November 1997

gamelan:

Graphical Analysis macros for the MetaPost Language

MANUAL

Wolfgang Kilian[1]

Theoretische Physik 1
Walter-Flex-Str. 3, Universität Siegen
D-57068 Siegen, Germany

PRELIMINARY DRAFT
December 7, 2011

ABSTRACT

gamelan is a package for data and function plotting within a LaTeX document. It is based on macros written in MetaPost, originally derived from John Hobby's 'graph.mp' macro package. This manual describes version 0.40.

[1] kilian@physik.uni-siegen.de

Contents

Chapter 1

The LaTeX interface

1.1 Using gamelan

gamelan consists of two parts: A LaTeX style file and a MetaPost macro package. The style file implements two environments at top level: `gmlpreamble` and `gmlfile`. The former is part of the LaTeX preamble; it is used to pass certain macro declarations down to gamelan. The `gmlfile` environment is part of the main document. It encloses the actual graphs to be drawn by gamelan.

Inside a `gmlfile` environment, four additional environments are defined: `gmlcode`, `gmlfigure`, `gmlgraph`, and `gmltex`. The text enclosed by these is not parsed by the LaTeX interpreter, but written verbatim to a file. This file serves as an input file to gamelan, which transforms it into `eps` (encapsulated postscript) figures. The figures are automatically inserted into the original document in an additional LaTeX run.

The document structure is depicted schematically in Fig.??. There is at most one `gmlpreamble`, but there may be several `gmlfile` environments, each one producing a separate input file for gamelan. Inside a `gmlfile`, the number of graphs and figures is unlimited.

To load the style file, put a corresponding `\usepackage` declaration in the document header:

```
\usepackage{gamelan}
```

Since gamelan uses the standard `verbatim` and `graphics` packages, those two need not be called separately if you need them; they are already present.

1.2 The `gmlfile` environment

`gmlfile` The `gmlfile` environment activates gamelan's functionality by defining

the figure-drawing environments, and by specifying where gamelan commands should be written to. It has an optional argument which specifies a filename:

```
\begin{gmlfile}      or
\begin{gmlfile}[⟨filename⟩]
```

If the optional argument is omitted, the ⟨*filename*⟩ is set to the filename of the main document, given by `\jobname`. Any code inside the graph-drawing environments will be written to the file with the specified name and extension `.mp`.

The `gmlfile` environment should be placed inside the document's `\begin{document}` ... `\end{document}` body. Apart from gamelan figures and graphs, there can be arbitrary text, figures, sections, chapters, etc., inside it. In a short document where only a single `gmlfile` environment is used, the `\begin{gmlfile}` command may immediately follow `\begin{document}`. However, in a long document it is more convenient to use several `gmlfile`s with different names and to restrict their scope to a particular section or figure.

1.2.1 Writing gamelan code

gamelan interprets code written in MetaPost language. This language is not intellegible to LaTeX. However, the environments described in this section provide an interface such that MetaPost code can inserted in a LaTeX document. On the other hand, gamelan will envoke LaTeX to format textual labels.

`gmlcode` The simplest environment is `gmlcode`:

```
\begin{gmlcode}
  ⟨gamelan declarations and commands⟩
\end{gmlcode}
```

This environment does not produce any graphical output; it encloses global declarations and similar things . For instance, you may wish to define global variables and macros:

```
\begin{gmlcode}
  scale:= 1000;
  vardef foo(expr x) = x + 42 enddef;
\end{gmlcode}
```

or a dataset is read in, to be used in several distinct graphs:

```
\begin{gmlcode}
  directory "data/";
```

```
  fromfile "run1.dat":
    table plot(run1)();
  endfrom
\end{gmlcode}
```

The gmlcode environment is also a way to try out some of the examples in this manual which do not involve graphical output:

```
\begin{gmlcode}
  a=4; show 3a;
\end{gmlcode}
```

The gmlcode environment must reside inside a gmlfile, so the latter example reads:

```
\documentclass{article}
\usepackage{gamelan}
\begin{document}
\begin{gmlfile}
\begin{gmlcode}
  a=4; show 3a;
\end{gmlcode}
\end{gmlfile}
\end{document}
```

If you type this code in your favorite editor, save it into a file named, e.g., foo.tex, and start LaTeX

```
> latex foo
```

a file foo.mp is created containing, among other stuff, the line enclosed between \begin{gmlcode} and \end{gmlcode}. Now write

```
> gml foo
```

and gamelan is started with the input file foo.mp, resulting in the output

The number 12 is the result of the show 3a; command. Clearly, neither graphics nor text has been generated. We will come to real figures in a minute.

1.2.2 Figures

`gmlfigure` The `gmlfigure` environment is a wrapper for gamelan code intended to generate graphical output:

```
\begin{gmlfigure}
  ⟨gamelan code⟩
\end{gmlfigure}
```

The gamelan code is written to file, to be executed by the interpreter, and the result transformed into an encapsulated Postscript file. Here is an example:

```
\documentclass{article}
\usepackage{gamelan}
\begin{document}
\begin{gmlfile}
\begin{gmlfigure}
  draw pentagram scaled 2cm;
\end{gmlfigure}
\end{gmlfile}
\end{document}
```

This file (named, again, `foo.tex`) should be run through LaTeX

```
> latex foo
```

LaTeX will create a file `foo.mp`, and it will complain that the figure `foo.1` has not been found. To generate this figure, run gamelan on `foo.mp`

```
> gml foo
```

with the screen output

Now, in a second LaTeX run the figure `foo.1` is included in the document:

```
> latex foo
```

and the result shows the desired pentagram

```
> xdvi foo
```

1.2.3 Graphs

gmlgraph Now that we are able to produce graphics, why is there a second environment for the same purpose? The reason is, that for the task of visualizing data, you need to specify the dimensions of the graph in advance: gamelan knows the size of `1cm` on paper, but it has to be told how to translate a temperature value of 35°F.

Thus, the `gmlgraph` environment has the same syntax as the familiar `picture`: The dimensions of the graph on paper are specified in round brackets, in terms of `\unitlength`. By default, this is equal to `1pt`; it is usually a good idea to set it explicitly immediately before the graph environment[1]

```
\documentclass{article}
\usepackage{gamelan}
\begin{document}
\begin{gmlfile}
\unitlength 1mm
\begin{gmlgraph}(60,30)
  draw plot((#0,#1), (#1,#2), (#2,#6),
            (#3,#1), (#4,#2), (#5,#1));
\end{gmlgraph}
\end{gmlfile}
\end{document}
```

If you run this example through LaTeX and gamelan

```
> latex foo; gml foo; latex foo
```

and try to view the result

```
> xdvi foo
```

the `PostScript` interpreter inside `xdvi` will probably crash. Similarly, you will not be able to view or print the EPS file `foo.1` by itself. The reason is missing font information: In order keep the size of the EPS files small, MetaPost tells the `PostScript` interpreter to take its font information from the main document, which does not yet exist in `PostScript` format. This is easily remedied by transforming the main document into `PostScript`

```
> dvips foo
```

and to display the result as usual

[1] Usually, one wraps additional `figure` and `center` environments, or the like, around a graph. In that case, `\unitlength` may be reset and you need the explicit declaration if you do not like to think in `pt`.

```
> ghostview foo
```

Now the graph, including axis labels, is visible, and can be printed.

This extra translation step has to be repeated anytime you want to have a look at the graphics. However, with this behavior the `.dvi` file is *really* device-independent and can be distributed, to be translated into `PostScript` on anybody's local machine. It is guaranteed that the fonts inside gamelan graphs match the text fonts in the same document, and are appropriate for the local printer if the local `dvips` driver has been set up correctly. This is not the case for self-contained EPS files.

1.2.4 LaTeX embedded in gamelan figures

For typesetting labels, gamelan calls LaTeX in the background. This is normally invisible to the user (except for some delay in processing the input file). However, since this background process is distinct from the LaTeX run which typesets the main document, there must be some means to communicate macros and style settings to the subprocess. This is illustrated in the following example:

```
\documentclass{article}
\usepackage{gamelan}
\begin{gmlpreamble}
  \usepackage{amsfonts}
\end{gmlpreamble}
\begin{document}
\begin{gmlfile}
\begin{gmltex}\large\end{gmltex}
\begin{gmlfigure}
  draw (-10,0)--(150,0) witharrow;
  draw (0,-10)--(0,50) witharrow;
  label(<<$\mathbb{C}$ (the complex plane)>>,
        (80,40));
\end{gmlfigure}
\end{gmlfile}
\end{document}
```

This has to be processed by LaTeX and gamelan, as usual. gamelan will take care of the embedded LaTeX sequence (the text enclosed between << and >>):

```
> latex foo; gml foo; latex foo; dvips foo
> ghostview foo
```

`gmlpreamble` Here, gamelan has to access the letter C in the `amsfonts` package. This is achieved by wrapping the corresponding `\usepackage` command in a `gmlpreamble` environment. The code enclosed in this environment is written verbatim to an auxiliary file with extension `.ltp` (`foo.ltp` in this case). This file is included at the end of the preamble both by the main document and by any LaTeX subprocess which is started by `gamelan`.

Several `gmlpreamble` environments may appear in the preamble of the main document. Their contents are concatenated and the resulting file is executed once `\begin{document}` is reached.

`gmltex` Finally, LaTeX declarations that do not belong to the preamble may be inserted in the current `gmlfile` via a `gmltex` environment. In the previous example, a `\large` declaration is inserted in this way which applies to all following labels.

1.3 Miscellaneous LaTeX commands

Apart from the environments described in the previous section, there are only a few additional LaTeX commands defined by gamelan. They provide an interface to certain gamelan declarations which are frequently encountered. Such declarations must be placed inside a `gmlfile` environment, before the `gmlfigure` or `gmlgraph` where their effect is needed.

From now on, we do not show the document header in the examples. It is understood that they are part of a LaTeX document, wrapped in an appropriate `gmlfile` environment.

1.3.1 Function definitions

`\gmlfunction` The `\gmlfunction` macro has three arguments: the function name, a dummy variable, and a gamelan floating-point expression which provides the function definition. The following example shows how to plot the function $f(x) = e^{-x/10}\cos x$ between $x = 0$ and $x = 10$:

```
\gmlfunction{f}{x}{exp(neg x over #10) times cos(x)}
\begin{gmlgraph}(150,80)
  fromfunction f(#0,#10): draw table plot(); endfrom
\end{gmlgraph}
```

1.3.2 Default pen scale and color

`\gmlpenscale` gamelan draws its lines and shapes with an imaginary circular pen of

diameter `0.5bp`. This can be locally reset by appropriate declarations or drawing options, but there is also a global declaration

```
\gmlpenscale{⟨pen diameter⟩}
```

which is in effect for all following `gmlfigure`s and `gmlgraph`s inside the current `gmlfile`. The pen diameter is usually a number (in `bp`, if not specified otherwise), but it may be any MetaPost expression which evaluates to a number.

`\gmlpencolor` Similarly, the default drawing color may be specified by `\gmlpencolor`, which has four possible forms:

```
\gmlpencolor{⟨R-value⟩,⟨G-value⟩,⟨B-value⟩}
\gmlpencolor{⟨color expression⟩}
\gmlpencolor{"rrr ggg bbb"}
\gmlpencolor{"RRGGBB"}
```

where the ⟨*R-value*⟩ etc. are numbers between 0 and 1 which make up a RGB tripel. In the second form, ⟨*color expression*⟩ is a MetaPost color expression, allowing for named colors like

```
\gmlpencolor{blue}
```

The string `"rrr ggg bbb"` consists of three-digit RGB values between 0 and 255, separated by single blanks. Finally, `"RRGGBB"` stands for a string of six hexadecimal digits. Thus, the color `chartreuse` may be defined as the current drawing color by one of these four equivalent commands:

```
\gmlpencolor{0.5, 1, 0}
\gmlpencolor{0.5red + green}
\gmlpencolor{"127 255 000}
\gmlpencolor{"7FFF00"}
```

1.3.3 Defining new colors

Internally, gamelan deals with colors by the usual RGB model: each color is represented by a tripel of numbers between 0 and 1. The three components represent the red, green, and blue saturation, respectively. Black is represented by `(0,0,0)`, white by `(1,1,1)`. Eight basic colors are predefined as variables:

`black`, `white`, `red`, `green`, `blue`, `cyan`, `magenta`, `yellow`

`\gmlcolor` Additional named colors can be defined by the `\gmlcolor` declaration. The syntax is analogous to `\gmlpencolor`:

```
\gmlcolor{⟨color name⟩}{⟨R-value⟩,⟨G-value⟩,⟨B-value⟩}
\gmlcolor{⟨color name⟩}{⟨color expression⟩}
\gmlcolor{⟨color name⟩}{"rrr ggg bbb"}
\gmlcolor{⟨color name⟩}{"RRGGBB"}
```

After a color has been defined, it may be used as the default drawing color

```
\gmlcolor{chartreuse}{0.5, 1, 0}
\gmlpencolor{chartreuse}
```

or, within a `gmlgraph` or `gmlfigure`, it can be used wherever a color is appropriate:

```
\begin{gmlfigure}
  fill fullcircle scaled 5mm withcolor chartreuse;
\end{gmlfigure}
```

1.3.4 Color databases

gamelan comes with two files containing predefined colors: `gmlcolors.tex` and `gmlextracolors.tex`. They contain a collection of `\gmlcolor` commands, based on the `rgb.txt` color list which is distributed as part of the X Window system. The color definitions may be copied into the document, or one may simply include them

```
\input gmlcolors
\input gmlextracolors
```

1.3.5 Expanded macros in gamelan code

`\gml` The environments discussed so far have one particular feature: LaTeX control sequences inside the gamelan code are not expanded when the main document is formatted. Usually, this is as desired, since LaTeX code in embedded labels should not be interpreted before gamelan has called its LaTeX subjob. However, in some cases one would like to expand LaTeX control sequences *before* code is written to file. This is achieved by using the `\gml` command. For example, let us introduce a string in gamelan which contains the current date:

```
\gml{string date; date="Today's date: \today";}
```

Now you can write

```
\begin{gmlcode}
  message date;
\end{gmlcode}
```

to display today's date online when the gamelan job is executed.

1.3.6 Grouping

In gamelan, all declarations are global by default. The graph-drawing environments `gmlfigure` and `gmlgraph` allow to make enclosed declarations local, if the corresponding tokens occur in their optional argument:

```
\begin{gmlfigure}[a,qqw]
  ⟨gamelan code⟩
\end{gmlfigure}
```

Here, any macros and variable names whose first token is either `a` or `qqw` will be local to this enviroment.

`gmlgroup` Furthermore, there is a generic environment

```
\begin{gmlgroup}
  ⟨LaTeX and gamelan code⟩
\begin{gmlgroup}
```

which generates a surrounding group for all gamelan code generated by enclosed `gmlfigure`, `gmlgraph`, and `gmlcode` environments *and* for all LaTeX text embedded in these graphs.

For an application, consider the `\gmlpenscale` and `\gmlpencolor` commands described above: They are automatically local, so if you write

```
\begin{gmlgroup}
\gmlpencolor{red}
\begin{gmlfigure}
  ⟨gamelan code⟩
\end{gmlfigure}
\end{gmlgroup}
```

the current figure is drawn with a red pen. After the closing `\end{gmlgroup}`, the pen color is black again (or whatever had been its previous value).

1.3.7 Turning gamelan on and off

`\gmlon` `\gmloff` In a long document containing many figures, it is time-consuming to run gamelan on the whole input file if only a few of its figures have been changed. Therefore, there are two switches

`\gmlon` and `\gmloff`

which can be put anywhere inside a `gmlfile` environment. When `\gmloff` is encountered, the gamelan interpreter is told to skip all following `gmlfigure`s and `gmlgraph`s. The environments `gmlcode` and `gmltex` are unaffected. `\gmlon` turns the interpreter on again. The generated EPS files, if present, will be included in any case.

Chapter 2

Drawing figures

2.1 Numeric values and variables

2.1.1 Numeric type

gamelan has a fixed-point number system inherited from METAFONT. In this scheme, any number is represented as an integer multiple of 1/65536. The range is limited between −16384 and +16384. If such numbers are to represent physical distances on paper (measured in `bp`), this covers ordinary paper sizes with a high accuracy. However, numerical values can represent other quantities as well. There is no distinct integer type since fixed-point numbers are representable exactly. In particular, in places where other programming languages accept integer values only — such as for loop counters and array subscripts — fractional values are allowed in gamelan.

Variables may consist of more than one token:

```
numeric a, foo.bar, xx yy zz;
```

Tokens (*suffixes*) may be separated by a whitespace and/or a single dot. To declare arrays, use empty square brackets:

```
numeric a[], b[][], c[]dd q[];
a3=1; b0[-1]; c[3.1]dd q0;
```

Numeric subscripts (which indicate array elements) may be arbitrarily mixed with other suffixes. Subscripts may be negative or even fractional. For a positive subscript the square brackets are optional. (The explicit `numeric` declaration is unnecessary for numerical values, but is mandatory for other types.)

2.1.2 Equations and assignment

Variables of numeric type need not be declared explicitly. Nevertheless, an explicit declaration is possible:

```
numeric a, b, x;
a=4b; x=3; b=2x;
show a,b,x;
>> 24
>> 6
>> 3
```

The usual = sign may be used to assign variables which do not have a value yet. However, as this example demonstrates, the = operator implies equality, not assignment. If necessary, MetaPost automatically solves a system of linear equations among variables to determine their actual values. Once this is possible, an additional equation defining the same variable would be either redundant or inconsistent:

```
b=b+1; show b;
! Inconsistent equation (off by 1).
```

To *assign* a variable, i.e., to remove its previous value (if any), one has to either redeclare the variable, or use the := operator:

```
a:=a+1; show a;
>> 25
```

The := assignment will remove any previous value while an = assignment (i.e., equation) produces an error if a value is accidentally overwritten. So, in most cases the use of either form is partly a matter of taste. However, if one deals with point locations on paper, the implicit equation solver is a powerful tool to describe a graph in a rather concise and abstract way.

The equation-solving mechanism requires variables to allow for an *unknown* state. Variables of any type are `unknown` before their value is fully determined (and, if they have not been declared otherwise, they are assumed `unknown numeric`). The same is true for array elements — therefore, arrays never have a definite `size` or `length`.

This feature is useful in other places: For instance, the command which determines the extension of a graph in data coordinates may be given unknown arguments; the appropriate values will then be determined automatically:

```
graphrange (#0,#0), (#10,??);
```

`??` There is one variable which is *always* `unknown`. This is the `??` macro (a.k.a. `whatever`) which has been used in the example above.

2.1.3 Fixed-point arithmetics

The basic arithmetic operations as well as some elementary transcendental functions are available (see Table **??**):

```
show 3+4, 5.3*(2.1-3), 5/2, 6**3, sqrt 2, sind 45;
>> 7
>> -4.76997
>> 2.5
>> 216.00002
>> 1.41422
>> 0.7071
```

Clearly, the accuracy of calculations cannot be overwhelming, given the fact that 1/65536 is the smallest unit. For the applications MetaPost has originally been designed for, this is sufficient. Unfortunately, for the purpose of data handling, it is not. Therefore, floating numbers have been introduced in the gamelan package (see below).

2.1.4 Units

To describe distances on paper, a number of constants are predefined[1].

```
bp  pt  in  mm  cm
```

So, knowing that a multiplication sign is optional if a number is immediately followed by a variable, distances may be expressed in standard notation:

```
show 2cm, 1.4mm, 1in;
>> 56.6929
>> 3.96848
>> 72
```

The default unit is `1bp`, equal to 1/72 of an inch.

2.1.5 Floating-point numbers

The range and accuracy of fixed-point numbers is limited. However, at least the first limitation may be circumvented by doing calculations with logarithms instead. This is the approach introduced in John Hobby's `graph.mp` macro package which has been followed by gamelan.

[1] To be exact, they are just ordinary variables, but it is not a good idea to change their values. Any named variable may serve as a distance unit.

Consequently, in gamelanfloating numbers do not make up a distinct type, but they are *emulated* in this way on top of fixed-point numerics[2]. In gamelan code, a floating-point number is indicated by a preceding hashmark:

```
numeric a,b,c,d;
a = #3;  b = #.0000005;  c = #6.023e23;
showfloat a,b,c;
>> "3.00000026E+00"
>> "5.00000047E-07"
>> "6.02300063E+23"
```

`showfloat` The output representation of a floating-point number is a string. They are also input as strings (otherwise, MetaPost would not be able to parse scientific notation such as `#6.023e23`). Fortunately, you do not have to type quotation marks here: A preprocessor will insert them for you. The `#` sign is mandatory, however.

`\#` The operator which transforms a fixed-point number into a floating-point one is the same `#` sign:

```
numeric a,b;  a = 3;  b = #a;
show a;  showfloat b;
>> 3
>> "3.00000026E+00" )
```

`\$` `\$#$` The reverse transformation into a number or a string is done by `#$` resp. `$#$`. These two are seldom needed. `$#$` is the operation applied by `showfloat`. Note that `#$` may overflow, because there are floating-point numbers which are not representable in the fixed-point system.

Floating-point numbers are distinguishable from fixed-point ones by context only. Therefore, overloading or arithmetic operators is not possible, and a completely distinct system of arithmetic operations had to be defined for them. The interpreter can't help if the two systems are unintentionally mixed. However, this is not critical as long as three rules are obeyed:

1. Every floating-point constant appearing in gamelan code must be preceded by a `#` sign.

2. For integer/fixed-point numbers and variables use the ordinary operators `+`, `-`, `*`, `/`, etc. For floating-point numbers write full operator names instead: `plus`, `minus`, `times`, `over` etc.

[2]A better solution would be to introduce floating-point numerics in the C sources of MetaPost itself. This, however, has not (yet?) been done.

3. For debugging, use `show` to display fixed-point numbers, and `showfloat` for floating-point numbers.

The necessity of writing long operator names makes calculational code less readable. However, gamelan should not be used for complicated calculations anyway[3]. The complete list of operators is found in Tab.**??**.

Needless to say, automatically solving linear equations is not implemented for floating-point numbers. If you try it, the result will be just garbage.

2.2 Other types

2.2.1 Pair type

Two numeric values may be grouped into a pair:

```
pair a; a=(3,2*5);
```

Variables of type `pair` must be declared before an assignment can be made. Pair arrays can be defined as for numerics (or any other type):

```
pair b[]x[];  b3x[-.22] = 33;
```

Locations on paper are described by fixed-point pair values:

```
draw (0,0)--(10cm,5cm);
```

To access the components of a pair value, say `p`, use the `xpart` and `ypart` operators:

```
pair p; p=(3,2);
show xpart p, ypart p;
>> 3
>> 2
```

Pair values can mathematically be treated as vectors. There is scalar multiplication and division (with the `*` optional in unambiguous cases), addition and subtraction, and a dot product:

```
show 3(2,1), (4,6)/2, (2,1)+(1,-1), (2,1) dotprod (1,-1);
>> (6,3)
>> (2,3)
>> (3,0)
>> 1
```

[3]Because they are inefficient and imprecise. Nevertheless, anything is possible in principle.

The square bracket notation describes a location like *one-third on the straight line from `(2,1)` to `(1,0)`*:

```
show 1/3[(2,1), (1,0)];
>> (1.66667,0.66667)
```

Square brackets are also used for selecting array elements. However, since the dimensions in a multi-dimensional array are *not* separated by commas (instead of `a[3,2]`, you must write `a[3][2]` or `a3 2`), there is no ambiguity here.

The arithmetic operations described above are defined for fixed-point pairs only. Floating-point pairs are useful, nevertheless; they represent data values:

```
draw plot((#0,#0), (#10,#5));
```

For floating-point pairs vector operations are not implemented: calculations must be done on the x and y components separately.

2.2.2 Path type

An ordered set of points (i.e., pair values) defines a path. In order for gamelan to know whether line segment are straight, joined smoothly, or otherwise, a connection method has to be specified. For instance, one can define a path consisting of straight line segments

```
path p; p = (0,0)--(10,5)--(4,6);
```

or Bézier curves (cubic splines)

```
path p; p = (0,0)..(10,5)..(4,6);
```

The connection is defined by the two-character tokens `--` and `..`, respectively. These connectors can be mixed:

```
path p; p = (0,0)..(10,5)--(4,6);
```

Furthermore, we should mention how to declare cyclic paths

```
path q; q = (0,0)--(10,5)--(4,6)--cycle;
```

and how to extract the path length and to select a point within a path

```
show length p,  point 1 of p,  point 1.5 of p;
>> 2
>> (10,5)
>> (7,5.50002)
```

Point indices count beginning from zero, and they need not be integer. For cyclic paths, they are cyclic, so negative values count from the end:

```
show length q,  point -1 of q;
>> 3
>> (4,6)
```

A subpath is selected as follows:

```
tracingonline:=1; show subpath (0, 1.5) of p;
>> Path at line 0:
(0,0)..controls (3.33333,1.66667) and (6.66667,3.33333)
 ..(10,5)..controls (9,5.16667) and (8,5.33334)
 ..(7,5.50002)
```

Setting `tracingonline:=1` allows paths to be displayed on screen (otherwise, the `show` command would write them to the logfile only). Here, the points are listed together with (invisible) Bézier control points which internally define the path shape. There are many ways to access these explicitly, and to get finer control on the path shape; see the MetaPost and METAFONT manuals.

2.2.3 Predefined paths

The collection of predefined paths is helpful for designing figures and symbols. There are general macros for polygons, star-shaped paths, and crosses:

`polygon` ⟨n⟩, `polygram` ⟨n⟩, and `polycross` ⟨n⟩

where ⟨n⟩ is an integer which specifies the number of edges. A couple of special cases are named

`triagon` (= `triangle`), `tetragon` (= `diamond`), `pentagon`, `hexagon`
`triagram`, `tetragram`, `pentagram`, `hexagram`
`triacross`, `tetracross` (= `cross`), `pentacross`, `hexacross`

Here, `triagon` is equivalent to `polygon 3`, `tetragon` means `polygon 4`, and so on. Finally, there are the obvious ones

`circle` and `square`

where a `circle` consists of eight points connected by Bézier curves. These paths are shown in Table **??**. Transformations (e.g., `scaled`, `xscaled`, `yscaled`, `rotated`) and path operations turn them into more general path shapes[4].

[4]For generic oval shapes, consider the `superellipse` command described in the MetaPost and METAFONT manuals.

2.2.4 Pen type

penscale By default, a virtual circular pen of diameter `0.5bp` is used for drawing lines. This can be changed by a declaration

`penscale` ⟨*pen-diameter*⟩`;`

withpenscale to any value. This declaration is local to the current figure. To affect the the current line only, append an option to the corresponding drawing command:

`draw` ⟨*object*⟩ `withpenscale` ⟨*pen-diameter*⟩`;`

withpen However, one can change not only the width, but also the *shape* of the virtual pen. See this example:

```
path p; p=(0,0){right}..(2cm,0.5cm)..(0,1cm)..{right}(2cm,1.5cm);
penscale 5mm;
draw p;
draw p shifted (3cm,0) withpen penrazor scaled 5mm;
draw p shifted (6cm,0) withpen penrazor rotated 45 scaled 5mm;
draw p shifted (9cm,0) withlinecap butt;
```

The same path is drawn in four different ways: First, using a thick, but otherwise ordinary, pen. For the next two images we have used a flat pen in two different orientations. (Note that the global `penscale` declaration applies only to the default pen.)

pickup A global change of pen shape and size is achieved by a `pickup` command, which is in effect for the rest of the current figure. The following two commands are equivalent:

```
penscale 5mm;
pickup pencircle scale 5mm;
```

withlinecap In the last image the virtual pen is circular again, but the line ends are cut off by a `withlinecap` option. To enforce this globally, set the `linecap` parameter explicitly: `linecap:=butt;` Alternatives are `butt`, `rounded` (default), and `squared`.

In fact, a pen is a type of variable which can be generated from any closed path. Let us draw the path again, now using a triangular pen:

```
pen tripen; tripen=makepen(triagon);
draw p withpen tripen scaled 5mm;
```

withlinejoin Finally, the appeareance of sharp edges is controlled by the `linejoin` parameter (globally) or the `withlinejoin` option (locally):

```
path p; p=(0,0)--(2cm,0.5cm)--(0,1cm);
penscale 5mm;
draw p withlinejoin mitered;
draw p shifted (4cm,0) withlinejoin rounded;
draw p shifted (8cm,0) withlinejoin beveled;
```

2.2.5 Color type

Colors are defined as RGB tripels. Internally, they are treated completely analogous to pair values, and they can be understood as points in a three-dimensional space. Operations that work on pair values also apply to colors, including addition, scalar multiplication, and even linear interpolation:

```
show red, .5red, red+blue, .5[red,white];
>> (1,0,0)
>> (0.5,0,0)
>> (1,0,1)
>> (1,0.5,0.5)
```

When rendering colors, negative components are mapped to 0, and component values greater than 1 are mapped back to 1.

Color variables are easily defined:

```
color c;  c=(.5,.1,.9);
```

withcolor The three components of a color value or variable, say `p`, are accessed by

`redpart p`, `greenpart p`, and `bluepart p`

The `withcolor` drawing option allows for any color expression as argument. Here, we take a predefined color:

```
fill square scaled 5mm withcolor red;
draw square scaled 8mm withpenscale 2 withcolor blue;
```

The predefined colors, as well as the LaTeX interface to gamelan colors, have been introduced above in Sec. 1.3.

2.2.6 Picture type

`image` Multiple graph elements can be collected into pictures, which by themselves can be stored in variables. The `image` macro is a wrapper for defining pictures:

```
path p; p = square scaled 1cm;
picture q; q = image(draw p; draw p rotated 45);
```

The result may be integrated into the final figure (which, incidentally, is a picture variable by itself, called `currentpicture`) by another `draw` command:

```
draw q;
```

`nullpicture` `addto` Operations that can be done with pictures include assignment and junction. The last line in the previous example is equivalent to

```
picture q;  q = nullpicture;
addto q also p;  addto q also p rotated 45;
```

Transformations such as `rotated` will be discussed below.

2.2.7 Transform type

gamelan is well prepared to apply affine transformations to any graphical object: pairs, paths, and pictures. These include shifts, reflections, rotations, rescalings, and more. They may be concatenated:

```
draw circle scaled 3mm;
draw circle scaled 5mm shifted (2cm,0);
draw square rotated 45;
picture p;
p=triangle scaled 5mm reflectedabout ((-1,0)--(1,0));
```

As a rule, in a drawing command transforms come before any drawing options (such as `withcolor`, etc.) since they modify the object *before* it is being drawn. If they are concatenated, they are applied from left to right.

Transforms may be stored in variables:

```
transform t; t=identity rotated 45 xscaled 2 yscaled 4;
draw q transformed t;
```

Here, the trivial transform `identity` serves as a starting point for defining the transform variable. Of course, another transform may act on `t` afterwards, and one could apply additional transformations in the drawing command.

2.2.8 String type

Strings are not very important in gamelan. Nevertheless, they are available, and some operations can be done with them:

```
string s,t,u;  s = "foo";  t = "bar"
u := s&t;
show u, length u, substring(1,4) of u;
>> "foobar"
>> 6
>> "oob"
```

As for paths, indices count beginning with zero. Think of the characters as corresponding to line segments, so substring indices correspond to the points in between.

Strings may be used to display messages:

```
message "Hello, world!"
```

More important, however, is their use as graph labels

```
label("foobar", (5cm,2cm));
```

They are typeset in a particular font `defaultfont` (a string variable), which is set to `"cmr10"`, `"cmr11"`, or `"cmr12"`, depending on the default font of the enclosing LaTeX document.

2.2.9 LaTeX strings

<< >> The form of labels that can be directly represented as strings is very limited, even if the character set is extended by additional symbols (such as greek letters, square root sign, etc.) Fortunately, gamelan has the full power of LaTeX at hand: Any text which is enclosed by << and >> signs is processed by LaTeX and transformed into a `picture` expression, before gamelan comes to see it. So, LaTeX labels can be assigned to a picture variable, or directly be integrated into the figure. See this example:

```
picture p;  p = <<$E = mc^2$>>;
for x=0 step 1/12 until 1:
  draw p colored ((1-x)*white) shifted ((4x, x**2)*5mm);
endfor
```

The LaTeX code is translated by a subprocess of gamelan, which acts as a preprocessor on the input file. To pass declarations, packages, and definitions to this subprocess, use the `gmltex` and `gmlpreamble` environments <<! described in the previous chapter. Alternatively, LaTeX code which does not produce output may be inserted into the gamelan text, wrapped into the brackets <<! and >>.

2.2.10 Boolean type

There are two boolean values

`false` and `true`

and a distinct variable type `boolean` with the usual operators:

```
boolean a,b;  a=true; b=false;
show a or b, not a and b;
>> true
>> false
```

The main use of these are control structures (see below).

2.3 Drawing commands and options

2.4 Shapes and boxes

2.5 Control structures

Chapter 3

Visualizing data

Chapter 4

Graph appearance

4.1 Scale

4.2 Labels

4.3 Legend

4.4 Axes

4.5 Frame