MetaPost: A Reference Manual

Peter Grogono

29 April 2011

Contents

1 Introduction **1**

2 Basic Concepts **2**
2.1 Version 2
2.2 Input format 2
2.3 The LaTeX File 2
2.4 Comments 4
2.5 Units 4
2.6 Scaling a Picture 5
2.7 Coordinates 5
2.8 Variables 5
2.9 Internal Variables 6
2.10 Declarations 6
2.11 Scope 7
2.12 Expressions 8
2.13 Assignment 9
2.14 Equations 9

3 Types **10**
3.1 Booleans 10
3.2 Strings 10
3.3 Colors 12
3.4 Numerics 13
3.5 Pairs 14
3.6 Pens 15
3.7 Transforms 17
3.8 Pictures 19
3.9 Lists 21

4 Paths **21**
4.1 Straight lines 22
4.2 Dots and Dashes 22
4.3 Circles, Disks, and Arcs 23
4.4 Curves 23

4.5 Parametric Paths . . . 25
4.6 Path Constructors . . . 27

5 Commands **28**
5.1 Drawing Commands . . . 28
5.1.1 `btex` and `verbatimtex` . . . 28
5.1.2 `clip` . . . 29
5.1.3 `draw` . . . 30
5.1.4 `drawarrow` . . . 30
5.1.5 `fill` . . . 31
5.1.6 `label` . . . 31
5.2 Non-drawing Commands . . . 32
5.2.1 `drawoptions` . . . 32
5.2.2 `filenametemplate` . . . 32
5.2.3 `for` . . . 33
5.2.4 `if` . . . 34
5.2.5 `message` . . . 35
5.2.6 `readfrom` . . . 35
5.2.7 `save` . . . 36
5.2.8 `show` and friends . . . 36
5.2.9 `write` . . . 36

6 Macros **36**
6.1 `def` macros . . . 37
6.2 `vardef` and other macro forms . . . 38

7 Macro Packages **39**
7.1 Boxes . . . 39
7.1.1 Creating and Drawing Boxes . . . 40
7.1.2 Positioning Boxes . . . 41
7.1.3 Oval Boxes . . . 42
7.2 Graphs . . . 43
7.3 TEX . . . 43
7.4 MetaUML . . . 44

8 Debugging **45**

9 Examples **46**
9.1 Euler Integration . . . 46
9.2 The Lorentz Transformation . . . 47
9.3 Dissipation . . . 50
9.4 Desargues' Theorem . . . 51
9.5 Cobweb Plots . . . 51
9.6 Three Dimensions . . . 52
9.7 Reading a File . . . 55

10 Links **56**

Index **57**

List of Figures

1 The effect of `linecap` and `linejoin` 16
2 Parametric curves 25
3 LaTeX fonts 29
4 Escape sequences for `filenametemplate` 32
5 Using `for...within` to obtain a partial analysis of a picture. 35
6 Box variables 40
7 Oval variables 42
8 Simulated coin tossing 44
9 The UML diagram generated from the code in Figure 10 45
10 Using `MetaUML` 45
11 Euler Integration 47
12 The Lorentz Transformation 48
13 Point numbering used in Figure 12 49
14 Area reduction in a dissipative system 50
15 Desargue's Theorem 52
16 Programming Desargues' Theorem 53
17 Cobweb plot for the logistic map 54
18 Coding the cobweb plot in Figure 17 54
19 Perspective view of a cube 55

1 Introduction

MetaPost is a picture-drawing language that generates diagrams and pictures in embedded postscript. MetaPost, an extension of Donald Knuth's MetaFont, was designed and implemented by John Hobby at AT&T Bell Laboratories.

The official guide for MetaPost, Hobby's *A User's Manual for* MetaPost, referred to in this manual as UMM, is written in a narrative style that is easy to read but hard to learn from. Several other people have written MetaPost manuals in narrative form. This manual is intended both to help beginners and to serve as a reference.

Examples of MetaPost code in this manual are intentionally short. This should make it easier to see the point of the example. With long and complex examples, it is sometimes hard to relate the code to the output. There are some longer examples at the end (§9 : 46).

This manual is not complete. Important omissions are indicated by a reference to UMM. In particular, UMM includes a complete grammar for MetaPost, whereas this manual relies mostly on examples.

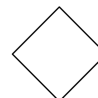
Conventions

- Since there are a large number of cross-references in this manual, we use a concise notation. For example, (§3.7 : 17) is a reference to Section 3.7 which is to be found on page 17.
- Most lists in MetaPost use comma as a separator. Unless stated otherwise, the term "list of X" means a list of the form $X_1, X_2, X_3, \ldots$.
- In grammar rules, ⟨*pred*⟩ indicates a predicate (`boolean`-valued expression) and ⟨*seq*⟩ indicates a sequence of commands.
- The command "`show` ⟨*expr*⟩" displays '`>>`' followed by the value of the expression (§5.2.8 : 36). In examples containing `show`, we put the code on the left of the page and the output to the right of it, like this:

```
show (3,4) + (5,6);                          >> (8,10)
```

- Similarly, we illustrate drawing techniques by putting code on the left and the picture produced on the right. In these examples, we usually omit the commands `beginfig` and `endfig`, as here:[1]

```
draw unitsquare rotated 45 scaled 25;
```



\# 15

- We use the symbol $\triangleq$ to mean "is defined to be equal to".

Acknowledgments I am indebted to Brian Shearing for ingenious hints on MetaPost usage — and, indeed, for informing me of its existence in the first place.

[1] Numbers in the margin, such as #15 here, are an editing aid and will not appear in in the final version of this manual.

2 Basic Concepts

2.1 Version

Features were added to MetaPost as it evolved. This manual describes Version 1.000 but also includes some useful features of later versions. The string constant `mpversion` (§3.2 : 10) will tell you what version you are using, but only if it is a fairly recent version.

2.2 Input format

MetaPost is a program that reads an input file with a name of the form ⟨*name*⟩`.mp` and writes several output files. By default, the names of the output files are ⟨*name*⟩`.1`, ⟨*name*⟩`.2`, ⟨*name*⟩`.3`, etc. For example, if the input file was `fig.mp` then the output files would be are `fig.1`, `fig.2`, `fig.3`, and so on. The precise form of the output file name is controlled by the command `filenametemplate` (§5.2.2 : 32). The output files are read by TEX, LATEX, or any program that accepts embedded postscript files.

In the input file, each figure is introduced by `beginfig(`n`)`, where n is the number of the figure, and terminated with `endfig`. Commands may occur outside figures as well as inside them. An input file has the general form:

```
⟨seq⟩

beginfig(1);
    ⟨seq⟩
endfig;

⟨seq⟩

beginfig(2);
    ⟨seq⟩
endfig;

....

beginfig(n);
    ⟨seq⟩
endfig;

end
```

2.3 The LATEX File

There are several ways of including MetaPost diagrams in a LATEX files. Two of them are described below.

Using the `hyperref` Package Assume that the file generated by METAPOST for figure n is called ⟨*name*⟩.⟨*number*⟩, where

- ⟨*name*⟩`.mp` is the name of the file processed by METAPOST
- ⟨*number*⟩ is the number of the figure

This is the format that METAPOST uses by default.

Include the `hyperref` package by writing

```
\usepackage[...]{hyperref}
```

in the preamble of your LaTeX document. This package has many useful options, indicated here by '...'. To insert METAPOST figure n in your document, write

```
\convertMPtoPDF{⟨name⟩.⟨number⟩}{⟨Xscale⟩}{⟨Yscale⟩}
```

in which:

- ⟨*Xscale*⟩ is the horizontal magnification factor
- ⟨*Yscale*⟩ is the vertical magnification factor

For example, to include Figure 3 generated from `figs.mp` without scaling, write:

```
\convertMPtoPDF{figs.3}{1}{1}}
```

Using the `graphicx` Package Include the `graphicx` package by writing

```
\usepackage{graphicx}
```

in the preamble of your LaTeX document.

The `graphicx` package may not recognize files created by METAPOST with default settings, because these files have a numeric extension. To avoid this problem, include this directive at the beginning of the METAPOST source file (§5.2.2 : 32):

```
filenametemplate "figs-%c.mps";
```

To insert METAPOST figure n in your document, write

```
\includegraphics{figs-n}
```

Use optional arguments to `\includegraphics` to scale the diagram if necessary. Use standard LaTeX commands to position it. One possibility is to use an equation environment to centre the diagram. For example:

```
$$ \includegraphics[width=5in]{mypics.7} $$
```

The file `mproof.tex` The TeX file `mproof` creates a `.dvi` file from MetaPost output. After the following run, `mproof.dvi` contains the text "`figs.65`" and the corresponding figure.

```
D:\Pubs>tex mproof figs.65
This is TeX, Version 3.141592 (MiKTeX 2.6)
(figs.65
figs.65: BoundingBox: llx = -1 lly = -3 urx = 61 ury = 72
figs.65: scaled width = 62.23178pt scaled height = 75.28038pt
)
[1]
Output written on mproof.dvi (1 page, 396 bytes).
```

2.4 Comments

MetaPost comments are delimited by the character '`%`' and the end of the line on which `%` appears, as in LaTeX.

2.5 Units

The default units of MetaPost are *Postscript points*. Postscript points are also called "big points" because they are slightly bigger than printer's points. The abbreviations for Postscript points and printer's points are `bp` and `pt`, respectively, as in TeX:

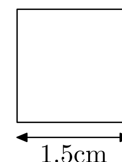
$$
\begin{aligned}
1 \text{ bp} &= 1/72 \text{ inches} \\
1 \text{ pt} &= 1/72.27 \text{ inches}
\end{aligned}
$$

MetaPost also understands the other units provided by TeX: `cm` (centimetres), `mm` (millimetres), `in` (inches), and `pc` (picas: 1`pc` $= \frac{1}{6}$`in`)). The units behave like constants defined with the appropriate values. For example, the value of `10*mm` is the length 10 millimetres expressed in Postscript points. The multiplication operator, `*`, can be omitted if the left operand is a number and the right operand is a variable. At the last line, MetaPost does not report an error for `n mm`, but parses it as a compound name (§2.8 : 5), `n.mm`, with no defined value.

```
n = 10;
show 10*mm;          >> 28.3464
show 10mm;           >> 28.3464
show n*mm;           >> 28.3464
show n mm;           >> n.mm
```

It is sometimes convenient to define a unit and use it to scale every number. Here is an example of picture scaling. The first line can be written as either "`u = 1cm;`" or "`u = cm;`".

```
u = 1cm;
draw (0u,0u) -- (1.5u,0u) -- (1.5u,1.5u) -- (0u,1.5u) -- cycle;
z1 = (0u,-0.2u);
z2 = (1.5u,-0.2u);
drawdblarrow z1 -- z2;
label.bot("1.5cm", 0.5[z1, z2]);
```



20

2.6 Scaling a Picture

There are several ways of making sure that a MetaPost diagram has the required size:

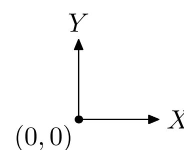
1. The first method, which might be called the *direct method*, is to use MetaPost's natural units (Postscript points). Lines of text on this page are about 450bp long, so a diagram that is 400bp wide and 300bp high fits nicely onto a page.
2. The second method is to use a better-known unit of length, such as inches or centimetres. The example above shows how to construct a diagram using centimetres as units.
3. The third method is to construct a diagram of any size and then scale it using the optional argument of `\includegraphics`. Here are some examples:

```
\includegraphics[width=4in]{pic-1.mps}
\includegraphics[height=7cm]{pic-2.mps}
\includegraphics[scale=2]{pic-2.mps}
```

2.7 Coordinates

MetaPost uses the conventional coordinate system of mathematics: X values increase across the page to the right, and Y values increase up the page.

```
drawarrow (0,0) -- (0,30);
drawarrow (0,0) -- (30,0);
dotlabel.llft(btex $(0,0)$ etex, (0,0));
label.rt(btex $X$ etex, (30,0));
label.top(btex $Y$ etex, (0,30));
```



22

Usually, it does not matter where the origin is, because the Postscript file that MetaPost generates defines a bounding box for all of the visible components of the picture. The origin might be centered, at the lower left, somewhere else inside the bounding box, or even outside it. However, if the origin is inside the picture, some of the coordinates may be negative. Postscript processors occasionally object to negative coordinates. This problem is rare but, if it occurs, it can be corrected by concluding each figure with the command

```
currentpicture := currentpicture shifted (-llcorner currentpicture);
```

In this command, `currentpicture` (§3.8 : 19) is an internal variable of type `picture`, `shifted` is a transform (§3.7 : 17) and `llcorner` gives the coordinates of the lower left corner of a picture.

2.8 Variables

Almost any string to which MetaPost does not assign a meaning can be used as a variable name. For example, `@&#$@` is a legal name. In practice, it is usually better to use more conventional names.

In general, a name consists of a *tag* optionally followed by a *suffix*. Any sequence of upper and lower case letters is a valid tag. Tags may also contain other characters, of which the most common is '.'. For example, MetaPost uses tags such as `b.n` and `b.s` to denote points on boxes.

The suffix may be a simple numeric, as in `p3`, or an expression in brackets, as in `p[i+3]`. Multidimensional indexes must be separated with dots or brackets: `p.3.4` is equivalent to `p[3][4]`.

The names `x`, `y`, and `z` play a special role. First, they are implicitly local (§2.11 : 7). Second, they are linked by the implicit equations

$$\begin{aligned} z &= (x, y) \\ \texttt{xpart}\ z &= x \\ \texttt{ypart}\ z &= y \end{aligned}$$

where x, y, and z stand for names that start with `x`, `y`, or `z` and are followed by a suffix. Thus `z.a[3]` denotes the pair `(x.a[3], y.a[3])`.

2.9 Internal Variables

MetaPost has a large number of internal variable names. Internal variables are implicitly declared within MetaPost but can be given new values by assignment (§2.13 : 9).

The command **`newinternal`** introduces new variables that behave like internal variables. This is not very useful in simple diagrams but might be useful in a macro package. To introduce new internal variables `Jack` and `Jill`, write:

```
newinternal Jack, Jill;
```

2.10 Declarations

A type name followed by a list of variable names serves to *declare* the variables named in the list. The type names are: `boolean`, `color`, `numeric`, `pair`, `path`, `pen`, `picture`, `string`, `transform`. Each type is described in detail later (§3 : 10).

If a variable is used without declaration, it is assumed to be a `numeric`. Occasionally, MetaPost is able to determine the type of a variable from its use, and an explicit declaration is not required. In most cases, however, a declaration is required. In particular, a declaration is required when:

- The type of the variable cannot be determined from the context in which it is used.
- A declaration can be used to "undefine" the variable, that is, to change its value to "unknown".
- An array is being declared. Following the declaration

  ```
  pair p[];
  ```

  the variables `p0`, `p1`, `p2`, . . . , `p[i]`, in which `i` is an integer expression, are all available for use.
- Multidimensional arrays can be declared with flexible syntax:

  ```
  path p[][], q[]r[];
  ```

  Corresponding values would include `p[1][2]`, `q[i]r6`, etc.

Declarations with numeric suffixes, such as

```
pair p1, p2;
```

are *not* allowed; the general form `p[]` described above must be used.

2.11 Scope

In general, the scope of a name starts at the point of its declaration or first use and extends to the end of the source file. There are two exceptions to this rule. First, the names `x`, `y`, and `z` play a special role to be described shortly. Second, the constructs `begingroup` and `endgroup` provide an environment in which local names can be declared.

MetaPost uses *groups* to provide local variables. The syntax of a group is

`begingroup` ⟨*seq*⟩ `endgroup`

in which ⟨*seq*⟩ is a sequence of commands. The last item in the sequence may be an expression, in which case the value of this expression becomes the value of the group.

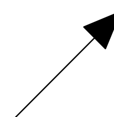
Within a group, the command `save` (§5.2.7 : 36) followed by a list of variable names makes those names local to the group. The global values of these names, if any, are saved. In the code below:

- `a` is used both globally and locally; its global value, 1, is restored on group exit.
- `b` is defined as a local variable within the group and becomes undefined on group exit.
- `c` is defined within the group but is not declared as a local variable. Consequently, it is treated as a global variable and its value is retained on group exit.

```
a = 1;
begingroup
    save a, b;
    a = 2;
    b = 3;
    c = 4;
endgroup;                                   >> 1
show a;                                     >> b
show b;                                     >> 4
show c;
```

The command `interim` allows you to redefine a internal variable inside a group without changing its global value. For example, the following statements draw an arrow (§5.1.4 : 30) with an elongated head without changing the global value of `ahlength`:

```
begingroup;
    interim ahlength := 15;
    drawarrow (0,0) -- (40,40);
endgroup;
```



66

The scope of the special names `x`, `y`, and `z` is restricted to a particular figure. That is, the code on the left has the effect of the code on the right (of course, the `beginfig`/`endfig` environment also creates a figure file, etc.):

```
beginfig(n)
    ⟨seq⟩
endfig
```

```
begingroup
    save x, y, z;
    ⟨seq⟩
endgroup
```

> ***Recommendation:*** `save` ***all of the variables in a figure that are not needed for other figures.***

2.12 Expressions

An expression in METAPOST is a ⟨*primary*⟩, ⟨*secondary*⟩, ⟨*tertiary*⟩, or ⟨*expression*⟩. A binary operator with highest precedence is called a ⟨*primary binop*⟩ and its operands must either be ⟨*primary*⟩s or be enclosed between parentheses. Since unary operators have higher precedence than binary operators, there are four levels of precedence. Binary operators are organized as follows, with the top line having highest priority.

⟨*unop*⟩	`abs` `angle` `arclength` ...
⟨*primary binop*⟩	`*` `/` `**` `and` `dotprod` `div` `mod` `infont`
⟨*secondary binop*⟩	`+` `-` `++` `+-+` `or` `intersectionpoint` `intersectiontimes`
⟨*tertiary binop*⟩	`&` `<` `<=` `<>` `=` `>=` `>` `cutafter` `cutbefore`

However, division (`/`) has highest precedence if both of its operands are numbers. This means that, for example:

$$\texttt{sqrt 2/3} \equiv \sqrt{2/3}$$
$$\texttt{sqrt n/d} \equiv \sqrt{n}/d$$
$$\texttt{sqrt(n/d)} \equiv \sqrt{n/d}$$

The exponent operator `**` has the same precedence as multiply (`*`). Consequently:

$$\texttt{3*a**2} \equiv (3a)^2.$$

The meaning of most of the operators are conventional. There are special operators for the hypotenuse and difference of squares:

$$\texttt{a ++ b} \equiv \sqrt{a^2+b^2}$$
$$\texttt{a +-+ b} \equiv \sqrt{a^2-b^2}$$

There are also many operators with names, both unary and binary. The operators that may be used with each type, and their meanings, are discussed with the corresponding types (§3 : 10).

Some expressions use the keyword `of`. The general format of these expressions is

⟨*of operator*⟩ ⟨*expression*⟩ `of` ⟨*primary*⟩

where ⟨*of operator*⟩ is one of:

```
arctime   direction   directiontime   directionpoint   penoffset   point
postcontrol   precontrol   subpath   substring
```

For example (§3.2 : 10):

```
substring (2,4) of "abcde"  ≡  "cd".
```

2.13 Assignment

The assignment operator is `:=`, and it has the usual effect of giving the variable on its left side the new value obtained by evaluating its left side. Previous values of the variable are discarded.

Assignment (`:=`) should be distinguished from equality (`=`), which is used in equations, as described in the next section.

2.14 Equations

Variables may receive values either explicitly by assignment, as above, or implicitly by linear equations. Equations use the equality comparison, `=`. Equations may be built up from numerics (§3.4 : 13), pairs (§3.5 : 14), or colors (§3.3 : 12).

MetaPost *must be able solve equations before any pictures that use their values are drawn.*

MetaPost easily solves the equations below, obtaining $a = 3,\ b = 4,\ c = 5$. Note that `2*a` can be abbreviated to `2a`, etc. The effect of the command `showdependencies` is to display the inferences that MetaPost has made from the equations at that point, as shown at the right.

```
2a + 2b + c = 19;
3a-b = 5;                                      >> a=-0.125c+3.625
showdependencies;                              >> b=-0.375c+5.875
4b-3c = 1;
```

A value obtained by solving equations may be changed by assignment. However, the assignment changes *only* the variable assigned.

```
a = 2;              % gives a the value 2
a = b;              % gives b the value 2
a := 3;             % gives a the value 3 but does not change b
```

The symbol `whatever` introduces a new, anonymous variable. It may be used to avoid introducing variables unnecessarily. For example, we could find the intersection `i` of two lines `(p,q)` and `(r,s)` by using the fact that `a[p,q]` is a point on the line from `p` to `q` (§2.12 : 8).

```
i = a[p,q];
i = b[r,s];
```

Since the names of the numerics `a` and `b` are not needed, we could write instead:

```
i = whatever[p,q] = whatever[r,s];
```

It does not matter that the two instances of `whatever` will have different values.

3 Types

The types provided by MetaPost are: `boolean`, `string`, `color`, `numeric`, `pair`, `path`, `pen`, `picture`, and `transform`. All of them are described in this section except `path`, which has its own section (§4 : 21).

3.1 Booleans

The values of `boolean` are `true` and `false`. The binary comparison operators `=`, `<>`, `<`, `<=`, `=>`, and `>` return `boolean` values. The unary operator `not` and the binary operators `and` and `or` take `boolean` operands and return the expected `boolean` values. Booleans are also needed for `if` statements (§5.2.4 : 34).

The equality operators `=` and `<>` work with all types. The comparison operators, `<`, `<=`, `=>`, and `>`, work with most types and have expected values for reasonable types.

Type names can be used as predicates:

`boolean` b	$\triangleq$	b is a `boolean`
`color` c	$\triangleq$	c is a `color`
`rgbcolor` c	$\triangleq$	c is a `rgbcolor`
`cmykcolor` c	$\triangleq$	c is a `cmykcolor`
`numeric` n	$\triangleq$	n is a `numeric`
`pair` p	$\triangleq$	p is a `pair`
`path` p	$\triangleq$	p is a `path`
`pen` p	$\triangleq$	p is a `pen`
`picture` p	$\triangleq$	p is a `picture`
`string` s	$\triangleq$	s is a `string`
`transform` t	$\triangleq$	t is a `transform`

Other predicates include:

`odd` n	$\triangleq$	the closest integer to n is odd
`cycle` p	$\triangleq$	path p is a cycle
`known` x	$\triangleq$	x has a value
`unknown` x	$\triangleq$	x does not have a value

3.2 Strings

Strings are sequences of characters bounded by double quotes (`"`). Strings may not contain double quotes or line breaks. The binary comparison operators `=`, `<>`, `<`, `<=`, `=>`, and `>` accept `string` operands and use the underlying character codes for ordering.

Functions for strings include:

`string` s	$\triangleq$	true if s is a `string`
`length` s	$\triangleq$	the number of characters in s

Constants and functions yielding strings include:

`ditto`	$\triangleq$	the one-character string `"`
`mpversion`	$\triangleq$	a string giving the version of MetaPost
`substring` (m, n) `of` s	$\triangleq$	characters m to n of string s
`scantokens`(s)	$\triangleq$	the result of parsing the token sequence s
`char` n	$\triangleq$	a string consisting of the character with ASCII code n
`decimal` n	$\triangleq$	a string representing the decimal representation of the number n
`str` s	$\triangleq$	the string representation of the suffix s
s `&` t	$\triangleq$	the concatenation of strings s and t

The indexing convention for strings is the same as C: the first character has index 0 and the second argument of `substring` indexes the first character following the selection:

`substring (2,4) of "abcde"` $\equiv$ `"cd"`.

The expression

s `infont` f

yields a picture (§3.8 : 19) consisting of the string s typeset in the font f. There is a `defaultfont` (usually `cmtex10`) with a size `defaultscale` (usually 1).

```
draw "MetaPost" infont "cmsl12" scaled 2 rotated 30;
```

MetaPost

74

Calling `scantokens` invokes MetaPost's input routine. It can be used as a general conversion function. For example, this call gives `pi` the numerical value 3.14159:

```
pi := scantokens("3.14159");
```

`scantokens` is one of the few MetaPost operations that creates a list (§3.9 : 21). See also (§9.7 : 55) for a more interesting application of `scantokens`.

MetaPost does not invoke functions in `btex ... etex` environments but see (§7.3 : 43).

3.3 Colors

A value of type `color` has three components, corresponding to the red, green, and blue components of a colour. Each component is clamped to the range $[0, 1]$. Colours are written `(r,g,b)`. MetaPost can solve linear equations involving colours. Two colours may be added or subtracted; a colours may be multiplied and divided by numerics; colours may be used in linear equations.

```
show white/4;                    >> (0.25,0.25,0.25)
show 0.2green + 0.6blue;         >> (0,0.2,0.6) )
```

Constants and functions for colours include:

`color` c	$\triangleq$	true if c is a RGB colour
`black`	$\triangleq$	$(0, 0, 0)$
`white`	$\triangleq$	$(1, 1, 1)$
`red`	$\triangleq$	$(1, 0, 0)$
`green`	$\triangleq$	$(0, 1, 0)$
`blue`	$\triangleq$	$(0, 0, 1)$
`redpart`(r, g, b)	$\triangleq$	r
`greenpart`(r, g, b)	$\triangleq$	g
`bluepart`(r, g, b)	$\triangleq$	b
`background`	$\triangleq$	the background colour (default $=$ `white`)

To change the foreground colour (i.e., the drawing colour), use `drawoptions` (§5.2.1 : 32).

Later versions MetaPost support the CMYK colour model as well as RGB. A CMYK colour is represented as a four-tuple `(c,m,y,k)` with `(1,1,1,1)` representing black and `(0,0,0,0)` representing white. The following constants and functions are provided:

`rgbcolor`	$\triangleq$	a synonym for `color`
`cmykcolor` c	$\triangleq$	true if c is a CMYK colour
`cyanpart`(c, y, m, k)	$\triangleq$	c
`magentapart`(c, y, m, k)	$\triangleq$	y
`yellowpart`(c, y, m, k)	$\triangleq$	m
`blackpart`(c, y, m, k)	$\triangleq$	k

Colour expressions usually appear in `withcolor` clauses in conjunction with commands such as `draw` (§5.1.3 : 30), `drawoptions` (§5.2.1 : 32), `fill` and `filldraw` (§5.1.5 : 31). `withcolor` works for both RGB and CMYK colours, but the two systems should not be mixed within a single picture.

3.4 Numerics

MetaPost uses fixed-point arithmetic with $2^{-16} = 1/65536$ as the unit for the `numeric` type. The largest number that can be represented is about 4096.

The lengths that MetaPostcan represent range from 1/65536 bp $\approx$ 5.38 nanometres to 4096 bp $\approx$ 1.4 metres. For comparison, the wavelength of blue light is about 400 nanometres and the length of standard paper is about 0.29 metres, depending on where you live.

The arithmetic operators `+`, `-`, `*`, and `/` can be used with numerics. The binary comparison operators `=`, `<>`, `<`, `<=`, `=>`, and `>` accept `numeric` operands and provide the conventional ordering.

Numeric constants and expressions include:

`epsilon` $\triangleq$ 1/65536 (the smallest value that MetaPost can represent)
`infinity` $\approx$ 4095.99998 (the largest value that MetaPost can represent)
`day` $\triangleq$ current day (of the month)
`month` $\triangleq$ current month
`year` $\triangleq$ current year
`time` $\triangleq$ job start time (minutes since midnight)

Numeric functions include:

`decimal` n $\triangleq$ the decimal string corresponding to n
`numeric` n $\triangleq$ `true` if n is numeric
`odd` n $\triangleq$ true if the closest integer to n is odd
`m div n` $\triangleq$ $\lfloor m/n \rfloor$ (integer division)
`m mod n` $\triangleq$ $m - n\lfloor m/n \rfloor$ (integer remainder)

`abs` x $\triangleq$ $|x|$, the absolute value of x
`sqrt` x $\triangleq$ $\sqrt{x}$
`sind` x $\triangleq$ $\sin x$ (x in degrees)
`cosd` x $\triangleq$ $\cos x$ (x in degrees)
`mexp` x $\triangleq$ $e^{x/256}$
`mlog` x $\triangleq$ $256 \ln x$
`floor` x $\triangleq$ the greatest integer less than or equal to x
`ceiling` x $\triangleq$ the least integer greater than or equal to x
`round` x $\triangleq$ the integer closest to x
`x++y` $\triangleq$ $\sqrt{x^2 + y^2}$
`x+-+y` $\triangleq$ $\sqrt{x^2 - y^2}$
`uniformdeviate` x $\triangleq$ a random number uniformly distributed in $[0, x]$
`normaldeviate` $\triangleq$ a random number with normal distribution, $\mu = 0$, $\sigma = 1$

3.5 Pairs

A `pair` is a tuple of two `numerics` that is most often used to represent a position in the two-dimensional plane of the picture but can also be used to represent a direction. Literal pairs are written as `(m,n)` in which `m` and `n` are `numerics`.

Pairs can be added and subtracted (like vectors), multiplied or divided by `numerics`, and used in linear equations. *Mediation* abbreviates a common operation:

$$\mathtt{r[p,q]} \triangleq \mathtt{p} + \mathtt{r} * (\mathtt{q} - \mathtt{p}).$$

If $0 \leq \mathtt{r} \leq 1$, the result is a point between the points `p` and `q`, with $\mathtt{r} = 0$ corresponding to `p` and $\mathtt{r} = 1$ corresponding to `q`. All values of `r` are legal:

```
z1 = (0,0);
z2 = (0,50);
draw z1--z2;
dotlabel.rt(btex $4 \over 3$ etex, 4/3[z1,z2]);
dotlabel.rt(btex $2 \over 3$ etex, 2/3[z1,z2]);
dotlabel.rt(btex $1 \over 3$ etex, 1/3[z1,z2]);
```

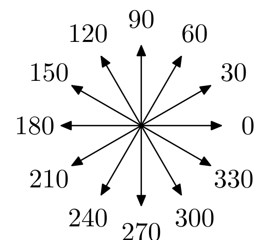
23

The names `x`, `y`, and `z` play a special role:

- They are undefined by `beginfig` (§2.2 : 2).
- `z`⟨*suffix*⟩ is defined to be equal to `(x`⟨*suffix*⟩`,y`⟨*suffix*⟩`)`, where ⟨*suffix*⟩ is a number, index, or name (§2.8 : 5). Thus `z[3].t` can be abbreviated to `z3t` and is equivalent to the pair `(x3t,y3t)`.

The `pair (x,y)` can be used to represent the direction $\theta = \tan^{-1}(\mathtt{y}/\mathtt{x})$. Angles in MetaPost are measured in degrees. For example, `dir 90` ≡ `pair(0,1)`.

```
smallRad = 30;
bigRad = 40;
for angle = 0 step 30 until 330:
    s := sind angle;
    c := cosd angle;
    drawarrow origin -- (smallRad*c, smallRad*s);
    label(decimal angle, (bigRad*c, bigRad*s));
endfor;
```



13

Constant pairs include:

`origin`	$\triangleq$	$(0,0)$
`up`	$\triangleq$	`(0,1)`
`down`	$\triangleq$	`(0,-1)`
`left`	$\triangleq$	`(-1,0)`
`right`	$\triangleq$	`(1,0)`

Functions for pairs include:

$$
\begin{array}{llll}
\texttt{xpart}(x,y) & \triangleq & x & \\
\texttt{ypart}(x,y) & \triangleq & y & \\
\texttt{abs}\ z & \triangleq & \sqrt{x^2+y^2} & \text{(hypotenuse)} \\
\texttt{unitvector}\ z & \triangleq & z/\texttt{abs}\ z & \text{(normalize)} \\
\texttt{angle}\ z & \triangleq & \tan^{-1}(y/x) & \text{(in degrees)} \\
\texttt{round}\ z & \triangleq & (\texttt{round}\ x\texttt{,}\ \texttt{round}\ y) & \\
z_1\ \texttt{dotprod}\ z_2 & \triangleq & x_1y_1 + x_2y_2 & \text{(inner product)} \\
r\,\texttt{[}z_1, z_2\texttt{]} & \triangleq & z_1 + r(z_2 - z_1) & \text{(mediation)} \\
\texttt{dir}\ \theta & \triangleq & (\texttt{cosd}\,\theta,\ \texttt{sind}\,\theta) & \text{(direction)}
\end{array}
$$

3.6 Pens

All objects are drawn with a pen. The command

`pickup` ⟨*pen expression*⟩

selects a pen with characteristics defined by ⟨*pen expression*⟩ with syntax

$$
\left.\begin{array}{l}\texttt{pencircle}\\ \texttt{pensquare}\end{array}\right\}\ \langle\textit{transform}\rangle
$$

where ⟨*transform*⟩ is an affine transformation (§3.7 : 17). Naturally, `pencircle` gives a round nib and `pensquare` gives a square nib. The operator `pensquare` is actually a macro defined by

```
pensquare ≜ makepen((-0.5,-0.5)--(0.5,-0.5)--(0.5,0.5)--(-0.5,0.5)--cycle)
```

and, in general, `makepen` will construct a pen nib from any closed path (§4 : 21). The operator `makepath` is the inverse of `makepen`: if p is a pen, then `makepath` p is the polygon that it uses as a nib. Thus:

`makepath pencircle` ≡ `fullcircle`.

It is best to use a round nib (`pencircle`) when drawing dotted and dashed lines (§4.2 : 22); square or polygonal pens yield unpredictable results.

The constant `nullpen` is a pen with no useful properties. A function that is supposed to return a `pen` but cannot do so should return `nullpen`.

The default pen is quite thin:

`defaultpen` ≜ `pencircle scaled 0.5 bp`.

Thick pens are defined by simple scaling:

```
pencircle scaled 5pt
```

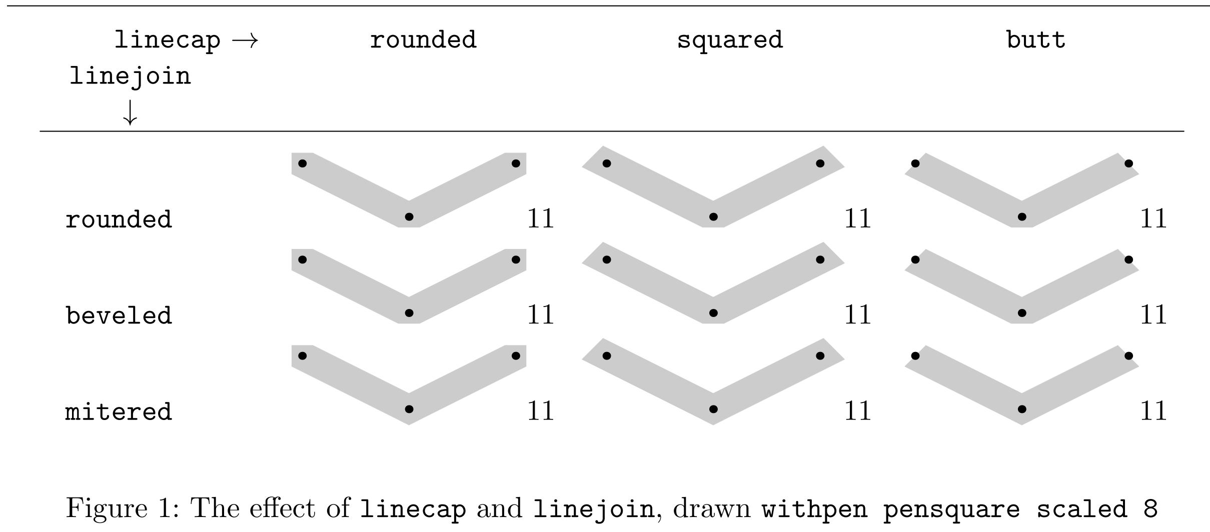


Figure 1: The effect of `linecap` and `linejoin`, drawn `withpen pensquare scaled 8`

You can use `xscaled` and `yscaled` to obtain an elliptical or rectangular nib:

```
path p;
drawoptions(withpen pencircle
    xscaled 2bp yscaled 0.1bp);
p = (0,0) for i = 1 upto 100:
    ..  (0.6i*cosd(36i),0.6i*sind(36i)) endfor;
draw p;
drawoptions();
```

68

The shape of the end of each line is determined by the internal variable `linecap`. When lines change direction, the shape of the corner is determined by `linejoin`. Figure 1 illustrates both.

$$\texttt{linecap} \;:=\; \begin{cases} \texttt{rounded} & \text{(default)} \\ \texttt{squared} & \\ \texttt{butt} & \end{cases} \qquad \texttt{linejoin} \;:=\; \begin{cases} \texttt{rounded} & \text{(default)} \\ \texttt{beveled} & \\ \texttt{mitered} & \end{cases}$$

A long spike may be produced when `mitered` lines meet at an acute angle. MetaPost changes the join shape to `beveled` if

$$\frac{\text{miter length}}{\text{line width}} > \texttt{miterlimit}.$$

The default value of the internal variable `miterlimit` is 10.0; it can be changed by assignment.

For a sequence of `draw` (§5.1.3 : 30) and `fill` (§5.1.5 : 31) commands with a particular kind of pen, use

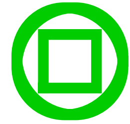
`drawoptions ( ⟨text⟩ )`

to set the pen characteristics and

`drawoptions ( )`

to restore the defaults. The argument ⟨*text*⟩ can specify `dashed` (§4.2 : 22), `withcolor` (§3.3 : 12), and `withpen` (§3.6 : 15) values.

```
drawoptions(
    withcolor 0.8 green
    withpen pensquare scaled 4 );
d = 10;
draw unitsquare scaled 2d shifted (-d,-d);
draw fullcircle scaled 4d;
drawoptions();
```



26

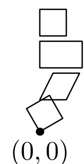
3.7 Transforms

A variety of affine transformations can be applied to pictures. These are transform expressions (all angles are measured in in degrees):

(x, y)	`rotated`	θ	$\triangleq$	$(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$
(x, y)	`slanted`	a	$\triangleq$	$(x + ay, y)$
(x, y)	`scaled`	a	$\triangleq$	(ax, ay)
(x, y)	`xscaled`	a	$\triangleq$	(ax, y)
(x, y)	`yscaled`	a	$\triangleq$	(x, ay)
(x, y)	`shifted`	(a, b)	$\triangleq$	$(x + a, y + b)$
(x, y)	`zscaled`	(a, b)	$\triangleq$	$(ax - by, bx + ay)$
(x, y)	`reflectedabout`	(p, q)	$\triangleq$	The reflection of (x, y) in the line (p, q).
(x, y)	`rotatedaround`	(p, θ)	$\triangleq$	(x, y) rotated about point p through angle θ.

Transforms are applied by writing a transform expression after a picture expression. Transform expressions can be combined by concatenating them.

```
dotlabel.bot(btex (0,0) etex, origin);
path bx;
bx = unitsquare scaled 10;
draw bx shifted (0,36);
draw bx xscaled 1.6 shifted (0,24);
draw bx slanted 0.5 shifted (0,12);
draw bx rotated 30;
```



8

As usual, the type may be used as a predicate:

$$\texttt{transform}\ t \triangleq \texttt{true}\ \text{if}\ t\ \text{is a transform.}$$

Variables of type `transform` may be declared. If t is a transform variable, then

p `transformed` t

is the picture p transformed by t and

`inverse` t

is the inverse of t. The constant `identity` is a transform that has no effect:

$$\texttt{draw}\ p\ \texttt{transformed identity} \quad \equiv \quad \texttt{draw}\ p$$

The following example defines and applies a transform called `reflect`.

```
transform reflect;
reflect = identity xscaled -1;
draw btex Mirror, mirror, on the wall etex
    transformed reflect;
```

Mirror, mirror, on the wall

27

A transform has six parameters. The mapping from (x, y) to (x', y') is given, in homogeneous coordinates, by

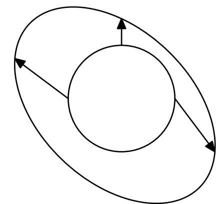
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{xx} & t_{xy} & t_x \\ t_{yx} & t_{yy} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The values of the parameters for a transform T are referred to in MetaPost as

$$\begin{array}{lcl@{\qquad\qquad}lcl} t_x & = & \texttt{xpart}\ T & t_y & = & \texttt{ypart}\ T \\ t_{xx} & = & \texttt{xxpart}\ T & t_{yx} & = & \texttt{yxpart}\ T \\ t_{xy} & = & \texttt{xypart}\ T & t_{yy} & = & \texttt{yypart}\ T \end{array}$$

Transforms may be defined by equations. Each equation says that a point p is mapped to a point p' and determines two of the transform parameters; consequently, three equations are needed. In the following example, the transform `t` is defined by its effect on three points.

```
z1 = (-20,0); z2 = (-40,15);
z3 = (0,20); z4 = (0,30);
z5 = (20,0); z6 = (35,-20);
draw fullcircle scaled 40;
drawarrow z1--z2;
drawarrow z3--z4;
drawarrow z5--z6;
transform t;
z1 transformed t = z2;
z3 transformed t = z4;
z5 transformed t = z6;
draw fullcircle scaled 40 transformed t;
```



28

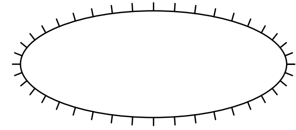
The parameters themselves can be used to define the transformation. For example, the equations

```
xxpart t = yypart t;
xypart t = -yxpart t;
```

specify that transform `t` preserves shapes. Two additional equations, defining the effect of `t` on two points, are required to define `t` completely.

The following example combines several features of METAPOST: the command `direction` (§4.5 : 25) gives the direction of the tangent vector at a point on the curve; the command `angle` (§3.4 : 13) converts this direction to an angle θ; rotating by $\theta + 90°$ gives a direction normal to the curve; and the command `point` (§4.5 : 25) gives the position of a point on the curve.

```
path p;
p := fullcircle xscaled 100 yscaled 40;
draw p;
for i = 0 upto 40:
    t := i/5;
    draw ((0,0)--(3,0))
        rotated (angle direction t of p - 90)
        shifted (point t of p);
endfor;
```



47

3.8 Pictures

Anything that can be drawn by METAPOST can be stored in a picture variable. Several primitives, including `draw` (§4 : 21), store their results in the internal variable `currentpicture`. Pictures can be transformed (§3.7 : 17).

The following expressions yield a picture:

s `infont` f	:	string s set in font f (§3.2 : 10)
`btex ... etex`	:	raw TEX strings (§5.1.1 : 28)
`TEX( ... )`	:	processed TEX strings(§7.3 : 43)
`nullpicture`	:	an empty picture
`currentpicture`	:	the “current” picture, destination of `draw`, etc.

These operators return a value associated with a picture p:

`center` p	$\triangleq$	the centre of p.
`length` p	$\triangleq$	the number of components of p.
`llcorner` p	$\triangleq$	the lower left corner of p.
`lrcorner` p	$\triangleq$	the lower right corner of p.
`ulcorner` p	$\triangleq$	the upper left corner of p.
`urcorner` p	$\triangleq$	the upper right corner of p.

After defining the picture `p` with the commands

```
picture p;
p = btex $\displaystyle\int_0^\infty e^{-x} \, \sin x \, dx $ etex;
```

we can draw the picture and its corner points:

```
draw p;
dotlabel.ulft("UL", ulcorner p);
dotlabel.urt ("UR", urcorner p);
dotlabel.llft("LL", llcorner p);
dotlabel.lrt ("LR", lrcorner p);
```

UL UR

$$\int_0^\infty e^{-x} \sin x\, dx$$

LL LR

2

The operators `llcorner`, `lrcorner`, `ulcorner`, and `urcorner` define the *bounding box* of a picture. They can be used to measure the size of a box. Following on from the above example,

```
pair dim;
dim := (urcorner p) - (llcorner p);
width := xpart dim;
height = ypart dim;
```

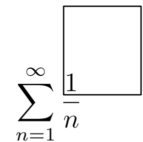
gives `width` = 71.3919 and height = 23.1722.

If the bounding box is not what you want, you can change it. The command

`setbounds` v `to` p

makes the picture variable v behave as if its bounding box is the path p.

```
picture p;
p = btex $\displaystyle\sum_{n=1}^{\infty} {1 \over n}$ etex;
path q;
q = bbox p shifted (20,20);
setbounds p to q;
draw p;
draw q;
```



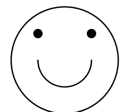
67

Recent versions of MetaPost include a macro `image` that can be used in places that would otherwise require `currentpicture`. The expression

`image(` ⟨*seq*⟩ `)`

yields the `picture` object constructed by the sequence ⟨*seq*⟩ of drawing commands.

```
picture smiley;
smiley := image(
    draw fullcircle scaled 40;
    draw halfcircle scaled 20 rotated 180;
    filldraw fullcircle scaled 3 shifted (-10,10);
    filldraw fullcircle scaled 3 shifted (10,10);
);
draw smiley;
```



70

The following predicates can be applied to any object, but are most meaningful for parts of a picture. A "stroked picture component" is a part of a picture that was drawn by moving a pen

(as opposed to setting text, etc.). These predicates can be used to analyze the structure of a picture (§5.2.3 : 33).

`bounded` x $\triangleq$ x is a picture with a bounding box
`clipped` x $\triangleq$ x is a picture that has been clipped
`dashpart` x $\triangleq$ x is the dash pattern of a path in a stroked picture
`filled` x $\triangleq$ x is a filled outline
`pathpart` x $\triangleq$ x is the path of a stroked picture component
`penpart` x $\triangleq$ x is the pen of a stroked picture component
`stroked` x $\triangleq$ x is a stroked line
`textual` x $\triangleq$ x is typeset text

3.9 Lists

Lists have a rather ghostly existence in METAPOST: there is no type *list*, but there are a few constructions that create and consume lists.

The function `scantokens` converts a string to a list of tokens. The expression x `step` d `until` y generates the list x, $x + d$, $x + 2d, \ldots$, stopping when $x + nd \geq y$. For the common case $d = 1$, there are macros `upto` and `downto`:

`upto` $\triangleq$ `step 1 until`
and `downto` $\triangleq$ `step -1 until`

The `for` statement (§5.2.3 : 33) has a form

`for` ⟨*var*⟩ `=` ⟨*list*⟩`:` ⟨*seq*⟩ `endfor`

which assigns the variable ⟨*var*⟩ to each item in ⟨*list*⟩. The list can be an actual list of tokens separated by commas, as in

```
for i = 1, 2, 4, 8, 16:  ...  endfor
```

or a list generated by `scantokens`, as in

```
for i = scantokens("1, 2, 4, 8, 16"):  ...  endfor
```

4 Paths

Paths are piecewise straight lines, curves, and any combination of these. Path variables of type `path` may be declared and there are various kinds of *path expressions*. The following examples use the command `draw` (§5.1.3 : 30) to draw paths.

4.1 Straight lines

The binary operator '`--`' constructs a straight line between its point operands. It may be used in a sequence, so that

```
draw z0 -- z1 -- z2 -- z3;
```

draws three line segments, connecting `z0` to `z1`, and so on.

If the last point is the internal name `cycle`, the last line segment returns to the starting point. The keyword `cycle` can also be used as a predicate:

$$\texttt{cycle}\ p \quad \triangleq \quad \text{true if path } p \text{ is a cycle}$$

The expression `p & q` yields the concatenation of the paths `p` and `q`, provided that the last point of `p` is identical to the first point of `q`.

4.2 Dots and Dashes

The general syntax for a dashed line is:

draw ⟨*path*⟩ dashed ⟨*pattern*⟩

The basic patterns are:

	`evenly`	:	evenly-spaced dashes, 3 bp long,
and	`withdots`	:	dots 5 bp apart.

Patterns may be transformed (§3.7 : 17):

	`shifted` ⟨*pair*⟩	:	each dash is displaced by ⟨*pair*⟩,
and	`scaled` ⟨*numeric*⟩	:	each dash is scaled by ⟨*numeric*⟩.

```
draw (0,30)--(120,30) dashed withdots;
draw (0,20)--(120,20) dashed evenly;
draw (0,10)--(120,10) dashed evenly shifted (3,3);
draw (0, 0)--(120, 0) dashed evenly scaled 1.5;
```

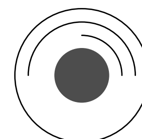
3

Dashed and dotted lines are best drawn using a pen with a circular nib (§3.6 : 15).

4.3 Circles, Disks, and Arcs

The command `fullcircle` draws a circle with unit diameter at the origin. Apply transformations (§3.7 : 17) to the circle to modify the size and position. The command `halfcircle` is similar to `fullcircle` but draws only the part of the circle above the X axis. The command `quartercircle` draws only the first quadrant of the circle.

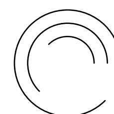
```
draw fullcircle scaled 50;
draw halfcircle scaled 40;
draw quartercircle scaled 30;
filldraw fullcircle scaled 20 withcolor 0.3white;
```



14

General arcs can be drawn using the `subpath` command (§4.5 : 25). The path length of a full circle is 8, and so one unit corresponds to 45°. Thus the arcs in the picture below have lengths of 135°, 225°, and 315°.

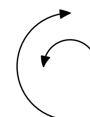
```
draw subpath (0, 3) of fullcircle scaled 20;
draw subpath (0, 5) of fullcircle scaled 30;
draw subpath (0, 7) of fullcircle scaled 40;
```



38

Curved arrows can be drawn in the same way. The normal direction of an arc is counterclockwise, but you can use `reverse` (§4.5 : 25) to get a clockwise arc.

```
drawarrow subpath (0, 4) of fullcircle scaled 20;
drawarrow reverse (subpath (2, 6) of fullcircle scaled 40);
```



81

4.4 Curves

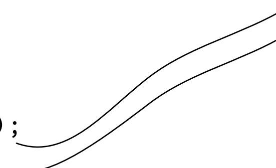
The binary operator '`..`' is used like '`--`', but it draws cubic Bézier curve instead of straight lines. The command

```
draw z0..z1..z2..z3;
```

draws a Bézier curve that passes through the points `z0`, `z1`, `z2`, and `z3`.

The binary operator '`...`' is similar to '`..`' but tries to avoid inflection points. The difference between '`..`' and '`...`' is generally small, but can be seen in the following example, in which the lower curve uses '`...`'.

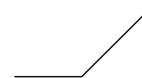
```
z0 = (0,0); z1 = (50,25); z2 = (150,50);
draw z0{dir -20} .. z1 .. {dir 30}z2;
draw (z0{dir -20} ... z1 ... {dir 30}z2) shifted (0,-10);
```



53

The binary operator '`---`' is similar to '`--`' but ensures a smooth transition between straight and curved sections of a path, as illustrated in the two examples below.

```
draw (0,0) -- (25,0) .. (50,25);
```



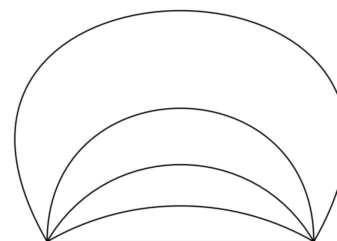
51

```
draw (0,0) --- (25,0) .. (50,25);
```

52

The direction of the curve may be specified at a point by a qualifier `{p}`, in which `p` is a pair representing a direction (§3.5 : 14). The qualifier may be placed either before or after the point.

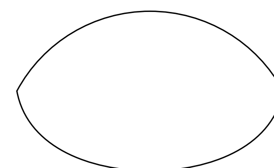
```
for i = 0 step 30 until 120:
    draw (0,0){dir i}..(100,0);
endfor;
```



12

The curvature of the curve can be specified at its end points using `curl`. The argument of `curl` can have any positive value; smaller values give smaller curvature.

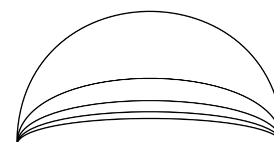
```
draw (0,0){curl 1} .. (50,30) .. {curl 1}(100,0);
draw (0,0){curl 5} .. (50,-30) .. {curl 5}(100,0);
```



44

The `tension` of a curve may be changed between any pair of points. The minimum value of `tension` is $\frac{3}{4}$; the default value is 1; higher values give straighter curves.

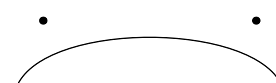
```
for i = 1 upto 5:
    draw (0,0){dir 90} .. tension i .. {dir 270}(100,0);
endfor;
```



43

Control points are points that are not on the curve but which “attract” the curve. They can be specified between points with syntax “`controls` p `and` q”, where p and q are points.

```
z1 = (10,25);
z2 = (90,25);
dotlabel("", z1);
dotlabel("", z2);
draw (0,0) .. controls z1 and z2 .. (100,0);
```



24

4.5 Parametric Paths

A path may be considered as a set of points $\{(X(t), Y(t)\}$, where t is the path *parameter*. Given a value of t, we can find the corresponding point $(X(t), Y(t))$ on the path and the direction of the path. Conversely, given a point or a direction on the path, we can find the value of t. We can use t to find where curves intersect and to extract parts of curves.

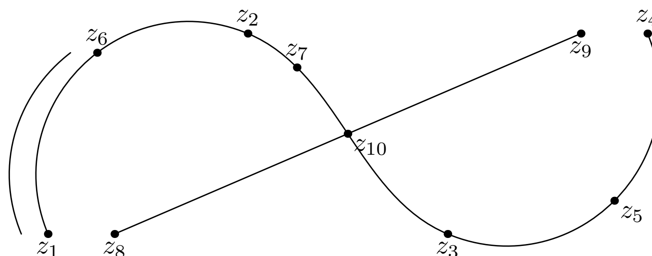


Figure 2: Parametric curves

We illustrate concepts related to parametric curves using Figure 2. First we define four key points and draw the path, `a`, through them.

```
z1 = (0,0);
z2 = (75,75);
z3 = (150,0);
z4 = (225,75);
path a;
a = z1 ..  z2 ..  z3 ..  z4;
draw a;
```

The path has a *length* associated with it:

```
show length a;                                   >> 3
```

It is convenient to think of the path being drawn by a point which moves along it, starting at time 0 and ending at time 3. The "times" of the control points are 0 for `z1`, 1, for `z2`, 2 for `z3`, and 3 for `z2`. The expression `point...of` finds the point on a path corresponding to a particular time (note the position of `z5` in the diagram):

```
z5 = point 2.5 of a;
```

We can find the direction of the path, expressed as a tangent vector, at any time.

```
show direction 0 of a;                           >> (-19.5705,47.24738)
```

If we prefer, we can work with the arclength instead of the time:

```
lena = arclength a;
show lena;                                       >> 378.67822
t1 = arctime lena/5 of a;
show t1;                                         >> 0.56165
z6 = point t1 of a;
```

The expression `subpath` (t_1, t_2) `of` p returns the part of the path p between times t_1 and t_2. The effect of

```
draw subpath (0,t1) of a shifted (-10,0);
```

can be seen in Figure 2 as the short arc on the left.

We can find the time on a path when the direction first achieves a particular value:

```
t2 = directiontime (1,-1) of a;
z7 = point t2 of a;                               >> 1.21132
show t2;
```

The straight line in Figure 2 is drawn as path `b` with the commands:

```
z8 = (25,0);
z9 = (200,75);
path b;
b = z8 -- z9;
draw b;
```

The point of intersection of paths `a` and `b` is given by:

```
z10 = a intersectionpoint b;
```

and the command

```
show a intersectiontimes b;                       >> (1.5,0.5)
```

shows that the intersection occurs at time 1.5 for path `a` and at time 0.5 for path `b`. If the paths `a` and `b` do not intersect, the result is `(-1,-1)`.

The expression

```
a cutbefore b
```

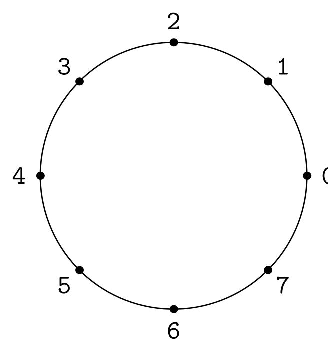
yields the part of path `a` from `z1` to `z10` and the expression

```
a cutafter b
```

yields the part of path `a` from `z10` to `z4`.

A circle drawn by MetaPost has eight control points (§4.3 : 23):

```
diam =100;
rad = 0.5*diam+8;
path circ;
circ = fullcircle scaled diam;
draw circ;
for i = 0 upto 7:
   dotlabel("", point i of circ);
   ang := 45 * i;
   label(decimal i, (rad*cosd(ang), rad*sind(ang)));
endfor;
```



77

Summary of path expressions:

Path expressions yielding `boolean`:

$$\texttt{path}\,p \triangleq p \text{ is a path}$$
$$\texttt{cycle}\,p \triangleq p \text{ is a cycle (closed path)}$$

Path expressions yielding `numerics`:

$$\texttt{length}\,a \triangleq \text{the value of } t \text{ at the end of } a$$
$$\texttt{arctime}\,\ell\ \texttt{of}\ a \triangleq \text{the value of } t \text{ at which the length of } a \text{ is } \ell$$
$$\texttt{directiontime}\,d\ \texttt{of}\ a \triangleq \text{the value of } t \text{ at which } a \text{ first has the direction } d$$
$$\texttt{arclength}\,a \triangleq \text{the total arc length of } a$$

Path expressions yielding `pairs`:

$$\texttt{point}\,t\ \texttt{of}\ a \triangleq \text{the position of } a \text{ at time } t$$
$$\texttt{direction}\,t\ \texttt{of}\ a \triangleq \text{the direction of } a \text{ at time } t$$
$$\texttt{directionpoint}\,d\ \texttt{of}\ a \triangleq \text{the first point at which the direction of } a \text{ is } d$$
$$a\ \texttt{intersectionpoint}\ b \triangleq \text{the point at which } a \text{ intersects } b$$
$$a\ \texttt{intersectiontimes}\ b \triangleq \text{the pair } (t_a, t_b) \text{ of intersection times}$$

Path expressions yielding `paths`:

$$\texttt{fullcircle} \triangleq \text{the unit circle (origin at centre)}$$
$$\texttt{unitsquare} \triangleq \text{the unit square (size } 1 \times 1\text{, origin at lower left corner)}$$
$$\texttt{bbox}\,a \triangleq \text{the bounding box of path } a$$
$$\texttt{reverse}\,a \triangleq \text{the path } a \text{ drawn backwards}$$
$$\texttt{subpath}\,(t_1, t_2)\ \texttt{of}\ a \triangleq \text{the part of } a \text{ for which } t_1 \leq t \leq t_2$$

$$a\ \texttt{\&}\ b \triangleq \text{the concatenation of paths } a \text{ and } b$$
$$a\ \texttt{cutbefore}\ b \triangleq \text{the part of } a \text{ before its intersection with } b$$
$$a\ \texttt{cutafter}\ b \triangleq \text{the part of } a \text{ after its intersection with } b$$

4.6 Path Constructors

The following functions and macros construct paths:

$$\texttt{reverse}\,p \triangleq \text{the path } p \text{ with its direction reversed}$$
$$\texttt{unitsquare} \triangleq \texttt{(0,0) --(1,0) --(1,1) --(0,1) -- cycle}$$
$$\texttt{bbox}\,p \triangleq \texttt{llcorner}\ p\ \texttt{--}\ \texttt{lrcorner}\ p\ \texttt{--}\ \texttt{urcorner}\ p\ \texttt{--}\ \texttt{ulcorner}\ p\ \texttt{--}\ \texttt{cycle}$$

The effect of `reverse` is defined by

$$\texttt{point}\ t\ \texttt{of}\ \texttt{reverse}\ p \equiv \texttt{point}\ (\texttt{length}\ p - t)\ \texttt{of}\ p$$

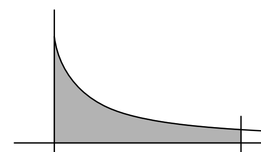
The `unitsquare` can be transformed (§3.7 : 17) to produce an arbitrary parallelogram. The macro `bbox`, applied to a picture *p*, returns the bounding box of *p*.

```
picture p;
p = btex $\spadesuit \heartsuit \diamondsuit \clubsuit$ etex;
draw p;
draw bbox p;
```

\# 19

The command `buildcycle` is especially useful for boundaries of shaded areas (§5.1.5 : 31). Given several paths, `buildcycle` tries to piece them together to form an enclosed figure, which is returned as the result of the expression. Draw the bounding path, if required, *after* shading the interior area with `filldraw`.

```
path p[];
p1 = (10,-5) -- (10,50);
p2 = (-5,0) -- (90,0);
p3 = (80,-5) -- (80,10);
p4 = (10,40) for i = 2 upto 9:
    ..  (10*i, 40/i) endfor;
p5 = buildcycle(p1, p2, p3, p4);
filldraw p5 withcolor 0.7 white;
draw p1; draw p2; draw p3; draw p4;
```



\# 41

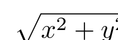
5 Commands

5.1 Drawing Commands

5.1.1 `btex` and `verbatimtex`

The command `btex` ⟨*text*⟩ `etex` invokes TEX to process ⟨*text*⟩ and returns the result as a picture:

```
draw btex $\sqrt{x^2+y^2}$ etex;
```



\# 17

Commands between `btex` and `etex` are written in TEX, *not* LATEX, and the text is set in HR mode. For example, `$$ ... $$` does not work, but `$\displaystyle ... $` usually gives the right effect. For more elaborate effects, use the TEX package (§7.3 : 43).

Text is set in the default font, which is usually `cmr10`. The default font can be changed by assigning to the internal variable `defaultfont`. The size of the text is determined by the default scale, which is 1. The scale can be changed by assigning to the variable `defaultscale`.

The command `verbatimtex` ⟨*text*⟩ `etex` is similar, but does not produce a picture. It is used to configure TEX for subsequent processing. For example, the following commands define the font commands `\bkm` to be 11pt Bookman and `lit` to be 10pt Typewriter:

```
verbatimtex
   \font\bkm = pbkli scaled 1100
   \font\lit = cmtex10 scaled 1000
etex
```

Font	Name	Sizes: $s =$
Bold	cmb⟨s⟩	10
Bold extended	cmbx⟨s⟩	5, 6, 7, 8, 9, 10, 12
Italic	cmti⟨s⟩	7, 8, 9, 10, 12
Roman	cmr⟨s⟩	5, 6, 7, 8, 9, 10, 12, 17
Sans serif	cmss⟨s⟩	8, 9, 10, 12, 17
Sans serif italic	cmssi⟨s⟩	8, 9, 10, 12, 17
Slanted	cmsl⟨s⟩	8, 9, 10, 12
Small caps	cmcsc⟨s⟩	10
Typewriter	cmtt⟨s⟩	8, 9, 10, 12
Typewriter italic	cmitt⟨s⟩	10
Typewriter slanted	cmsltt⟨s⟩	10

Figure 3: LaTeX fonts

These fonts can then be used in a `btex` command:

```
label.bot(btex\lit server etex, 0.5[a.e, b.w]);
```

Figure 3 shows some of the standard LaTeX fonts.

The command `verbatimtex` can be used to create an environment in which LaTeX can be used in `btex` commands. The following MetaPost file creates a LaTeX table:

```
verbatimtex
%&latex
\documentclass{article}
\begin{document}
etex

beginfig(1);
label(btex \begin{tabular}{c} First\\Second \end{tabular} etex, origin);
endfig;

end
```

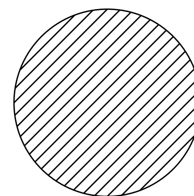
5.1.2 `clip`

The command

`clip` p `to` b

draws all parts of the picture p that lie within the boundary b. The picture p must be given as a picture variable, not a picture expression. The argument b is a path.

```
for i = -50 step 5 until 50:
    draw (i,-50) -- (i+100,50);
endfor;
path c;
c = fullcircle scaled 70 shifted (50,0);
clip currentpicture to c;
draw c;
```



54

5.1.3 draw

The command

`draw` p

draws the path or picture defined by the picture expression p. A picture can be erased by drawing it with the background colour:

`undraw` $p \triangleq$ `draw` p `withcolor background`

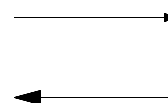
5.1.4 drawarrow

The following commands draw paths with arrows. The argument of each of these commands is a path, not a picture.

`drawarrow` p	:	draw path p with an arrow at the end
`drawarrow reverse` p	:	draw path p with an arrow at the start
`drawdblarrow` p	:	draw path p with an arrow at both ends

The size of the arrow head is determined by `ahlength` and `ahangle`. These internal variables have default values `ahlength` $= 4$ and `ahangle` $= 45°$, respectively. The code in the following example draws an arrow with default values and another arrow with length 10 and angle 30°.

```
begingroup;
    drawarrow (0,30) -- (60,30);
    interim ahlength := 10;
    interim ahangle := 30;
    drawarrow reverse ((0,0) -- (60,0));
endgroup;
```



65

5.1.5 `fill`

The following commands "paint" areas of the figure. The path must be a cycle (§4.1 : 22), otherwise the paint may leak out.

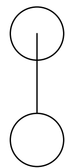
`fill` p	:	paint the area enclosed by path p with black
`fill` p `withcolor` c	:	paint the area enclosed by path p with colour c

There are some useful macros that use `fill`:

`filldraw` p	$\triangleq$	`draw (fill` p`)`
`filldraw` p `withcolor` c	$\triangleq$	`draw (fill` p `withcolor` c`)`
`unfill` p	$\triangleq$	`fill` p `withcolor background`

Filling can be used to avoid tricky coding with `cutbefore` and `cutafter`. In this example, `filldraw` erases the part of the line that would be inside the circle at `z2`.

```
z1 = (0,20); z2 = (0,-20);
draw z1--z2;
filldraw fullcircle scaled 20 shifted z2 withcolor white;
draw fullcircle scaled 20 shifted z1;
draw fullcircle scaled 20 shifted z2;
```



37

5.1.6 `label`

The command

`label` ⟨*suffix*⟩ `(` ⟨*picture expression*⟩ `,` ⟨*pair*⟩ `)`

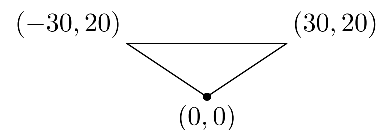
writes a label. ⟨*picture*⟩ describes the label to be written. It is usually a string (§3.2 : 10) or `btex ... etex` (§5.1.1 : 28). The ⟨*pair*⟩ determines the position of the label. The ⟨*suffix*⟩ may be omitted; if it is present, it must be one of

`.lft   .rt   .top   .bot   .ulft   .urt   .llft   .lrt`

and it defines the position of the label with respect to ⟨*pair*⟩.

The command `dotlabel` is similar to `label` but draws a large dot at the labelled point.

```
draw (0,0) -- (30,20) -- (-30,20) -- cycle;
label.ulft(btex $(-30,20)$ etex, (-30,20));
label.urt(btex $(30,20)$ etex, (30,20));
dotlabel.bot(btex $(0,0)$ etex, (0,0));
```



1

The command `dotlabels` provides an abbreviation that can be used when all of the points to be labelled have the form `z`⟨*suffix*⟩ and the suffixes are suitable for use as labels.

```
z.alpha = (0,0);
z.beta = (20,20);
z.gamma = (40,0);
dotlabels.top(alpha, beta, gamma);
```

beta
alpha gamma
18

The command `thelabel` is also similar to `label` but returns the result as a picture without drawing it. Thus:

$$\texttt{label}(p, z) \quad \triangleq \quad \texttt{draw thelabel}(p, z)$$

5.2 Non-drawing Commands

5.2.1 drawoptions

The command `drawoptions(`⟨*text*⟩`)` adds the options specified by ⟨*text*⟩ to all drawing commends until it is cancelled by the command `drawoption()`. The options allowed in ⟨*text*⟩ include `dashed` (§4.2 : 22), `withcolor` (§3.3 : 12), and `withpen` (§3.6 : 15). For example:

```
drawoptions(withcolor blue);
drawboxed(fred);
drawoptions();
```

The options in the scope of `drawoptions` affect only the commands for which they "make sense". For example, if a picture is drawn with `drawoptions(dashed evenly)`, all of its paths will be drawn with dashes, but its labels will not be affected.

5.2.2 filenametemplate

By default, MetaPost reads from a file called ⟨*jobname*⟩`.mp` and writes the output created by `beginfig(`n`) ... endfig` to `jobname.`n. The name of the output file can be changed by including a command of this form near the beginning of the input file:

`filenametemplate` ⟨*format string*⟩ `;`

The resulting file name is the ⟨*format string*⟩ after escape sequences have been processed. Figure 4 lists the escape codes that MetaPost recognizes.

`%%`	Percent sign
`%j`	Job name
`%`δ`c`	The argument of `beginfig`
`%`δ`y`	Year
`%`δ`m`	Month
`%`δ`d`	Day
`%`δ`H`	Hour
`%`δ`M`	Minute

Figure 4: Escape sequences for `filenametemplate`. The symbol δ stands for an optional decimal digit $(0, 1, 2, \ldots, 9)$ that sets the size of the string.

Suppose that the input file is `doc.mp`. By default, MetaPost behaves as if

```
filenametemplate "%j.%c";
```

has been executed and generates figure files called `doc.1`, `doc.2`, etc. If the MetaPost program included the command

```
filenametemplate "doc-%4y-%3c.mps";
```

then the output file names would be `doc-2007-001.mps`, `doc-2007-002.mps`, etc. This is useful for version of LaTeX that do not recognize numbers as file extensions.

5.2.3 `for`

The MetaPost loop has the general form:

```
for ⟨name⟩ = ⟨list⟩:
    ⟨seq⟩
endfor
```

Although *list* is not a type of MetaPost, there are expressions that generate lists (§3.9 : 21). For example:

```
for n = 1 upto 10:
    show x;
endfor
```

in which `upto` is an abbreviation, as is `downto`:

$$\begin{array}{rrcl} & \texttt{upto} & \triangleq & \texttt{step 1 until} \\ \text{and} & \texttt{downto} & \triangleq & \texttt{step -1 until} \end{array}$$

The numerical values do not have to be integers:

```
for x = 0.5 step 0.1 until 1.5:
    show x;
endfor;
```

```
>> 0
>> 0.1
>> 0.20001
>> 0.30002
....
>> 0.90005
```

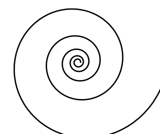
A list of expressions is accepted:

```
for e = 1, 256.1, "done":
    show e;
endfor;
```

```
>> 1
>> 256.1
>> "done"
```

MetaPost commands can be nested in fairly arbitrary ways. In the following example, the first point of a path is drawn in the usual way, but the remaining points are generated by a `for` loop. Note that there is no semicolon at the end of the loop body: a semicolon at this position would violate the syntax of `draw`. The semicolon after `endfor` serves to terminate the `draw` command.

```
draw (1,0)
for i := 0 upto 180:
..  (mexp(5i)*cosd(10i), mexp(5i)*sind(10i))
endfor;
```



79

There are not many MetaPost expressions that yield lists of tokens: one of them is `scantokens` (§3.9 : 21). Consequently, `for` and `scantokens` are often used together (§9.7 : 55).

If there are no natural loop variables, use

forever: ⟨*seq*⟩ exitif ⟨*pred*⟩; ⟨*seq*⟩ endfor

or

forever: ⟨*seq*⟩ exitunless ⟨*pred*⟩; ⟨*seq*⟩ endfor

A `for` loop can be used to obtain the parts of a picture. The effect of

for *c* within ⟨*picture expression*⟩ : ⟨*loop*⟩ endfor

is to bind each component of the picture in turn to *c* and then execute the text in ⟨*loop*⟩. Various predicates and selectors may be used in ⟨*loop*⟩, including `stroked`, `filled`, `textual`, `clipped`, `bounded`, `pathpart`, `penpart`, and `dashpart`. Figure 5 provides a simple example.

Like its relatives TEX and metafont, MetaPost is more like a macroprocessor than a formal language. This means that it is often possible to arbitrary chunks of text in a `for` loop, not just complete expressions. The shading example on page 28 illustrates this kind of usage.

5.2.4 `if`

Tests can be used in conditional statements which have the general form

if ⟨*pred*⟩: ⟨*seq*⟩
else: ⟨*seq*⟩
fi

Compound conditional statements use `elseif`:

if ⟨*pred*⟩: ⟨*seq*⟩
elseif ⟨*pred*⟩: ⟨*seq*⟩
else ⟨*seq*⟩
fi

For example (§7.1 : 39):

```
if pos = "top":
    label.top(title, name.n);
elseif pos = "bottom":
    label.bot(title, name.s);
fi;
```

```
draw (0,0) -- (100,0);
filldraw fullcircle scaled 50;
label.bot("line", (50,0));
picture p;
p = currentpicture;
show "p has " & decimal(length(p))
    & components.";

n = 0;
string msg;
for i within p:
    n := n + 1;
    msg := "Component " &
        decimal n & " is";
    if stroked i:
        msg := msg & " stroked";
    fi;
    if filled i:
        msg := msg & " filled";
    fi;
    if textual i:
        msg := msg & " textual";
    fi;
    show msg;
endfor;
```

```
>> "p has 3 components."
>> "Component 1 is stroked"
>> "Component 2 is filled"
>> "Component 3 is textual"
```

Figure 5: Using `for...within` to obtain a partial analysis of a picture.

5.2.5 `message`

The command `message` takes a `string` expression as its argument and displays the string on the console.

5.2.6 `readfrom`

The effect of

`readfrom` ⟨*file name*⟩

is to read one line from the named file and return the line as a string. If the file cannot be read, or if end of file has been reached, the string returned is `EOF`, which contains just the null character (i.e., `EOF` = `"\0"`).

If `readfrom` ⟨*file name*⟩ is executed again after it has returned `EOF`, the file is read again from the beginning.

See (§9.7 : 55) for an example of the use of `readfrom`. It is also possible to write to a file (§5.2.9 : 36).

5.2.7 `save`

The command **save**, followed by a list of variable names, saves the values of these variables and restores them again at the end of the scope, effectively making these variables local to the current scope. See also (§2.11 : 7) and (§6 : 36).

5.2.8 `show` and friends

The command

`show` e

writes `>>` followed by the value of the expression e to standard output. If the expression cannot be expressed in a simple textual form, MetaPost writes the type instead.

```
show (0,0) -- (10,0);                          >> path
```

There are variants of **show** that are used less often but can be handy for debugging:

`showdependencies`	$\triangleq$	the known dependencies for a set of equations
`showtoken`	$\triangleq$	the parameters and replacement text of a macro
`showvariable`	$\triangleq$	the properties associated with a name

The variable **showstopping** controls the behaviour of MetaPost after it has executed a **show** command. If **showstopping** > 0, MetaPost pauses after each **show** command.

5.2.9 `write`

The effect of the command

`write` ⟨*string expression*⟩ `to` ⟨*file name*⟩

is to write one line to the named file. If the file is not already open, MetaPost opens it before writing. The file is closed when the MetaPost program terminates, or explicitly by executing

`write EOF to` ⟨*file name*⟩

It is also possible to read from a file: (§5.2.6 : 35).

6 Macros

MetaPost provides many ways of defining macros. Here we describe just the most basic and useful techniques.

6.1 `def` macros

Simple macros have the form

def ⟨*name*⟩ = ⟨*seq*⟩ enddef

Simple macros with parameters have the form

def ⟨*name*⟩ (⟨*parameters*⟩) = ⟨*seq*⟩ enddef

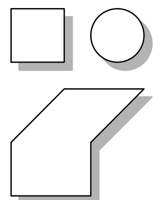
Following the macro definition, each occurrence of ⟨*name*⟩ is replaced by ⟨*seq*⟩, with parameters replaced by arguments when applicable. If the sequence ⟨*seq*⟩ ends with an expression, the value of that expression becomes the value of the macro. Note that there is no semicolon after the expression:

```
def hyp(expr a, b) =
    sqrt(a*a + b*b)
enddef;

show hyp(3,4);
```

```
>> 5
```

We can obtain a simple three-dimensional effect by adding a shadow to an outline. If the light source is above and to the left, the shadow should be below and to the right. The macro `shadow` gives a shadow to any path that can be filled — that is, to any closed path.

```
def shadow(expr p) =
    filldraw p shifted (3,-3) withcolor 0.7 white;
    filldraw p withcolor white;
    draw p;
enddef;

shadow(unitsquare scaled 20);
shadow(fullcircle scaled 20 shifted (40,0));
path p;
p = (0,0)--(0,20)--(20,40)--(50,40)--
    (30,20)--(30,0)--cycle;
shadow(p shifted (70,-20));
```



46

There are three kinds of macro parameter, distinguished by writing one of the keywords `expr`, `suffix`, or `text` before the parameter list.

- An `expr` parameter can be used as a variable in the body of the macro. Its value can be constrained by equations but cannot be re-assigned. The following macro says that `a`, `b`, `c`, and `d` are symmetrically placed around the origin but their separation is not known until `z1` is defined:

```
def vx(expr a, b, c, d) =
    a + c = (0,0);
    b + d = (0,0);
    xpart a = xpart d;                          >> (1,1)
    ypart a = ypart b;                          >> (-1,1)
enddef;                                         >> (-1,-1)
                                                >> (1,-1)
vx(z1,z2,z3,z4);
z1 = (1,1);
show z1; show z2; show z3; show z4;
```

- A `suffix` parameter is bound to an initializing expression but may also be used as a variable name. The macro `showfirst` would be illegal with an `expr` parameter, but is allowed with a `suffix` parameter:

  ```
  def showfirst(suffix a) = show a[0]; enddef;
  ```

- A `text` parameter is bound to an arbitrary sequence of tokens. When its name is encountered in the body, MetaPost simply substitutes the corresponding argument and parses the result. For example, if we define

  ```
  def doIt(text cmd) = cmd (0,0)--(5,5); enddef;
  ```

  then `doIt(draw)` draws the path but `doIt(show)` simply displays it.

If a macro has two or more different kinds of parameters, there must be a separate parameter list for each kind:

```
def tricky(expr a, b)(suffix p) = ...  enddef;
```

However, the corresponding invocation has just a single list of arguments:

```
tricky(2.3, xpart q, z);
```

6.2 `vardef` and other macro forms

A macro definition may be introduced by `vardef` instead of `def`. The most useful difference is that the body of a `vardef` macro is a group. A macro such as `area` can be used in the same way as a function in other languages:

```
vardef area(expr a, b, c) =
    save s;
    s := 0.5*(a+b+c);
    sqrt(s*(s-a)*(s-b)*(s-c))
enddef;

show area(3,4,5);                               >> 6
```

The name of a `vardef` macro may be a compound containing suffixes. Within the macro, `@` is the last token of the macro call and `#@` is everything that comes before the last token. Macros defined with `vardef` can be used to define special behaviour for particular variable names. For example, the special relationship between `x`, `y`, and `z` is a consequence of the macro definition

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

There are other forms of macro definition for which a full description goes beyond the scope of this manual. For example, macros can be used to define unary and binary operators. After defining

```
vardef neg primary x = -x enddef;
```

we can use `neg` to negate numeric values and pairs. For example,

```
neg((3,4))                                                    >> (-3,-4)
```

Similarly, after defining a midpoint operator

```
primarydef p mp q = 0.5[p,q] enddef;
```

we can write

```
z1 = (0,0);
z2 = (0,40);
draw z1 -- z2;
label.rt("mp", z1 mp z2);
```

mp

42

7 Macro Packages

Many MetaPost macro packages have been written, by Hobby and others. In this section, we describe a few of them. To use a package, include this command at the beginning of your MetaPost program:

`input` ⟨*package name*⟩

7.1 Boxes

There is a standard macro package, called `boxes`, for drawing boxes. To use it, include this statement at the beginning of your MetaPost program:

```
input boxes
```

The boxes produced by `boxes` have square corners. To get boxes with both square corners and rounded corners, use

```
input rboxes
```

7.1.1 Creating and Drawing Boxes

To create a box, write

> `boxit .` ⟨*box name*⟩ `(` ⟨*picture expression*⟩ `)`

The package `rboxes` provides the command `rboxit` as well. It works in the same way as `boxit`, but produces boxes with rounded corners.

To draw boxes, write

> `drawboxed (` $b_1, b_2, \ldots, b_n$ `)`

where $b_1, b_2, \ldots, b_n$ is a list of box names. The result of drawing a box is shown in Figure 6, in which the shaded inner rectangle corresponds to the picture and the outer rectangle is the box that is drawn. The box can be sized and positioned by applying the suffixes in Figure 6 to the box name. For example, if the box name is `bx`, then `bx.c` is its centre, `bx.ne` is its top right corner, and so on.

The values of `c`, `dx`, and `dy` are left unspecified. If MetaPost can infer their values from constraints on the positioning points, it does so. If not, it gives them the default values

$$\begin{aligned} \texttt{c} &= (0,0) \\ \texttt{dx} &= \texttt{defaultdx} \\ \texttt{dy} &= \texttt{defaultdy} \end{aligned}$$

The values of `defaultdx` and `defaultdy` can be changed by assignment. For example:

```
defaultdx := 20;
defaultdy := 15;
```

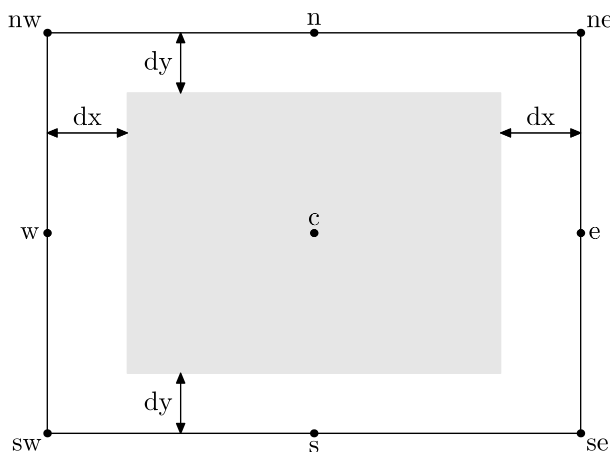


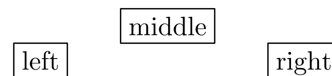
Figure 6: Box variables

The command `drawboxed` actually draws the boxes. Consequently, the best way to use boxes is to follow this sequence:

1. Use `boxit` to create the boxes
2. Write equations that specify positional relations between boxes

3. Use `drawboxed` to draw the boxes

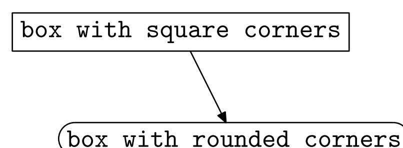
```
boxit.bl("left");
boxit.bm("middle");
boxit.br("right");
bm.sw - bl.ne = (20,0);
br.nw - bm.se = (20,0);
drawboxed(bl,bm,br);
```



30

This example is similar, but illustrates the use of the package `rboxes`. The macro `cuta` is defined at the end of this section.

```
boxit.sb("box with square corners");
rboxit.rb("box with rounded corners");
rb.c = sb.c + (100,0);
drawboxed(sb, rb);
cuta(sb,rb);
```

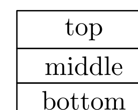


75

7.1.2 Positioning Boxes

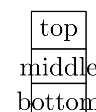
MetaPost provides another, shorter way of positioning boxes relative to one another. The macro `boxjoin` takes as argument a list of equations, separated by semicolons, describing the relationship between boxes `a` and `b`. The equations can be used to constrain both the positions and the sizes of boxes.

```
boxjoin(a.sw = b.nw; a.se = b.ne);
boxit.bt("top");
bt.ne - bt.nw = (50,0);
boxit.bm("middle");
boxit.bb("bottom");
drawboxed(bt,bm,bb);
```



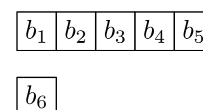
31

The equation `bt.ne - bt.nw = (50,0)` sets the width for the first box and `boxjoin` ensures that the other boxes have the same width. If it is omitted, the default size of the top box is used as the width:



The constraints imposed by `boxjoin` can be removed either by calling `boxjoin` with different equations or an empty argument.

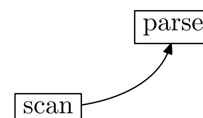
```
boxjoin(a.e = b.w);
boxit.b1(btex $b_1$ etex); boxit.b2(btex $b_2$ etex);
boxit.b3(btex $b_3$ etex); boxit.b4(btex $b_4$ etex);
boxit.b5(btex $b_5$ etex);
boxjoin();
boxit.b6(btex $b_6$ etex); b6.n = b1.s - (whatever,10);
drawboxed(b1,b2,b3,b4,b5,b6);
```



36

The path surrounding the box named b is `bpath` b. This is useful when boxes are joined by arrows.

```
boxjoin(a.e = b.w - (20,30));
boxit.scan("scan");
boxit.parse("parse");
drawboxed(scan, parse);
drawarrow scan.c{dir 0} .. {dir 90}parse.c
    cutbefore bpath scan
    cutafter bpath parse;
```



33

In fact, this pattern occurs so often that it is a good idea to define a macro for it. To join two boxes `b1` and `b2` with an arrow, write `cuta(b1, b2)`, where:

```
vardef cuta(suffix a, b) =
    drawarrow a.c -- b.c cutbefore bpath.a cutafter bpath.b;
enddef;
```

7.1.3 Oval Boxes

The macro `circleit` is similar to `boxit` except that the picture argument is enclosed in an oval rather than a rectangle. The corner points are not defined, but the other points correspond to `boxit`, as shown in Figure 7. Use `drawboxed` to draw the picture produced by `circleit` (not "drawcircled").

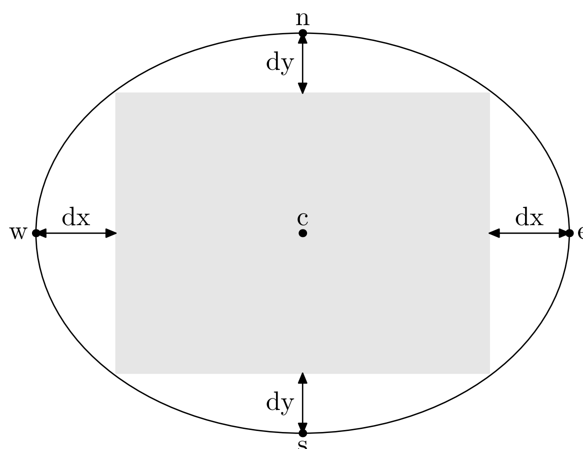
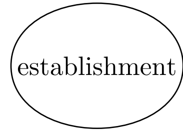


Figure 7: Oval variables

The path `bpath` c created by `circleit` is a circle unless at least one c.`dx`, c.`dy`, or c.`dx` $-$ c.`dy` is known. If any of these values are known, the path is an oval that contains the picture with a safety margin defined by `circmargin`, a length with default value 2 bp.

Assigning to `defaultdx` and `defaultdy`, as explained above, does *not* work for `circleit`. However, their values can be assigned explicitly, as in this example:

```
circleit.wd("establishment");
wd.dy = 20;
drawboxed(wd);
```



35

Other commands associated with boxes include:

`drawboxes`(...)	:	draw the outlines of the listed boxes but not their contents
`drawunboxed`(...)	:	draw the contents of the listed boxes but not their outlines
`pic` b	:	return the contents of the box b without its outline

Thus `drawunboxed(b1,b2)` is an abbreviation for

```
draw pic b1;
draw pic b2;
```

7.2 Graphs

John Hobby has written a graph package for MetaPost. A complete description of this package is beyond our scope, but we have space for a simple example. The file `hist.txt` contains the results of a simulated experiment in which a coin is tossed 50 times and the number of 'heads' are recorded. Each line of the file consists of two integers, as in the following extract:

```
....
23 96237
24 107870
25 112401
26 107747
27 96453
28 78692
....
```

Before drawing graphs, the graph macros must be read from `graph.mp`. The following MetaPost code is all that is needed to obtain Figure 8.

```
input graph

beginfig(1);
    draw begingraph(4in,3in);
    gdraw("hist.txt");
    endgraph;
endfig;
```

7.3 TEX

MetaPost does not process text between `btex` and `etex`. For example, if you write `btex n etex`, MetaPost will typeset the character "n", not the value of the variable `n`. The package `TEX.mp` provides an operator `TEX` that overcomes this problem. The expression

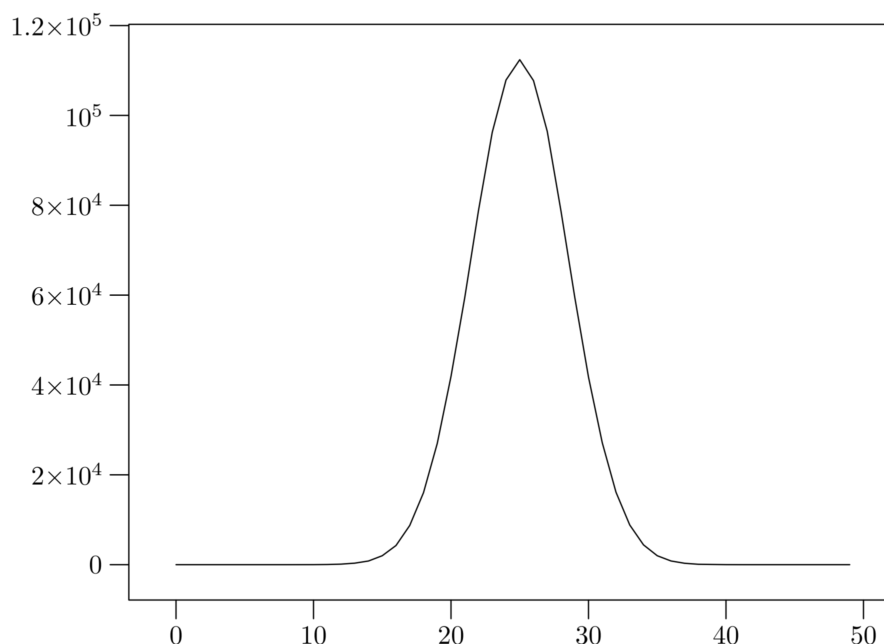


Figure 8: Simulated coin tossing

```
TEX("$X_{" & decimal(n) & "}$")
```

returns a picture that can be used, for example, in a `label` command. For $n = 0, 1, 2, \ldots$, the pictures will be $X_0, X_1, X_2, \ldots$. The `&` in this commend is the METAPOST concatenation operator, *not* TEX's alignment character. If the value of n is 35, for example, the command above constructs the string `$X_{35}$` and sends it to TEX.

The package also provides two further commands:

`TEXPRE(`s`)`	defines text that will be passed to TEX before each `TEX` command
`TEXPOST(`s`)`	defines text that will be passed to TEX after each `TEX` command

For example, you can use LATEX in labels by including these two commands at the beginning of your METAPOST program (`char(10)` generates a line break):

```
TEXPRE("%&latex" & char(10) & "\documentclass{article}\begin{document}");
TEXPOST("\end{document}");
```

7.4 MetaUML

The package `MetaUML.mp` extends METAPOST so that it can draw UML diagrams. This package is included with MIKTEX and is easy to find at SourceForge.

The good news is that MetaUML draws very nice UML diagrams when given correct input. The bad news is that rather small errors in the input tend to produce numerous error messages and may even cause METAPOST to loop.

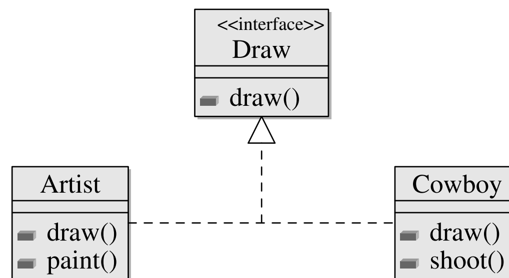


Figure 9: The UML diagram generated from the code in Figure 10

```
input metauml;
input TEX;

beginfig(82);
save art,cow,ifa;
Class.art("Artist")()("+draw()", "+paint()");
Class.cow("Cowboy")()("+draw()", "+shoot()");
Class.ifa("Draw")()("+draw()");
classStereotype.ifa("<<interface>>");

cow.w = art.e + (100,0);
z0 = 0.5[art.e,cow.w];
ifa.s = z0 + (0,40);

drawObjects(art,cow,ifa);

draw art.e -- cow.w dashed evenly;
z1 = ifa.s - (0,10);
draw z0 -- z1 dashed evenly;
link(inheritance)(z1--ifa.s);
endfig;
```

Figure 10: Using `MetaUML`

Figure 9 shows a diagram drawn by MetaUML and Figure 10 shows the code used to produce it. Note that the package `TEX.mp` must be loaded, as well as `metauml.mp`.

8 Debugging

It may seem odd to talk about "debugging" diagrams, but METAPOST is a programming language. The advantages of the programming language approach should be obvious from earlier parts of this manual; the disadvantages become clear when you actually try to use METAPOST.

As a simple illustration, suppose that you accidentally write "btext" instead of "btex". If you have written a few hundred lines of code, MetaPost's response may be quite worrying:

```
D:\Pubs\MetaPost>mp figs
(figs.mpCreating figs.mpx...
makempx: mpto failed on D:\Pubs\MetaPost\figs.mp.
mp: The operation failed for some reason.
```

As with other kinds of programming, the best approach to MetaPost is to proceed slowly and carefully with frequent tests. When MetaPost does fail, there is a good chance that the error is in the last few lines that you have written.

Here are a few additional tips for getting errant MetaPost programs to work:

- Use `show` and its friends (§5.2.8 : 36) to check that variables have the values that you expect.
- Use commands like `dotlabel("z5", z5)` to check that points are where you think they are. Sometimes, it is even quicker to use `dotlabels` (§5.1.6 : 31).
- When MetaPost fails, it often generates hundreds of lines of diagnostics. Sometimes, it is easy to see what went wrong. If not, it is often quicker to identify the line of code that MetaPost is complaining about, and to examine that line very carefully, than it is to try and understand the diagnostic.
- Errors such as "inconsistent equation" may occur if you use the same variable names in different figures. You can avoid this problem by using `save` (§5.2.7 : 36) for the variables of each figure.

9 Examples

In the last section of this manual, we give some complete examples that illustrate the practical use of MetaPost.

9.1 Euler Integration

The goal of this example is to obtain a figure like Figure 11 illustrating the error, e, of first-order Euler integration. If f' is the derivative of f:

$$f(x + \Delta x) \approx f(x) + f'(x)\,\Delta x$$

The first step is to draw a suitable path. A Bézier curve is the simplest to generate with MetaPost:

```
path c;
c = (0,0)dir 0 ..  (150,100);
draw c;
```

Next, we choose a point on the path, P_1:

```
t := 0.4;
```

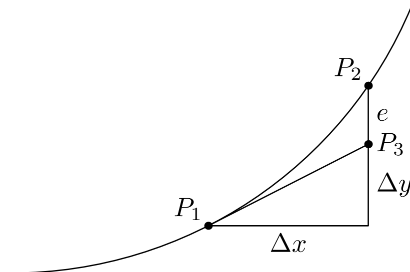


Figure 11: Euler Integration

```
pair P[];
P1 = point t of c;
```

The value of Δx is arbitrary, but we should choose a value large enough to make the diagram readable. We can then compute Δy using the direction (i.e., slope) of the path at t.

```
dx := 60;
(dx, dy) = whatever * direction t of c;
```

With Δx and Δy known, we can locate the other points that we need. P_4 is not labelled in Figure 11; it is the third vertex of the triangle with hypotenuse P_1P_3.

```
P4 = P1 + (dx, 0);
P3 = P4 + (0, dy);
P2 = c intersectionpoint (P4 -- P4+(0,100));
```

All that remains is to draw the lines and label the points and lengths:

```
draw P1--P4--P2;
draw P1--P3;
dotlabel.ulft(btex $P_1$ etex, P1);
dotlabel.ulft(btex $P_2$ etex, P2);
dotlabel.rt(btex $P_3$ etex, P3);
label.bot(btex $\Delta x$ etex, 0.5[P1,P4]);
label.rt(btex $\Delta y$ etex, 0.5[P4,P3]);
label.rt(btex $e$ etex, 0.5[P3,P2]);
```

9.2 The Lorentz Transformation

The Lorentz transformation for inertial frames with relative velocity v is

$$\begin{aligned} x' &= \gamma x + \gamma\beta ct \\ ct' &= \gamma\beta x + \gamma ct \end{aligned}$$

where c is the velocity of light, $\beta = v/c$, and $\gamma = 1/\sqrt{1 - v^2/c^2}$. Assuming $c = 1$, $v = 0.5$, writing `b` for β, `g` for γ, and including a translation of 200 units, we can express the transformation in METAPOST as:

```
b = 0.5;
```

```
g = sqrt(1 - b * b);
transform t;
xpart t = 200; ypart t = 0;
xxpart t = g; yxpart t = g * b;
xypart t = g * b; yypart t = g;
```

We illustrate the scenario shown on the left of Figure 12): a light flashes at a, is reflected by mirrors at b and b', and returns to the original point, displaced in time, at a'. We need eleven points for the diagram; Figure 13 shows the numbering scheme. The scale is determined by two constants, s and d:

```
s = 10;
d = 80;
```

It is easiest to define the x coordinates and the y coordinates separately:

```
x1 = x2 = x3 = 0;
x4 = x5 = x6 = x7 = x1 + d;
x8 = x9 = x10 = x4 + d;
x11 = x8 + s;
y1 = y4 = y8 = y11 = 0;
y5 = s;
y2 = y9 = y5 + d;
y6 = y2 + d;
y3 = y7 = y10 = y6 + s;
```

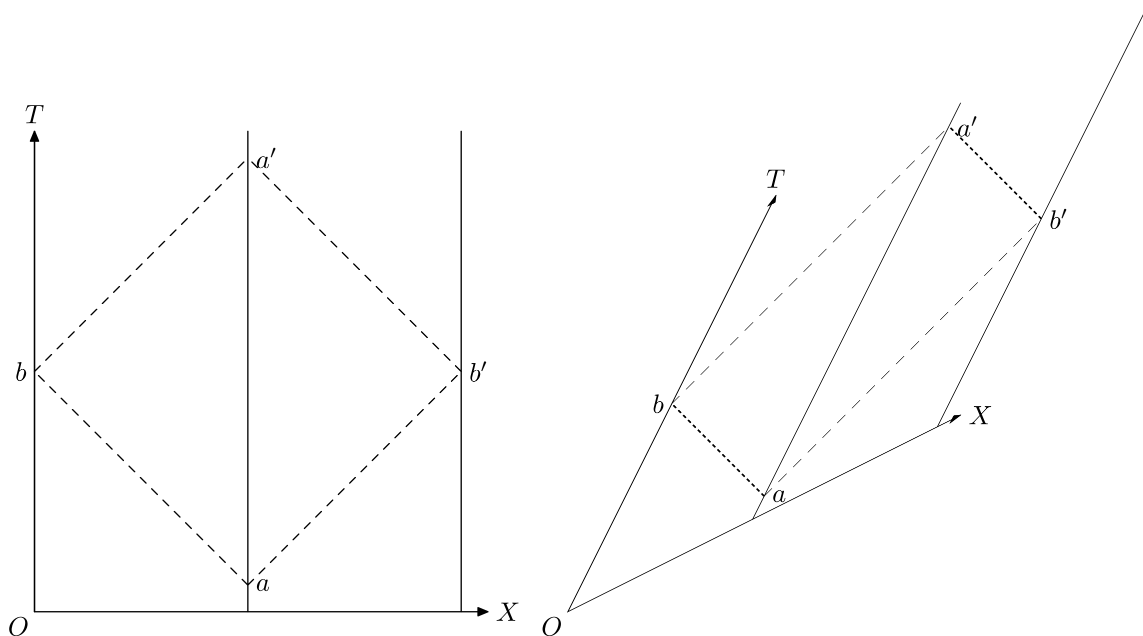


Figure 12: The Lorentz Transformation

We draw arrows for the axes:

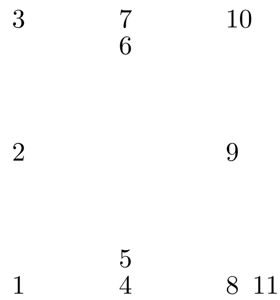


Figure 13: Point numbering used in Figure 12

```
drawarrow z1 -- z11;
drawarrow z1 -- z3;
```

solid lines for the time lines:

```
draw z1 -- z3;
draw z4 -- z7;
draw z8 -- z10;
```

and dashed lines for the light rays:

```
draw z5 -- z2 dashed evenly;
draw z2 -- z6 dashed evenly;
draw z5 -- z9 dashed evenly;
draw z9 -- z6 dashed evenly;
```

These commands have drawn into `currentpicture`. We store them in the picture variable `pic` and clear the current picture using the internal constant `nullpicture`:

```
picture pic;
pic := currentpicture;
currentpicture := nullpicture;
```

We next draw `p` twice, once as above, and once transformed:

```
draw pic;
draw pic transformed t;
```

We then label the original diagram:

```
label.llft(btex $O$ etex, z1);
label.rt(btex $X$ etex, z11);
label.top(btex $T$ etex, z3);
label.rt(btex $a$ etex, z5);
label.lft(btex $b$ etex, z2);
label.rt(btex $b'$ etex, z9);
label.rt(btex $c$ etex, z6);
```

and the transformed diagram:

```
label.llft(btex $O$ etex, z1 transformed t);
label.rt(btex $X$ etex, z11 transformed t);
label.top(btex $T$ etex, z3 transformed t);
label.rt(btex $a$ etex, z5 transformed t);
label.lft(btex $b$ etex, z2 transformed t);
label.rt(btex $b'$ etex, z9 transformed t);
label.rt(btex $c$ etex, z6 transformed t);
```

The result is Figure 12. Note that the events b and b' are simultaneous in the original diagram but not in the transformed diagram, and that the light rays (dashed lines) have the same slope (±1) in both diagrams.

This example would be slightly simpler if we included the labels in the first diagram. Then we would not have to write them out twice. Since MetaPost transforms *everything*, however, the labels would be distorted.

9.3 Dissipation

Figure 14 shows how the area between two trajectories of a dissipative system diminishes when measured at constant time intervals. The code for drawing it uses `point...of` to obtain points on the upper and lower curves, and `buildcycle` to construct the areas for shading.

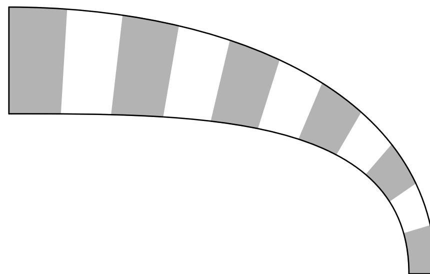


Figure 14: Area reduction in a dissipative system

Pick four points as end points for the curves:

```
z1 = (0,100); z2 = (160,0);
z3 = (0,60); z4 = (150,0);
```

Draw two curves, converging towards each other:

```
path p[];
p1 = z1{dir 0} .. {dir 270}z2;
p2 = z3{dir 0} .. {dir 270}z4;
```

Construct pairs of lines joining corresponding points on the curves. The lines are not drawn, but are used as arguments of `buildcycle` to create closed areas, which are shaded with `filldraw`.

```
for i = 0 step 2 until 11:
    p3 := point i/11 of p1 -- point i/11 of p2;
```

```
    p4 := point (i+1)/11 of p1 -- point (i+1)/11 of p2;
    filldraw buildcycle(p3, p1, p4, p2) withcolor 0.7 white;
endfor;
```

Finally draw the curves `p1` and `p2`, and a couple of lines, to bound the figure.

```
draw p1; draw p2;
draw z1 -- z3; draw z2 -- z4;
```

9.4 Desargues' Theorem

Figure 15 is obtained by the following construction.

- Choose a centre of projection, O.
- Draw a triangle, $A_1B_1C_1$.
- Draw the lines OA_1, OA_2, and OA_3.
- Draw another triangle, $A_2B_2C_2$, with the same centre of projection. That is A_2 lies on OA_1, etc.

  In METAPOST, this step is done by mediation. E.g., `a2 = 0.35[a1,o]`.
- Find the points of intersection of corresponding sides of the triangles. For example, B_1C_1 and B_2C_2 intersect at P_1.

  In METAPOST, this step is also done by mediation. E.g.:

  ```
  p1 = whatever[b1,c1] = whatever[b2,c2];
  ```
- Desargues' Theorem states that the three points of intersection, P_1, P_2, and P_3, are collinear.

Figure 16 shows the METAPOST code used the draw Figure 15. The label commands have been put at the end to ensure that the black dots are drawn after the coloured lines that go through them.

9.5 Cobweb Plots

Cobweb plots are used to illustrate the behaviour of dynamic systems. Figure 17 shows the result of computing 100 iterations of $x \to kx(1-x)$. Figure 18 shows the code used to create the diagram. The main points of interest is that the first `for` loop computes two points at each iteration, and that it is simpler to separate the computation of points from their display in the second `for` loop.

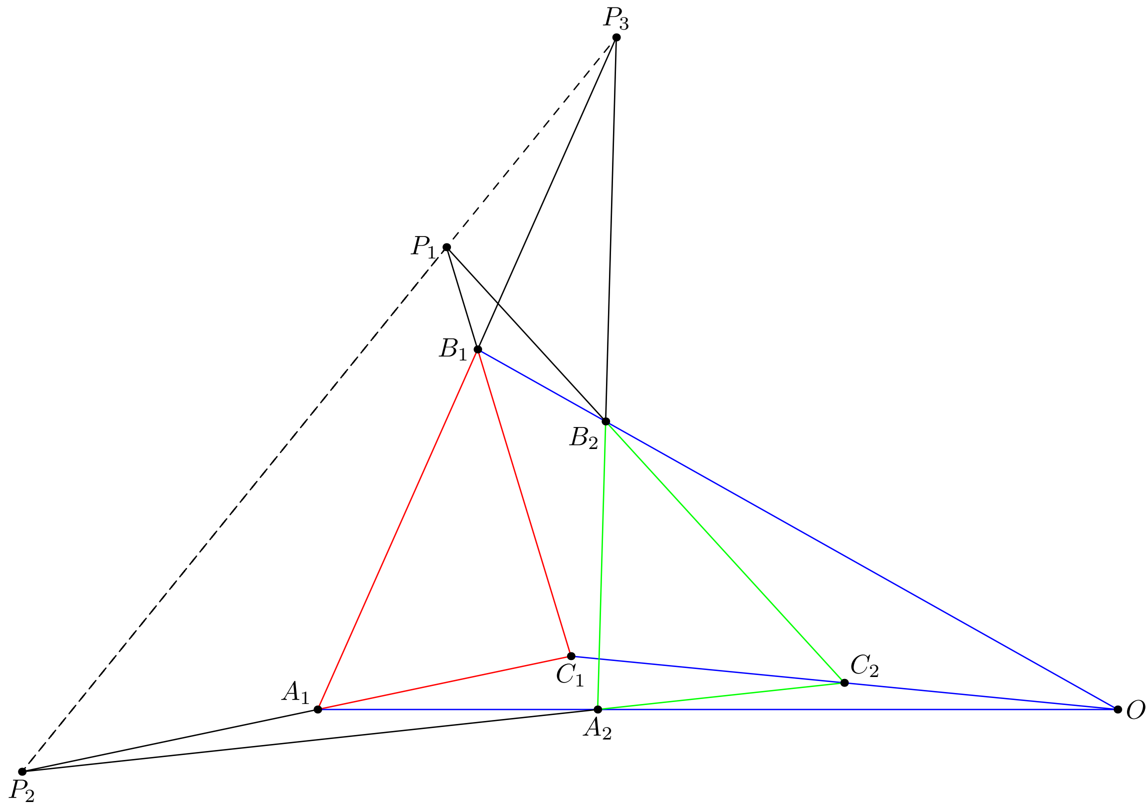


Figure 15: Desargue's Theorem

9.6 Three Dimensions

Although MetaPost is basically two-dimensional, it can be tricked into drawing perspective diagrams, giving the illusion of three dimensions. Many packages have been written for 3D MetaPost pictures (§10 : 56). One technique is to use the three components of the type `color` as XYZ coordinates. We give a simple example to illustrate the idea.

Assume that the viewpoint in 3D space is at $(0, 0, -d)$ and that we project onto the plane $z = 0$. This projection maps the 3D point (x, y, z) to the 2D point (tx, ty) where $t = d/(z + d)$. The macro `perspective` implements this transformation by converting a colour `c` $= (r, g, b)$ to a pair (tr, tg):

```
vardef perspective(expr c) =
    t := dist/(bluepart c + dist);
    (t * redpart c, t * greenpart c)
enddef;
```

We use the colour array `tp[]` to represent the vertexes of a cube with side 2 centered at the origin:

```
color tp[];
tp0 = (-1,-1,-1); tp4 = tp0 + (0,0,2);
```

```
pair a[],b[],c[],p[], o;
o = (300,0);

a1 = (0,0); b1 = (60,135); c1 = (95,20);

draw a1 -- b1 -- c1 -- cycle withcolor red;

draw a1 -- o withcolor blue;
draw b1 -- o withcolor blue;
draw c1 -- o withcolor blue;

a2 = 0.35[a1,o]; b2 = 0.2[b1,o]; c2 = 0.5[c1,o];

draw a2 -- b2 -- c2 -- cycle withcolor green;

p1 = whatever[b1,c1] = whatever[b2,c2];
p2 = whatever[c1,a1] = whatever[c2,a2];
p3 = whatever[a1,b1] = whatever[a2,b2];

draw b1 -- p1 -- b2;
draw a1 -- p2 -- a2;
draw b1 -- p3 -- b2;
draw p1 -- p2 -- p3 dashed evenly;

dotlabel.rt(btex $O$ etex, o);
dotlabel.ulft(btex $A_1$ etex, a1);
dotlabel.lft(btex $B_1$ etex, b1);
dotlabel.bot(btex $C_1$ etex, c1);

dotlabel.bot(btex $A_2$ etex, a2);
dotlabel.llft(btex $B_2$ etex, b2);
dotlabel.urt(btex $C_2$ etex, c2);

dotlabel.lft(btex $P_1$ etex, p1);
dotlabel.bot(btex $P_2$ etex, p2);
dotlabel.top(btex $P_3$ etex, p3);
```

Figure 16: Programming Desargues' Theorem

```
tp1 = ( 1,-1,-1); tp5 = tp1 + (0,0,2);
tp2 = ( 1, 1,-1); tp6 = tp2 + (0,0,2);
tp3 = (-1, 1,-1); tp7 = tp3 + (0,0,2);
```

Since MetaPost supports linear operations on colours, we can scale and translate the cube before using `perspective` with `dist` = 200 to transform the vertexes to the pair array `p[]`:

```
dist := 200;
```

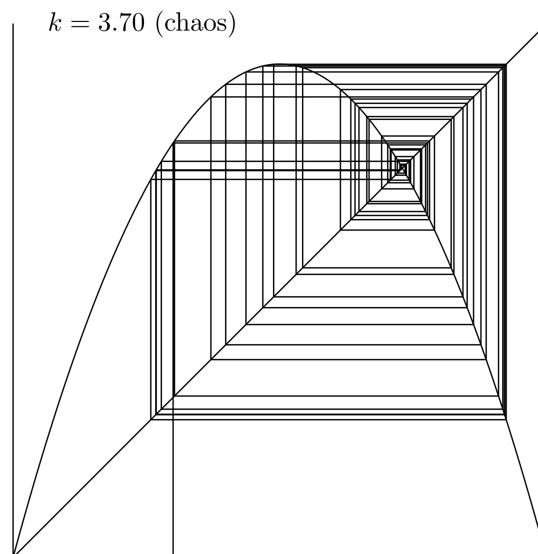


Figure 17: Cobweb plot for the logistic map

```
save scale, k, seed, xmax;
scale = 200; k = 3.70; seed = 0.3; iters = 100;
label.rt(btex $k=3.70$ (chaos) etex, (10, scale));

draw (0,0) -- (scale,0);
draw (0,0) -- (0,scale);
draw (0,0) -- (scale, scale);

draw (0,0) for x := 0.0 step 1/iters until 1.0:
  .. (scale * x, scale * k * x * (1-x))
endfor;

x0 = seed; y0 = 0;
for i := 1 step 2 until iters:
  x[i] := x[i-1];
  y[i] := k * x[i] * (1 - x[i]);
  y[i+1] := y[i];
  x[i+1] := y[i+1];
endfor;

draw scale * z0
  for i := 1 upto iters: -- scale * z[i]
endfor;
```

Figure 18: Coding the cobweb plot in Figure 17

```
pair p[];
for i = 0 upto 7:
    p[i] := perspective(40*tp[i]+(80,-70,30));
endfor;
```

Drawing the edges of the cube gives Figure 19:

```
draw p0--p1--p2--p3--cycle;
draw p4--p5--p6--p7--cycle;
draw p0--p4;
draw p1--p5;
draw p2--p6;
draw p3--p7;
```

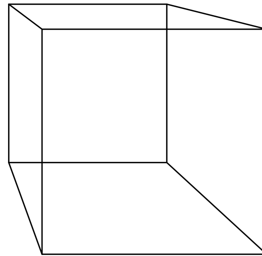


Figure 19: Perspective view of a cube

9.7 Reading a File

This example illustrates the use of `readfrom` and `scantokens`. Suppose that the file `"data.txt"` contains these two lines:

```
1, 2, x, "Go!"
"pi = ", 3.14159263
```

The code on the left produces the figure on the right:

```
boxjoin(a.sw = b.nw; a.se = b.ne);
picture b[];
i = 0;
x = 99;
forever:
    string line;
    line := readfrom "data.txt";
    exitif line = EOF;
    string token, desc;
    for token = scantokens line:
        if numeric token:
            desc := "Numeric:  " &
                decimal token;
        elseif string token:
            desc := "String:  " & token;
        fi;
        boxit.b[i](desc);
        b[i].e = b[i].w + (100,0);
        drawboxed(b[i]);
        i := i + 1;
    endfor;
endfor;
```

Numeric: 1
Numeric: 2
Numeric: 99
String: Go!
String: pi =
Numeric: 3.14159

72

The outer `forever` loop reads each line of the file, checking for end of file. The inner `for` loop uses `scantokens` to split the line into tokens. The type of each token is checked, using type names as predicates, and put into a descriptor, `desc`. Finally, each descriptor is drawn in a box.

Note that the third token in the file is `x` and it is displayed as `99`, the value of `x` in the program. Tokens in a file are considered to be strings only if they are quoted, like `"Go!"`.

10 Links

The TEX User's Group — probably all that you will need: http://www.tug.org/metapost.html

A FAQ from the UK: http://www.tex.ac.uk/cgi-bin/texfaq2html?label=MP

Urs Oswald's *Very Brief Tutorial*: http://www.ursoswald.ch/metapost/tutorial.html

MetaPost home page: http://ect.bell-labs.com/who/hobby/MetaPost.html

John Hobby's home page: http://plan9.bell-labs.com/who/hobby/

Marc van Dongen's contributions to MetaPost: http://www.cs.ucc.ie/ dongen/mpost/mpost.html

MetaFun by Hans Hagen: http://wiki.contextgarden.net/MetaFun

A WYSIWYG interface for MetaPost: http://w3.mecanica.upm.es/metapost/metagraf.php

A MetaPost generator for 3D: http://www.gnu.org/software/3dldf/LDF.html

Index

&, 11, 22, 27
**, 8
++, 8, 13
+-+, 8, 13
--, 22
---, 23
.., 23
..., 23
:=, 9
=, 9
[], 15
3D, 52

abs, 13, 15
affine transform, 17
ahangle, 30
ahlength, 30
and, 10
angle, 14
angle, 15, 19
anonymous variable, 9
arc, 23
arclength, 27
arctime, 9, 27
area
  shaded, 28
argument, 37
arithmetic, 13
array, 6
arrow, 30
assignment, 9

background, 12
bbox, 27
beginfig, 14, 32
beginfig, 2
begingroup, 7
beveled, 16
big points, 4
binary operator, 8
black, 12
blackpart, 12
blue, 12
bluepart, 12
boolean, 10
bot, 31
bounded, 21, 34
bounding box, 20
boxes, 39
boxit, 40
boxjoin, 41
bp (big point), 4
bpath, 42
btex, 28, 43
buildcycle, 28, 50
butt, 16

c, 40
ceiling, 13
center, 19
char, 11
circle, 23
circleit, 42
circmargin, 42
clip, 29
clipped, 21, 34
cm, 4
cmykcolor, 10, 12
color, 10, 12, 52
comments, 4
comparison, 10
concatenation
  of paths, 22
  of strings, 11
constants
  colours, 12
  numerics, 13
  pairs, 14
  strings, 11
control points, 24
conversion
  abs (pair to numeric), 15
  angle (pair to numeric), 15
  ceiling (numeric to integer), 13
  char (ASCII to string), 11
  decimal (numeric to string), 11
  dir (numeric to pair), 15
  floor (numeric to integer), 13
  round (numeric to integer), 13
  str (suffix to string), 11
cosd, 13

curl, 24
currentpicture, 5, 19, 49
curve, 23
cuta, 41, 42
cutafter, 27
cutbefore, 27
cyanpart, 12
cycle, 10, 22, 27

dashed, 22
dashed line, 22
dashpart, 21, 34
day, 13
decimal, 11, 13
decimal
  in LaTeX, 43
declaration, 6
def, 37
defaultdx, 40
defaultdy, 40
defaultfont, 11, 28
defaultpen, 15
defaultscale, 11, 28
difference of squares, 8, 13
dir, 15
direction, 15
  of curve, 24
  represented by pair, 14
direction, 9, 19, 27
directionpoint, 9, 27
directiontime, 9, 27
ditto, 11
div, 13
division, 13
dotlabel, 31
dotlabels, 31, 46
dotprod, 15
dotted line, 22
down, 14
downto, 21, 33
draw, 12, 30
drawarrow, 30
drawboxed, 40
drawboxes, 43
drawdblarrow, 30
drawoptions, 12, 16, 32
drawunboxed, 43
dx, 40
dy, 40

else, 34
elseif, 34
endfig, 2
endgroup, 7
EOF, 35, 36
epsilon, 13
equality, 9, 10
equation, 9
erase, 30
etex, 43
evenly, 22
exitif, 34
exitunless, 34
explicit declaration, 6
expr
  macro parameter, 37
expression, 8
  path, 27

false, 10
fi, 34
file, 2
  name, 32
  read, 35, 55
  write, 36
filenametemplate, 2, 32
fill, 12, 31
filldraw, 12, 28, 31
filled, 21, 34
floor, 13
font, 28, 29
for, 21, 33, 56
forever, 34, 56
fullcircle, 23, 27
functions
  colours, 12
  numerics, 13
  pairs, 15
  pictures, 19
  strings, 11

graphicx, 3
green, 12
greenpart, 12
group, 7

halfcircle, 23

`hyperref`, 3
hypotenuse, 8, 15

`identity`, 18
`if`, 34
`image`, 20
`in`, 4
`infinity`, 13
inflection points, 23
`infont`, 11
inner product, 15
`input`, 39
input file, 2
`interim`, 7
internal variable, 6
intersection, 26
`intersectionpoint`, 27
`intersectiontimes`, 27
`inverse`, 18

`known`, 10

`label`, 31, 44
LaTeX, 28
`left`, 14
length, 13
`length`, 11, 19, 27
`lft`, 31
line
  corner, 16
  end, 16
  straight, 22
`linecap`, 16
`linejoin`, 16
list, 21
`llcorner`, 19
`llft`, 31
local
  variable, 7
loop, 33
`lrcorner`, 19
`lrt`, 31

macro, 36
`magentapart`, 12
`makepath`, 15
`makepen`, 15
mediation, 14, 15
`message`, 35

`MetaUML`, 44
`mexp`, 13
`mitered`, 16
`miterlimit`, 16
`mlog`, 13
`mm`, 4
`mod`, 13
`month`, 13
`mproof`, 4
`mpversion`, 2, 11
multidimensional array, 6

name, 5
  local, 7
  of output file, 32
  of type, 6
  of variable, 5
negative coordinates, 5
`newinternal`, 6
nib, 15
normal, 19
  random number, 13
`normaldeviate`, 13
normalize, 15
`not`, 10
`nullpen`, 15
`nullpicture`, 19, 49
`numeric`, 10, 13

`odd`, 10, 13
`of`, 8
operator, 8
  assignment, 9
  comparison, 10
  difference of squares, 8
  equality, 9
  path, 22, 23
  sum of squares, 8
`or`, 10
`origin`, 14
output
  file name, 32
  to console, 35, 36
  to file, 2

painting, 31
`pair`, 10, 14
parameter, 37
  of macro, 37

path
    circle, 23
    dots and dashes, 22
    expression, 27
    of box, 42
    parametric, 25
    straight, 22
`path`, 10, 21, 27
`pathpart`, 21, 34
`pc`, 4
pen, 15
`pen`, 10
`pencircle`, 15
`penoffset`, 9
`penpart`, 21, 34
`pensquare`, 15
`pic`, 43
`pickup`, 15
picture, 19
`picture`, 10
point
    control, 24
    Postscript, 4
    printer's, 4
`point`, 9, 19, 27, 50
`postcontrol`, 9
precedence, 8
`precontrol`, 9
predicates, 10
primary, 8
`pt` (printer's point), 4

`quartercircle`, 23

random number, 13
`rboxes`, 39–41
`rboxit`, 40
read file, 35, 55
`readfrom`, 35, 55
`red`, 12
`redpart`, 12
`reflectedabout`, 17
remainder, 13
`reverse`, 23, 27
`rgbcolor`, 10, 12
`right`, 14
`rotated`, 17
`rotatedaround`, 17
`round`, 13, 15
`rounded`, 16
`rt`, 31

`save`, 7, 36
`scaled`, 17, 22
scaling, 4, 5
`scantokens`, 11, 21, 34, 55
scope, 7
secondary, 8
semicircle, 23
`setbounds`, 20
shading, 28, 31
shadow, 37
`shifted`, 17, 22
`show`, 36
`showdependencies`, 9, 36
`showstopping`, 36
`showtoken`, 36
`showvariable`, 36
`sind`, 13
`slanted`, 17
`sqrt`, 13
`squared`, 16
`step`, 21
`str`, 11
string, 10
`string`, 10, 11
`stroked`, 21, 34
`subpath`, 9, 23, 27
`substring`, 9, 11
suffix, 5
`suffix`
    macro parameter, 38
sum of squares, 8, 13

tag, 5
tangent, 19
`tension`, 24
tertiary, 8
`TEX`, 43
TeX, 28
`text`
    macro parameter, 38
`textual`, 21, 34
`thelabel`, 32
three dimensions, 52
`time`, 13
`top`, 31
transform, 17

`transform`, 10, 17
`transformed`, 17
`true`, 10

`ulcorner`, 19
`ulft`, 31
UML diagrams, 44
UMM, 1
unary operator, 8
undefine, 6
`undraw`, 30
`unfill`, 31
uniform random number, 13
`uniformdeviate`, 13
units, 4
`unitsquare`, 27
`unitvector`, 15
`unknown`, 10
`until`, 21
`up`, 14
`upto`, 21, 33
`urcorner`, 19
`urt`, 31

value
- colour, 12
- decimal, 11
- global, 7
- group, 7
- known, 10
- largest, 13
- smallest, 13
- unknown, 6, 10

`vardef`, 38
variable
- anonymous, 9
- declaration, 6
- internal, 6, 7
- local, 7
- name, 5

`verbatimtex`, 28
version, 2

`whatever`, 9
`white`, 12
`withcolor`, 12, 31
`withdots`, 22
`within`, 34
`withpen`, 17
write
- expression, 36
- string, 35
- to file, 36

`write`, 36

`x`, 6, 14, 38
`xpart`, 15, 18
`xscaled`, 16, 17
`xxpart`, 18
`xypart`, 18

`y`, 6, 14, 38
`year`, 13
`yellowpart`, 12
`ypart`, 15, 18
`yscaled`, 16, 17
`yxpart`, 18
`yypart`, 18

`z`, 6, 14, 38
`zscaled`, 17