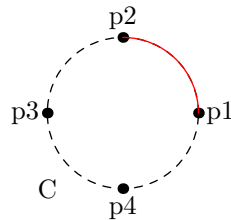


This example also demonstrates how you can declare arrays with []. It will be very useful when using loops.

To obtain a closed path, you can use **cycle** which connects the last point given to the first one. For example, to obtain a triangle from p[1], p[2] and p[3], you can write: **draw p[1] -- p[2] -- p[3] -- cycle;**, which generates:



Two operators on paths are very useful: **cutbefore** and **cutafter**. If C is a path and p a point, C **cutbefore** p gives back the path C where the part before p has been deleted. Example:



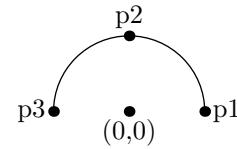
This picture has been generated with:

```
r := 1cm; % résumé du code
p[1] := (r, 0);
C := fullcircle scaled 2r;
draw C; % dessiné en pointillés

for i:=2 upto 4:
  p[i] := p[i-1] rotated 90;
endfor

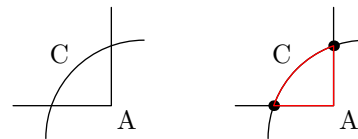
% dessiné en rouge
draw C cutbefore p[1] cutafter p[2];
```

This piece of code contains **fullcircle** which gives a circle centered on the origin and whose diameter is 1. The **halfcircle** and **quartercircle** commands are enough explicit too. For example, in the previous picture **draw halfcircle scaled 2r;** generate:

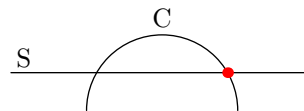


In order to generate a closed path, you can use **buildcycle**. If it is given as parameters n paths p1, ..., pn, it tries to build a cycle which follows p1 then ... then pn. If multiple results are possible, its behavior is hard to predict, but it happens rarely (see section 8.1 of [1]).

For example, given the paths A and C on the left picture, you can obtain the red path of the right picture by using **buildcycle (C,A);**



The binary operator **intersectionpoint** takes two paths in parameters and returns one of their intersection points (if there exists one, otherwise it fails). For example, here, **draw C intersectionpoint S;** gives:



2.3 Cycles (filling)

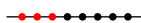
Given a closed path, you can **fill** it with the current color. For example, the following piece of code:

```
draw carre dashed evenly; %resume du code
fill coeur withcolor red;
```

produces:



But, **fill** erases what is present before, that's why:



```
fill coeur withcolor red; %resume du code
fill carre;
```

produces:



The **unfill** command erases an area:

```
drawoptions(withcolor red); %resume
fill carre;
drawoptions();
unfill coeur;
```

produces:



3 Colors, text and drawing options

You may have noticed that in the last examples, commands dedicated to drawing and colors have appeared. To deal with colors, METAPOST provides the **color** type. Basically, a color behaves like a point, except that it is a triplet (following the Red Green Blue convention).

A few colors are predefined: **black**, **white**, **red**, **green** and **blue**. For example, **black** corresponds to (0, 0, 0). To define new colors, you can subtract and add a color to a color or multiply a color by a constant.

For example, the following piece of code:

```
beginfig(1)
color c[];
c[0] := (0.4, 0.4, 0.4);
c[1] := 2*c[0];
c[2] := 0.5[white, red];
```

```
numeric i, j;
i := 2cm; j := 1cm;

draw (0,i) -- (5cm, i) withcolor c[0];
draw (0,j) -- (5cm, j) withcolor c[1];
draw (0,0) -- (5cm, 0) withcolor c[2];
endfig;
end.
```

produces:



As you can see in this example, instruction about colors can be given with **withcolor** when **drawing**. If you wish to use the same options for multiple **draw** commands you can give them to **drawoptions** (you can see it on the last figure of the previous section).

Another interesting type is **pen**. In fact, all drawing operations are done modulo the current pen. Let's see that with an example:

```
beginfig(1)
pen big;
big = pencircle scaled 4bp;

draw (0,1cm) -- (5cm, 1cm);
draw (0,0cm) -- (5cm, 0) withpen big;
endfig
end.
```

which produces:



The shape of the pen can be changed as well:

```
beginfig(1)
pickup(pencircle scaled 10bp);
draw (0,0);

pickup(pensquare scaled 10bp);
draw (5cm,0);
endfig;
end.
```

produces:



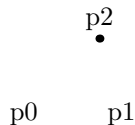
As you can see in this last example, the **pickup** command changes the current pen. The **makepen** operator allows you to build pens from scratch, see [1] for further information.

To insert some text, the **label** command is available. It takes as arguments the **string** to write and the position where to write. For example, the following piece of code:

```
beginfig(1)
  pair p[];
  p[0] := (0,0); p[1] := (1cm, 0cm);
  p[2] := (1cm, 1cm);

  label("p0", p[0]);
  label.rt("p1", p[1]);
  dotlabel.top("p2", p[2]);
endfig;
end.
```

produces:



We see in this example that **label** and **dotlabel** can take an additional parameter after a “.” and before the classical parameters. These arguments (**rt** and **top** in our example) allow to shift where the string is going to written. In fact, the command **label**(“p0”, p[0]); writes “p0” exactly on p[0], which is not very convenient.

The additional argument can be: **lft** (left), **rt** (right), **top**, **bot** (bottom), **ulft** (up left), **urt** (up right), **llft** (low left), **lrt** (low right).

Finally, the string given to the different labelling commands can be written in **TeX** if they are surrounded by **btex** and **etex**. It allows you to display mathematical formulae directly in **METAPOST**.

For example, the following piece of code:

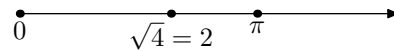
```
beginfig(1)
```

```
drawarrow (0,0) — (5cm,0);
dotlabel.bot("0", (0,0));

dotlabel.bot(btex $\sqrt{4} = 2$ etex,
             (2cm, 0));
dotlabel.bot(btex $\pi$ etex,
             (3.14cm, 0));

endfig;
end.
```

produces:



Note the use of **drawarrow** (see section 8.5 of [1] for variants).

4 Loops, conditionals

We finally come to features that make **METAPOST** efficient: loops and conditionals, inherited from classical programming languages. The construction:

```
for <variable> = <list> :           % résumé
  <body>;
endfor
```

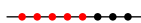
is the generic **for** loop of **METAPOST**. The **<list>** can be defined in different ways:

- **i upto j**
- **i downto j**
- **i step k until j**

For example, the following piece of code:

```
for i:=1 upto 3:
  show i;
endfor
end.
```

outputs “1 2 3” on the standard output. **5 downto 2** produces “5 4 3 2” and **1 step 2 until 7** is similar to **upto** excepts that, at each iteration, the variable is incremented by 2, thus it produces the list “1 3 5 7”.



It is also possible to explicitly give the list of values for the variable of the loop. For example, the following piece of code:

```
for i:=1, 3, -2, "tutut":
  show i;
endfor
end.
```

produces “1 3 -2 “tutut” ”.

Other commands are available, in particular **forever**:

```
forever:                                     % résumé
  <body>
endfor
```

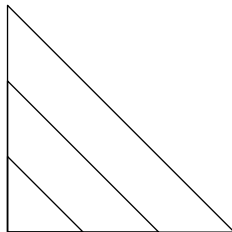
which executes the <body> forever. Two structures permit to exit from the loop: **exitif** <condition> and **exitunless** <condition>.

Another variants of **for** loops are available, have a look at section 10 of [1].

Finally, a graphical example, the following piece of code:

```
beginfig(1)
  for r:=1cm step 1cm until 3cm :
    draw (0,0) -- (0, r)
           -- (r, 0)-- cycle;
  endfor
endfig;
end.
```

which produces:



And the conditional structure, quite trivial:

```
boolean b;
boolean c;
```

```
b := true;
c := false;

if b = c:
  show "vrai";
else:
  show "faux";
fi

end.
```

That outputs “faux” on the terminal (see section 9.2 of [1] for variations).

5 Pictures and transformation

METAPOST can manipulate whole figures (type **picture**), by storing it into variables, in order to reuse them (drawing, transformation etc.) The **currentpicture** variable can be used to achieve that.

In fact, **draw** commands write to **currentpicture**, which means that you can save a picture in a **beginfig ... endfig**, and later, add elements to this figure in another bloc. For example, the following piece of code, in a file named *picture.mp*:

```
picture pic;
path c[];
c[0] = fullcircle scaled 2cm;
c[1] = fullcircle scaled 1cm;

beginfig(1)
  fill c[0] withcolor red;
  draw c[0];

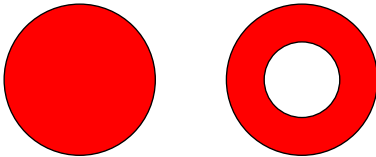
  pic := currentpicture;

endfig;

beginfig(2)
  draw pic;
  unfill c[1];
  draw c[1];

endfig;
end.
```

produces 2 eps figures *picture.1* (on the left) and *picture.2* (on the right):



The `pic := currentpicture` command has saved the first picture in `pic`. In the next figure, you can redraw it with `draw` and do modifications after!

It becomes interesting when you want to apply transformations (type **transformation**) to a picture. For example, if `pic` is a picture, `pic reflectedabout(p, q)` produces the symmetrical figure of `pic` with respect to the segment `[p, q]`

One has to know that transformations can be applied to **pictures**, but can be applied to **points**⁴, **paths** and even **pens**!

Here is the list of available transformations⁵, which does not need too much explanations⁵:

transformation	type of arguments
rotated	numeric
scaled	numeric
shifted	pair
slanted	numeric
transformed	transform
inverse	transform
xscaled	numeric
yscaled	numeric
zscaled	numeric
reflectedabout	pair, pair
rotatedaround	pair, numeric

Let's see that with an example:

```
numeric r ;
r := 1cm;

pair p [] ;
p[0] := (-2*r, 0);
p[1] := (-r, 0);
p[2] := (-2*r, r);
```

⁴as it was down in the first examples of this document, where **shifted** was applied to points

⁵see section 8.3 of [1] for further details

```
path t [] ;
t[0] := p[0] -- p[1] -- p[2] -- cycle;

beginfig(1)

  fill t[0] withcolor red;

  p[3] := (0, -0.5cm); p[4] := (0, 1.5cm);

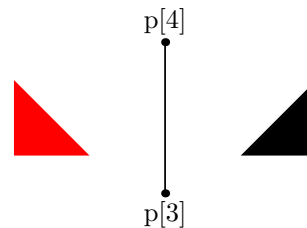
  draw p[3] -- p[4];

  draw currentpicture
    reflectedabout(p[3], p[4])
    withcolor black;

  dotlabel.bot("p[3]", p[3]);
  dotlabel.top("p[4]", p[4]);

endfig;
end.
```

which produces:



You also have to know that **identity** is the transformation neutral element's and thus, it can be used to create a transformation by composition from scratch.

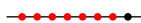
In the previous example, we transform the current picture, but you can also apply transformation to an object. For example, the following piece of code:

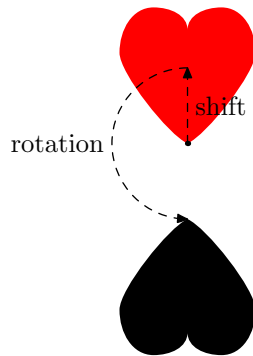
```
fill coeur withcolor red; % résumé

transform t;
t := identity shifted (0,1cm)
      rotated 180;

fill coeur transformed t; % black heart
```

produces:





6 Macros

This last section is quite short because macros are a little bit complicated in METAPOST (see [1] for all details). Anyway, METAPOST allows the user to define macros in this way:

```
def midpoint(expr p, q) =
  0.5[p, q];
enddef;

show midpoint((0,0), (1,0));

end.
```

This program outputs “(0.5, 0)”. **expr** indicates that p and q can be arbitrary expressions.

But macros defined with **def** are macros and thus, do not behave like a function or procedure in a usual programming language. That’s why, the following piece of code:

```
def middleshift(expr p, q) =
  0.5[p, q] + r
enddef;

pair r;
r := (1,1);

show middleshift((0,0), (1,1))

end.
```

works fine even if r isn’t declared when `middleshift` is defined (it works because r has been declared when the call to `middleshift` is done).

In order to have the behavior of a function, you have to use the **vardef** keyword.

7 Conclusion

METAPOST is a programming language dedicated to the generation of geometrical figures of high-quality in a format allowing to include them easily in ps or pdf documents. This tutorial has presented the main features of METAPOST, to allow you to quickly write your first METAPOST programs.

For all details about METAPOST, consider [1] as the absolute reference. For persons that want to discover extensions based on METAPOST, I suggest you to investigate an object oriented version of METAPOST[4] and an extension to draw 3D pictures [5, 6].

About this document

This document has been written to provide resources on METAPOST to non-English users (but this English version is, we hope, useful as well). At the time of this writing, it exists in French, Italian and English.

You’re invited to translate it, sources are available on the author’s [web page](#).

Thanks to Fabien Corder for carefully reviewing this document.

References

- [1] J. Hobby. A User’s Manual for MetaPost, 1994.
- [2] P. Wilson. Some Experiences in Running META-FONT and MetaPost, 2000.
- [3] D. P. Carlisle. Packages in the ‘graphics’ bundle, 2005.
- [4] D. Roegel. The METAOBJ tutorial and reference manual, 2001.
- [5] D. Roegel. La géométrie dans l’espace avec MetaPost, 2001.
- [6] D. Roegel. Space geometry with MetaPost, 2002.

