GLE Users Guide

David Parfitt, *david.parfitt@chch.ox.ac.uk* [1]

Oxford University Department of Physics

26th July 2003

Contents

1 Introduction

This document is intended to introduce the skills needed to produce technical diagrams and graphs using the graphics package GLE. The bulk of the text is a series of tutorials which illustrate many of the important commands and concepts. Readers who are already familiar with GLE 3.3 or 3.5 may wish to skip the tutorial section and use only the reference section in the appendix.

1.1 What is GLE?

GLE, or Graphics Language Editor, is a programming language whose command set is dedicated almost entirely to graphical output. Using this language it is possible to produce accurately drawn graphs and diagrams, with a high degree of control over the finished output. Like all other programming languages, it is necessary to learn some basic commands before very much can be achieved, and in this respect, GLE differs from more traditional methods of producing graphics. It lacks the immediate feedback of a point and click type interface, nor is it as easily understood.

The strength of GLE lies in its reproducibly and in its adaptability. Rather than a set of pixel values, a GLE diagram is stored as a series of vector drawing instructions. A line is stored as a line, a circle as a circle. The GLE package itself takes these instructions and converts them into a recognisable graphic. Many of the concepts from other programming languages can also be included in the code. Variables, subroutines, and programming loops can all greatly simplify the diagrams we wish to produce by allowing us to refer back to and reuse drawing instructions. We are also given a great deal of control over the finished output. There are very few complex commands in GLE; much of what we draw must be built up from a series of lines and other shapes. Although this can take a lot of time, the simplicity of the interface means that we are always in control of exactly what appears on screen.

GLE is a program written with scientists in mind; its command set is well suited to producing graphics for inclusion in reports and papers. These graphics are often difficult or impossible to produce in a graphics package. Scientists may require a large number of slightly different elements, or an exactly drawn mathematical shape. They may simply require line widths and font heights to be changed easily and repeatedly, (e.g. at the whim of a publisher!) If you either do not need, or can already produce, graphics that satisfy these requirements then it is probably not worth taking the time to learn GLE. If you need technical diagrams, and lack the artistic ability or tolerance to produce them by hand, then GLE can make their production great deal more painless.

1.2 What do I need to use GLE?

Through necessity this document must start from a basic level of computer and scientific literacy. A fair degree of experience using computers is assumed throughout; as is an acquaintance of programming concepts such as reusable code and subroutines. Obtaining the GLE program itself is covered in a following section, however some form of text editor with which to prepare the programming scripts is also required. Emacs or Pico in Linux, Edit in MS-Dos, or Notepad in Windows, are all perfectly suitable.

Many of the programming examples are mathematical in nature, and require the use of simple, and occasionally not so simple, expressions. These are on the whole self explanatory, at least at undergraduate level and above. Where possible some explanation has been offered or, failing that, a reference to a more detailed description has been given. As much of GLE is based on the Cartesian coordinate system, a good grasp of trigonometry is essential. Converting between this and polar coordinates, or even simply calculating the components of a vector are skills routinely used in most GLE scripts.

The examples included are meant to be representative of those found in text books or papers. A word of warning however, their purpose is to illustrate the use of commands, not to teach any meaningful science. The data used was not checked for accuracy, and in some cases entirely made up. Equally, to make the diagrams as simple and illistrative as possible some physical correctness may have been lost.

1.3 Where are we going?

Due to the public nature of the GLE code we shall beginning with some, unfortunately necessary, technical notes on the different versions available. Following this we shall examine a series of worked examples falling into the following sections:

Simple applications

As an introduction we look at diagrams that can be drawn using the simplest possible building blocks. These can be accomplished both within the GLE program, and with a traditional graphics package. The examples will illustrate the different approaches taken in each case.

Formal development

The programming aspect of the language is developed allowing us to produce much more complex routines. This shows many of the advantages, and some of the disadvantages, GLE has over its alternatives.

The graph module

A major part of GLE is its ability to take raw data and produce a graph with a high degree of control over the finished appearance. This section is largely self-contained and relies only a little on the previous sections.

More complex examples

A look at some of the more complex graphics that can be produced, using some of the less common commands.

We shall conclude with a reference section that explains the commands used in GLE, and contains most of the available formatting options.

2 Technical Details

Both the program files and the source code for GLE are available at no cost over the internet. The open nature of the source code has spawned several different versions of GLE; many of which have slightly different syntax for entering the programming script. This guide has been written explicitly for version 3.5, older versions will not contain all the features discussed here. A newer version, GLE2000, exists for Windows 95/98 which has some additional commands and a better, that is to say an icon driven, user interface. Many of these revisions come with their own comprehensive user documentation, which details not only installation and setup procedures, but should also cover any changes in syntax.

The existence of GLE in different versions and on different platforms means that some of the commands may differ slightly from those written below. It may be necessary to experiment with different names for fonts, shading, and colours, all of which are very sensitive to both the operating system and version. GLE 3.5, for example, uses a shorthand tagging system for its fonts, while GLE 2000 uses the full name as listed in the Windows directory. Where possible the examples have avoided the use of these less well defined variables; however experimentation may be necessary, not only to complete the examples presented here but also to exploit the additional features that may not.

2.1 Download and installation

GLE is available within the public domain and can be freely downloaded from several sites across the internet. For executables and source code for both PC and Linux machines see :

Derek Ingram's GLE2000 can be found at:

http://softlad.narod.ru/graphics_and_drawing/flow_chart_and_diagram_making/ .

Many of the sites from which GLE can be downloaded provide comprehensive installation instructions for different operating systems and machines. The installation of GLE 3.5 for DOS and Windows is detailed below, instructions for Linux can be found on Stephen Blundell's homepage at `http://users.ox.ac.uk/~sjb/gle/`

2.1.1 Installation for MS-DOS and Windows 2000/98/95

Files needed: `gle35-w32-RO-23.exe` (self extracting zip file)

Run `gle35-w32-RO-23.exe`, this will create a program directory directory `gle3.5`. Alternatively, if you have the non-executable zip file then unzip into your chosen directory (which we assume to be `c:\gle3.5` from now onwards).

We can enter the GLE directory either through the windows interface or, in MS-DOS, by typing

```
cd gle3.5
```

To compile a GLE script we first save it as a ASCII text file, then run GLE with the script. In MS-DOS mode we type,

```
gle_ps filename.gle
```

Or in Windows we may double click on the `gle_ps` icon, then type in the name of our file.

When GLE is run a 'font.dat' error may occur. In this case it is neccessary to set the environment variable to point to the directory where GLE is located. Start an MS-DOS prompt from the START menu in Windows, then:

```
cd \
edit autoexec.bat
```

The blue edit screen will now appear, scroll to the end of the file and insert the following line,

```
set GLE_TOP=c:\gle3.5\
```

Select save from the FILE menu, then exit. GLE should now run correctly, although it may be neccessary to restart the computer for changes to take effect.

2.2 Working with GLE

The GLE executable is merely an interface that takes a programming script and converts it into a postscript file for embedding within other documents. The script can be created as a plain ASCII file with any text editor. There are not, as yet, any dedicated editors that provide highlighted commands, debugging routines or other luxuries.

Once the script has been written then we can compile it using the GLE program itself. Ideally, this will produce a postscript, `.ps`, file in the same directory as the program file. More often though, GLE will produce a number of error messeges indicating the line number and type of error which occured. The error mesages are reasonably comprehensible, certainly compared with many other languages, and corrections should be made to the `.gle` file before it is saved and recompiled.

There are several options that can be added to the GLE command. For producing diagrams to be included within documents a useful option is the ability to produce encapsulated postscript, `.eps` files. To do this we add the option,

```
gle filename.gle -deps
```

to the command for Linux, and for MS-DOS we write,

```
gle_ps filename.gle /deps
```

There are two commands that help the external formatting of the text introduced in GLE 3.5. The first allows for the addition of a text comment in start of the postscript file that is produced. To add such a comment we insert the line,

```
pscomment text
```

Where 'text' represents the comment placed at the start of the postscript file. The command must be entered at the very start of the program, before the initial `size` command.

In some applications a GLE file will appear with a white border around it, a black bounding box, or may be missing a section of the graphic. This is usually due to confusion between GLE and some graphics programs, by default GLE adds a whitespace boundary to the graphic which may be interpreted in some applications (photoshop for example) as a change of origin. To reset the origin to (0,0) we add the line

```
tsbbtweak
```

Where again the command must come before the body of the program.

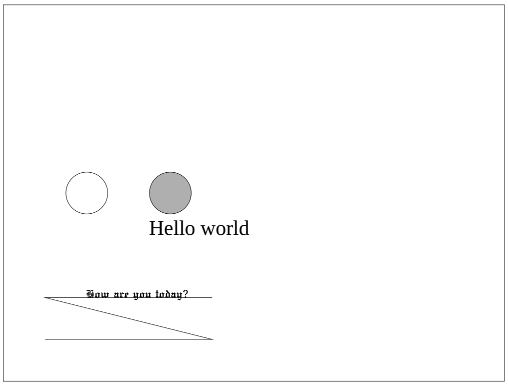
Although the diagrams included in this document are a close representation of the accompaning GLE code, there have been a number of alterations for the sake of typesetting. They have been scaled to fit within a sensible size on the page, in most cases a bounding box is added, and some of them have been clipped to remove unwanted construction lines. The scaling is due to the underlying document type (SGML), the bounding box and clipping are added using the `box` and `clip` commands covered in sections 3.1.5 (Shapes and sizes) and 6.4.3 (Clipping) respectively.

3 Simple Applications

We begin our tour of GLE with some basic examples of what can be done. For each we will present both the finished diagram and the script used to produce it along with a discussion of the important points raised within the script.

3.1 Hello world

This program demonstrates the use of absolute and relative movment and some basic text editing.




```
!Helloworld.gle - Illustrates the use of basic GLE commands

size 24 18                      !Sets the size of the graphic in cm

amove 2 2                       !Demonstrates the difference between relative
aline 10 2                      !and absolute movement
rline -8 2
rline 8 0
amove 7 7
                                !Displays some basic text
text Hello world
set font plge
set hei 0.5
amove 4 4
text How are you today?

rmove 0 5                       !Displays some simple shapes
circle 1
amove 8 9
circle 1 fill grey20
```

The small choice of commands in GLE means the syntax has not deviated too far from recognisable English. Even if the exact meaning of all the lines in the above code is not clear, it should be possible to follow through the steps which produce the results in the diagram.

3.1.1 The GLE code

Like all other programming languages GLE breaks down the task into a series of small steps. These are represented by the series of lines in the code, each line represents an individual step and the end of line character (produced whenever the return key is pressed) tells GLE where each step begins and ends. Although, as we shall see later, it is possible to combine several steps in the same command it is always necessary to begin each command on a separate line.

Most GLE commands require some sort of qualifier, an additional string of information that tells GLE how and where to perform the operation. In the above program we can see that the qualifiers are mainly numbers, in this case representing coordinates.

It is also worth noting early on the use of comments, as in many other high level programming languages GLE provides a system for the programmer to leave notes or descriptive passages within the code. These comments are invisible to the compiler and are not present in the finished diagram. Their sole purpose is to make the programming script both easier to read and more transparent when making revisions. In GLE we denote a comment by the use of the exclamation mark (!), any text entered between the excamlation mark and the end of line character that signifies the start of a new line is ignored entirley by the compiler. The `text` command is an exception to this rule, an exclamation placed after a `text` command is treated as an exclamation, and will appear in the diagram.

With the exception of the end of line character GLE ignores all whitespace in a program. The constructive use of this whitespace combined with suitable comments can make a program much easier to error check and revise, both for the original programmer and for others. Similarly GLE is case insensitive. The commands,

```
size 20 18
SIZE 20 18
sIZe 20 18
```

Are all, from the point of view of the compiler, identical. Again through the use of capitals we can make a script easier to read. There are no accepted conventions, but it make sense to stick to the same format throughout a program.

3.1.2 Measurments in the GLE environment

The first line of the 'Hello world' program is `size 24 18`. This defines the size of the diagram in cm, so that our working area is now 24cm by 18cm. Commands are referred to a grid of points from (0,0) to (24,18), any command that refers to a point outside this range will not appear in the final graphic. With a few exceptions GLE diagrams start with a `size` command, and should at least contain one somewhere within the program.

Notice that GLE will still compile a program with an illegal coordinate reference, one that refers to a point outside of our working area. The diagram will only show the part of the diagram that lies within the dimensions of the `size` command. This can be useful in cases where the diagram has a particular symmetry. For example, we may be interested in the lines of an electric field caused by a charge distribution. We can set up the equations that govern this distribution and then, using the `size` command, look only at a small portion of the field. Of course, the fact that GLE understands illegal coordinate references is no guarantee that any other publishing program will. Often a second graphics package will ignore the `size` command and attempt to draw out the entire graphic in all its glory. If this happens then the best solution is to add a `clip` command, covered in section 6.4.3 (Clipping), to the start of the program such that only the desired portion is viewed.

Fractional coordinates are also allowed, and in many cases essential. The resolution depends upon the output device but in general structures below 0.1 mm are not well resolved and lines less than 0.0001 mm are very faint. Negative coordinates are also allowed, although obviously lines cannot have a negative width nor fonts a negative height.

3.1.3 Relative and absolute motion

A GLE script consists of a set of instructions that GLE interprets to build up the completed picture. Each operation is referred to a coordinate point. At the start of the program the first command refers to the point (0,0) at the bottom left hand corner of the page. Thereafter the current point is defined by the end of the last operation, though what GLE defines as the end of an operation and what we define as the end of an operation may differ.

We have four basic commands for moving around the diagram, each expressed in terms of relative or absolute motion.

`amove x y`

Moves the cursor to the absolute coordinates (x,y).

`aline x y`

Draws a line from the current point to the coordinates (x,y).

`rmove x y`

Moves the cursor by a relative amount such that the new coordinates are the sum of the initial coordinates and the values of x and y.

`rline x y`

Draws a line from the current point to the point a vector (x,y) away.

Clearly both sets of commands are very similar, and the same diagram can be drawn with any combination of the above. However, absolute movements are less prone to errors and are easier to visualize. Relative movements are important when creating figures that can be expressed in terms of a continuous path or set of operations and can help simplify the construction of complex diagrams considerably.

We can add arrow heads to the lines we draw, both relative and absolute, by the `arrow start`, `arrow end` or `arrow both` commands. For example in our program we can draw an arrow on the final line with the command

```
rline 8 0 arrow end
```

The first few lines of our program therefore draws a simple shape using a combination of the above commands. Because of the nature of the GLE interface there is a lack of immediate feedback from any alteration to the program. To save time and typing it is a good idea to plan out the commands on graph paper, to get a general feel for the layout of the diagram, before attempting the GLE script.

3.1.4 Basic text support

The next section of our program deals with the basics of introducing text onto a diagram. The command `text` tells GLE that everything from the first character following the command to the next end of line character is to be printed onto the page. If the line of text is too long then the text will not all appear on the diagram; there is no provision for the text to run onto the next line. Writing text at a given coordinate point also does not change that coordinate point so if we had written:

```
text Hello
text How are you?
```

Then the ‘How are you’ would have been superimposed over the ‘Hello’ . To add a line of text after a previously written one we must instruct GLE to move the cursor to the end of the text box. For most fonts which do not have a constant letter width this would be difficult to do, however there is a command which we can use to simplify the problem. `xend()` and `yend()` are qualifiers which will return the coordinates of the last point drawn, thus to write a following string of text we would write:

```
text Hello
amove xend() yend()
text How are you
```

Where the command `amove xend() yend()` instructs GLE to move to the coordinates of the last letter of 'Hello' . Notice that we could *not* have solved the problem by writing:

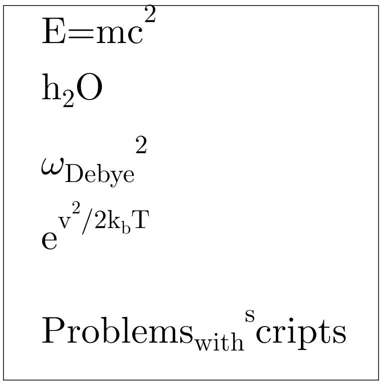
```
text Hello
text       How are you?
```

As GLE does not recognise the spaces placed immediately after the text command. GLE will, however, recognise many of the LaTeX special characters. In particular we may use `\:` to represent a space, and `\glass` to indicate the beginning of a line.

```
text Hello
text \glass \: \: \: \: \: How are you
```

The `\glass` command is neccessary as, by default, GLE will not print either real spaces or the character representation unless preeceded by another character.

A list of related LaTeX commands supported by GLE is given in the index. Note however that GLE is not a dedicated text compiler. The following program illistrates some of the problems that can occur in the use of some special characters.



```
!sub.gle - Demonstrates the problems that can occur when using sub- and superscripts
size 10 10
set font texcmr

amove 1 9       !Single sub- and super scripts work fine
text E=mc^2

amove 1 7.5
text h_2O
```

```
amove 1 5.5              !Problems occur when we try to add both sub- and
                         !superscripts
text \omega_{Debye}^{2}

amove 1 3.5              !We can of course nest expressions within each other
                         !using brackets
text e^{v^{2}/2k_{b}T}
amove 1 1                        !Without brackets GLE takes the first single
                                 !character
text Problems_{with}^scripts
```

The LaTeX commands for the use of subscripts and superscripts are _{} and ^{}, where the curly brackets can be replaced by normal brackets. In the absence of any brackets GLE will assume that only the first single character is to form the subscript or superscript. Greek letters are normally produced by a \ followed by the name of the letter.

The `set` command has a special role in GLE. It allows the defalt values for a number of commands to be specified. They will remain this way until another set command changes them, or until the program ends. Thus the command `set font plge` sets the font to Plotter Gothic English (plge), any text command that follows will display text in the Gothic font. We can also set the width of the lines in the stroked (plotter) fonts using the `set fontlwidth 0.2` command, which will set the line width to 0.2cm.

The `set hei 0.5` command defines the height of the font. For reasons stemming from the original typesetting of letters on printing presses a 10cm high font has letters that are only about 6.5cm high. Again the set command remebers the height, and includes it in all following text. If no `set` command is issued the default height is 1cm.

To write longer sections of text and tables there are dedicated commands that more accurately control the formatting. We mention one here, and defer a fuller treatment to another example. The `set just` command aligns the text either to the left, right or center. For example,

```
amove 4 4
set just right
text Hello
rmove -2 -2
set just center
text How are you
```

Will print two lines of text, one justified right of (4,4) and the other centred on (2,2).

3.1.5 Shapes and Sizes

The final section to our program draws some basic shapes. Many of these can be constructed from the `aline` and `rline` commands (though circles are difficult), but it makes sense to have dedicated keywords for the more common shapes. Following each shape command we add a qualifier that defines the size, and there are also optional qualifiers that can add fill shades and colours to the shape. The various fill styles are provided in a reference chart at the end of this document, they can take the form of either a named colour (black, white, red...), a grayscale (grey1 to grey100), or a predefined pattern. There is a difference between a white fill style and no fill style - a shape filled in white will obscure any drawing that has taken place beneath it, no fill style will allow it to show through.

We may also define the colour of the circle outline using the `set color` command (note the American spelling). We could have drawn a red circle with a blue filling using:

```
set color red
circle 2 fill blue
```

Another common predefined shape is the box, this requires two size variables a height and a width. Again we may specify line and fill styles as with the circle. For example:

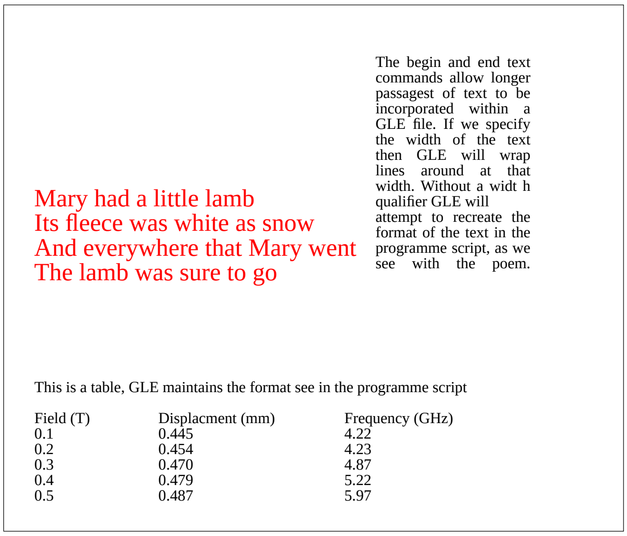
```
set color red
box 5 2 fill grey20
```

Draws a grey box with a red outline. To remove the outline we can add the qualifier `nobox` to the end of the box command which will tell GLE not to add an outline.

```
box 5 2 nobox
```

3.2 More text examples

In this section we take a longer look at some of the different text options we may use.



```
!text.gle - Demonstrates some of the text options available within GLE

size 20 17                      !Sets the size of the window

amove 1 10.5                    !Writes a poem
set color red
set hei 0.8
begin text
```

```
        Mary had a little lamb
        Its fleece was white as snow
        And everywhere that Mary went
        The lamb was sure to go
end text

amove 12 15                     !Writes an explanation
set color black
set hei 0.5
begin text width 5
        The begin and end text commands allow longer passagest of text to be
incorporated within a GLE file. If we specify the width of the text then GLE
will wrap lines around at that width. Without a width qualifier GLE will att
empt to recreate the format of the text in the program script, as we see w
ith the poem.
end text

amove 1 5                       !Draws a table
begin table

This is a table, GLE maintains the format seen in the program script

Field (T)       Displacment (mm)        Frequency (GHz)
0.1             0.445                   4.22
0.2             0.454                   4.23
0.3             0.470                   4.87
0.4             0.479                   5.22
0.5             0.487                   5.97

end table
```

The code demonstrates how the `begin text` and `end text` commands can be used to construct longer passages of text. Everything between the two commands is treated as a text file, GLE does not recognise any keywords or comments within the text however it will recognise most LaTeX commands for displaying special character. A list of these can be found in the appendix.

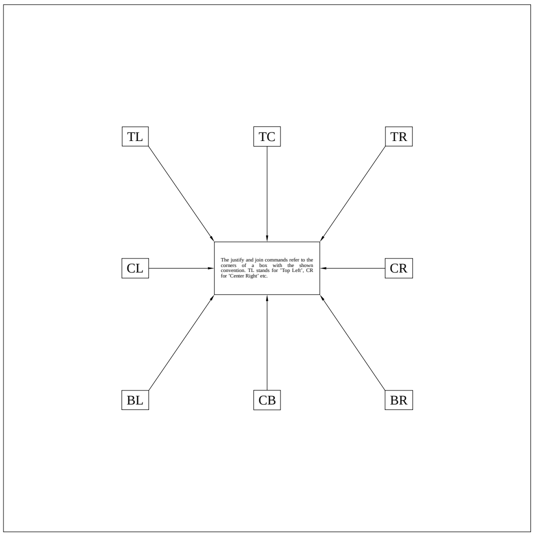
If no width is specified then GLE will attempt to format the text as it is set out in the script. With a width specified (in cm) GLE will wrap text around to the required width; though it will still respect end of line characters and multiple spaces.

The table option is used when the text must be aligned into columns. GLE reads the spaces and tabs and aligns the text accordingly. When creating tables it is best to use a fixed pitch font, that is one with a constant letter width, rather than a variable one. This aligns not only the left hand side of the text but the right hand ones as well.

3.3 All shapes and sizes

Here we look at more complex shapes that can be drawn as well as many shorthand commands that make the construction of flow diagrams or charts much easier. Be aware however that this section does not use the full programming ability of GLE. Some of the commands introduced in section 4 can make the following program much more compact.

So far we have accepted the default positioning of the shapes we have drawn. However it is possible to have a far greater control, both over the shapes and over the orientation of the text. The following program shows the naming convention that is used.



```
!justify.gle - Illistrates the shorthand names for the justify and join commands

size 20 20              !Sets the size of the viewing window

set just CC             !All text from now on will be center justified

amove 10 10             !Draws the box in the center and adds text
set hei 0.2
box 4 2 just CC name centerbox

begin text width 3.5
        The justify and join commands refer to the corners of a box with the shown
convention. TL stands for ’Top Left’, CR for ’Center Right’ etc.
end text

set hei 0.5             !Formats the following text

!The following commands write out the shorthand labels and draw boxes around them
amove 10 15
begin box add 0.2 name topbox
```

```
text TC
end box

amove 15 15
begin box add 0.2 name toprightbox
text TR
end box

amove 15 10
begin box add 0.2 name rightbox
text CR
end box

amove 15 5
begin box add 0.2 name bottomrightbox
text BR
end box

amove 10 5
begin box add 0.2 name bottombox
text BC
end box

amove 5 5
begin box add 0.2 name bottomleftbox
text BL
end box

amove 5 10
begin box add 0.2 name leftbox
text CL
end box

amove 5 15
begin box add 0.2 name topleftbox
text TL
end box

!These commands join the central box to the others with arrows
join centerbox.tc <- topbox.bc
join centerbox.tr <- toprightbox.bl
join centerbox.cr <- rightbox.cl
join centerbox.br <- bottomrightbox.tl
join centerbox.bc <- bottombox.tc
join centerbox.bl <- bottomleftbox.tr
join centerbox.cl <- leftbox.cr
join centerbox.tl <- topleftbox.br
```

3.3.1 Naming objects

Within GLE we can refer to a point on the screen or an object by a predefined names, GLE will substitute the absolute coordinates of the point. This has the advantage that, in a subsequent line of programming, any given point can be referred to by name as opposed to a string of coordinates. Thus, if we require the coordinates of the top left hand corner of a box, we can refer to it by its name followed by an extension for the left hand corner. We can also save an individual point for later use,

```
amove 4 4
save point1
amove 3 3
save fred
join fred - point1
```

This will join the two points (4,4) and (3,3).

3.3.2 The justify commands

The second line of the program `set just CC` tells GLE that all following text should be centered over the reference point. Notice that this is different from the `center` command which positions the text according to its bottom center. Similarly `left` is identical to `BL` (bottom left), and `right` is identical to `BR` (bottom right). We may use the labels interchangably, and we may also swap the order of letters, so `BR` is the same command as `RB` or `rb`.

The `set justify` command only applies to text, so when we come to draw the box around the text with the command,

```
box 4 2 just CC name centerbox
```

We must specify the positioning with the `just` command. `CC` can be replaced with any other handle as required.

3.3.3 The join command

The usefulness of naming objects is shown above when we wish to join together two objects. We could, of course, have used the `aline` command, but then we would need to know the coordinates of each point on the boxes. It is far easier to refer to each point by its name and let GLE work out exactly where it is.

The `join` command draws a line between two named points in the same way that the `aline` command draws a line from the reference point to the given coordinate. So the command:

```
join abox.tl - anotherbox.br
```

Will connect the top left hand corner of '`abox`' to the bottom right hand corner of '`anotherbox`'. The '-' can be replaced with a '<-' for a single arrow; a '->' for an arrow in the opposite direction; or '<->' for a double headed arrow.

Along with the commands above there are four other options we should look at. The first occurs if we fail to specify any justification at all on an object:

```
join object1 - object2.tl
```

In this case GLE will join the top left hand corner of `object2` to the centre of `object1`, however the line will be clipped at the edge of `object1`.

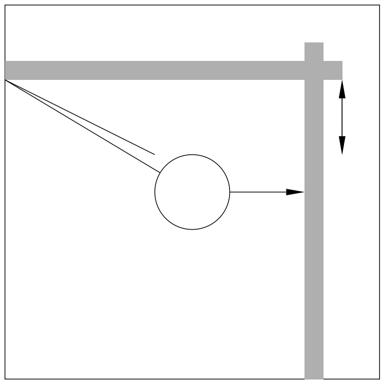
We may also specify that the line joining two objects be either horizontal or vertical, using the `h` and `v` qualifiers.

```
join object1.h object2
```

Indicates that `object1` should be joined to `object2` by a horizontal line. If two statements are present then GLE will draw a line which satifies both, but the lines will no longer neccesarly connect the two objects.

Finally we can use the command `ci`. This is similar to the absence of any justify type, but in this case GLE will clip the line at the circumference of the largest circle that can be drawn within the confines of the box. Obviously, the main application of the `ci` command is to join circles.

The following code demonstrates many of these remaining points:



```
!just2.gle - Demonstrates the remaining justify commands
size 10 10

amove 5 5
                                !Draws the centre circle
begin box name circle nobox
circle 1
end box

amove 0 8                       !Draws two boxes
box 9 0.5 name topbox nobox fill grey20

amove 8 0
box 0.5 9 name rightbox nobox fill grey20

join circle.v <-> topbox.rb             !Joins the various objects
join circle -> rightbox.h
join circle.ci - topbox.lb
```

```
join circle.tl - topbox.lb
join circle.v <-> topbox.rb
```

Notice that the `circle.v <-> topbox.rb` creates a line that does not come into contact with the circle itself. We also see that using the command `circle.tl - topbox.lb` joins to the corner of the box containing the circle. If we wish to connect to the circle itself then we must use the `ci` justifier.

3.3.4 Begin box and end box

Although we now have a fair degree of control over the positioning of the shapes, in many cases it is an advantage not to have to specify the dimensions of the shape at all. We may instead merely wish to have a box that encloses some text with a margin on each side, without having to worry about the text overrunning the sides of the box. For this the `begin box` command is provided. This draws a box around everything in between the `begin box` and `end box` command. This includes text, other shapes, or graphs. The usual qualifiers can be added to the box command with the addition of `add` which specifies the margin left between the extent of the enclosed object and the edge of the box. For example,

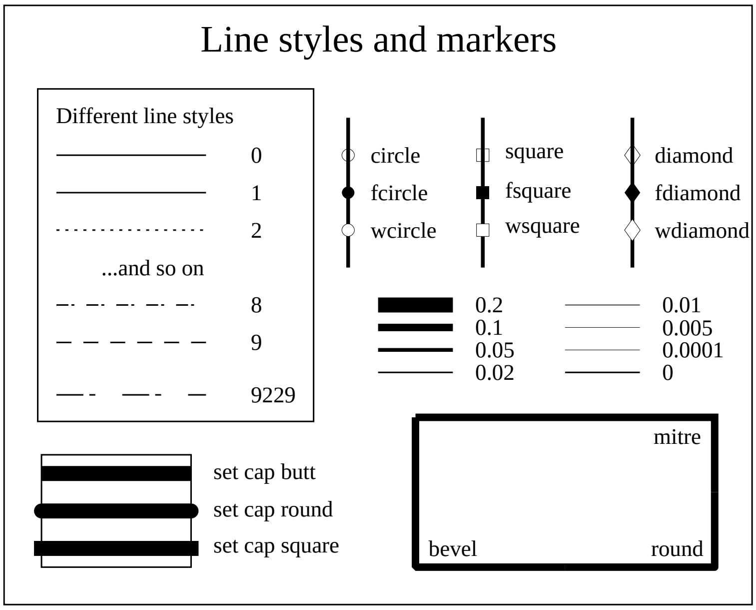
```
begin box fill grey20 add 1.5 nobox name mybox
        (...)
end box
```

Will draw a grey box without a border aroung the enclosed commands (...), the `add 1.5` command gives a 1.5cm margin on all sides. In the remaining program it will be possible to refer to the coordinates of the box using the name `mybox` and one of the corners. For this we do not know, or need to know, the absolute coordinates of the box.

3.4 Lines and line widths

We look at another example of the use of graphics. Notice how this program makes the distinction between absolute and relative movement. Relative movement is used in the construction of each individual block and absolute movement is used in moving between the blocks. In an ideal world there would be no advantage to either, but using this form ensures that if we make an error and wish to move one of the blocks then only one coordinate must be changed. Similarly, if we wish to resize one of the elements of the diagram then it is easier to deal in relative measures rather than having to work out the absolute position of each point.

```
!lines.gle - Demonstrates the various options for lines and markers
size 10 8

set hei 0.5
set just CC                     !Adds a title
amove 5 7.5
text Line styles and markers

set hei 0.3
set just LC             !Adds a title and box to the different line styles
amove 0.7 6.5
```




```
begin box add 0.25
        text Different line styles

        rmove 0 -0.5                    !Draws the various line style options
        set lstyle 0                    !Ideally this should be placed
                                        !within a subroutine
        rline 2 0                       !See following section
        text \glass \: \: 0

        rmove -2 -0.5
        set lstyle 1
        rline 2 0
        text \glass \: \: 1

        rmove -2 -0.5
        set lstyle 2
        rline 2 0
        text \glass \: \: 2

        rmove -2 -0.5
        text \glass \: \: ...and so on

        rmove 0 -0.5
        set lstyle 8
        rline 2 0
        text \glass \: \: 8

        rmove -2 -0.5
```

```
        set lstyle 9
        rline 2 0
        text \glass \: \: 9

        rmove -2 -0.7           !We can produce a line from a combination of
        set lstyle 9229         !the other line styles
        rline 2 0
        text \glass \: \: 9229

        set lstyle 1
end box


amove 4.6 6.5                   !Demonstrates the different markers available

set lwidth 0.05
rline 0 -2

set lwidth 0.0001
rmove 0 0.5
marker wcircle 0.8
text \glass \: wcircle

rmove 0 0.5
marker fcircle 0.8
text \glass \: fcircle

rmove 0 0.5
marker circle 0.8
text \glass \: circle

amove 6.4 6.5

set lwidth 0.05
rline 0 -2

set lwidth 0.0001
rmove 0 0.5
marker wsquare 0.8
text \glass \: wsquare

rmove 0 0.5
marker fsquare 0.8
text \glass \: fsquare

rmove 0 0.5
marker square 0.8
```

```
text \glass \: square

amove 8.4 6.5

set lwidth 0.05
rline 0 -2

set lwidth 0.0001
rmove 0 0.5
marker wdiamond
text \glass \: wdiamond

rmove 0 0.5
marker fdiamond
text \glass \: fdiamond

rmove 0 0.5
marker diamond
text \glass \: diamond



amove 5 4                        !Draws examples of the different line widths
set lwidth 0.2
rline 1 0
text \glass \: 0.2

rmove -1 -0.3
set lwidth 0.1
rline 1 0
text \glass \: 0.1

rmove -1 -0.3
set lwidth 0.05
rline 1 0
text \glass \: 0.05

rmove -1 -0.3
set lwidth 0.02
rline 1 0
text \glass \: 0.02

amove 7.5 4

set lwidth 0.01
rline 1 0
text \glass \: 0.01
```

```
rmove -1 -0.3
set lwidth 0.005
rline 1 0
text \glass \: 0.005

rmove -1 -0.3
set lwidth 0.0001
rline 1 0
text \glass \: 0.0001

rmove -1 -0.3
set lwidth 0
rline 1 0
text \glass \: 0


amove 0.5 0.5                   !Shows the different options for ending a line
box 2 1.5

rmove 0 0.25
set cap square
set lwidth 0.2
set hei 0.3
rline 2 0
text \glass \: set cap square

rmove -2 0.5
set cap round
rline 2 0
text \glass \: set cap round

rmove -2 0.5
set cap butt
rline 2 0
text \glass \: set cap butt


amove 5.5 2.5                   !Shows the different options for joining lines
set lwidth 0.1
set just CC                     !Notice that the path is split - this is significant
set join mitre
rline 4 0
rline 0 -1
set join round
rline 0 -1
rline -2 0
```

```
set join bevel
rline -2 0
rline 0 2

rmove 0.5 -0.25                 !Adds text labels to the various joins
rmove 3 0                       !We could not have done this within the
                                !join commands themselves - see main text
text mitre
rmove 0 -1.5
text round
rmove -3 0
text bevel
```

The purpose of many of the commands can be seen from the graphical output, this kind of program with a lot of repeated steps can be simplified greatly by the use of subroutines. However, we will postpone a discussion of other possible programming structures until the following section, dealing here only with the commands' literal meaning.

After defining the size of the page and adding a title, the script draws a number of different line styles using the command:

```
set lstyle n
```

Where n is an integer which represents the different line styles, there are 10 (0..9) pre-defined styles. We can make additional ones by combining digits,

```
set lstyle 9229
```

This produces a line style that is a sum of individual styles, we begin with a black line style 9, then superimpose on top a style 2 in white, followed by a style 2 in black and a style 9 in white.

Markers are shapes that are attached to a given point in a diagram, we are free to define their size using a height in cm following the marker command. As with fonts the actual height of the marker is only about 65% of the specified height.

We can also specify the width of a line created either with the `rline` and `aline` commands or with a `join` command. We use,

```
set lwidth 0.2
```

Which sets the line width to 0.2 cm. The default setting for the width is 0.02, if we set the line width to zero then GLE will adopt the default value. Thus, a line width of 0.0002 is (100 times) thiner than a line width of zero.

When working with significant line widths we can specify how the lines begin, end, or join. The first two are controlled by the `set cap` command. This can either produce a line that is clipped exactly, a line that has a rounded cap at each end, or a line that has a square cap at each end. The various forms are illustrated above.

The `set join command` is more complex, by default GLE will simply overlap two lines that are connected to the same point. To invoke the different types of join we must form a continuous path from one point through another and onto a third. That is,

```
set join round
amove 1 1
aline 1 2
aline 2 1
```

Will form a rounded join at (1,2). If we break the path by inserting a command in between,

```
set join round
amove 1 1
aline 1 2
text This is not a rounded join
aline 2 1
```

Then the join style will not apply. If either the incoming line or outgoing line is under a different join style then, again, the style will not apply.

```
set join mitre
amove 1 1
rline 1 0
rline 0 1
set join round
rline -1 0
```

This will produce a mitred join at (2,1) but the rounded join will not apply at (2,2) as the incomming line still has the previous `set join mitre` style. If we were to produce a rounded join then, as in the script above, we would need to split the line into two parallel, colinear, segments.

4 Formal development

So far we have looked only at programs that proceed from one command to another using each only once. Although this has made our first few programs easy to follow, it does not demonstrate the full potential of GLE as a programming language. In the following section we will develop the drawing routines already studied within the context of a full programming language. We shall look at how, by reusing code and defining subroutines, we can greatly simplify the construction of a complex program. These concept will be familiar territory to anyone with a knowledge of other high level programming languages such as C or Pascal.

To begin with we formalize some of the syntax we have been using. We define a command as a keyword that prepares GLE for a task. Following this is a set of qualifiers that define the parameters of the task; the width and height of a box or the coordinates of a point for example. We also distinguish between qualifiers that are single variables and those that are commands in their own right. The instruction set to draw a box may be written as:

```
box 2 4 fill grey20 justify CR
```

We see that 2 and 4 are single variables that are required to draw the box successfully. The other qualifiers take the form of a sub command followed by another qualifier, these are optional - removing them will result in the box being drawn with the default settings. The order of the sub-commands does not matter but we cannot separate command and qualifier, nor can we place the sub-commands before the single variables, the `2 4` must not be separate from the `box` command.

When GLE finds a keyword in the script it looks to the right of that word for any qualifiers that are necessary, then to the right of those for any optional sub-commands and their qualifiers. GLE ignores all multiple spaces, though each word must, at least, be separated by a single space. The end of line character is a signal to GLE that the current command sequence has ended and it should proceed to the next line of code.

The `set` command is a keyword and, if we have several variables to specify at the same time, then we may shorten the usage to a single line. Suppose we have the code:

```
set hei 3
set color red
set just center
set font ss
```

GLE reads the keyword `set`; sets the font height to 3, then forgets about the set command. In the second line it follows the same process but sets the colour to red. It makes sense that we can shorten the code to a single `set` command that we write as,

```
set hei 3 color red just center font ss
```

Which accomplishes the same thing but using the command to sub-command system.

4.1 Variables and expressions

So far we have explicitly provided values for each command we have used. For example, when drawing a box we gave numerical values for both the height and width. It is useful, however, to be able to define a variable that represents a number or a string of text. For example we may write:

```
a=3.1415
b=3
box a b
```

Which tells GLE to draw a box of width '`a`' and height '`b`', where '`a`' and '`b`' are defined to have the numerical vales 4 and 3.1415. If a variable is not defined before it is used in the program then it will take the default value of zero.

Any combination of letters and numbers may be used to represent a numerical variable, it makes sense to choose ones that are both memorable and sensible. Text strings can be represented by variables that end in a '$' sign, `text$` for example. This works fine for most commands but the `text` command treats text strings as plain text. For example, the set of commands:

```
a$=Hello
text a$
```

Will write the text 'a$' to the screen. To display the variable text we use the command `write`, the only difference from `text` being the ability to detect and display text strings.

```
text1$=color
text2$=red
set color text2$
write This is the text1$ text2$ text3$
```

The string variable '`text3$`' is undefined at this point in the program. On fininding this GLE will simple disply the plain text 'text3$' . The `write` command does not preclude the use of $ signs, just any defined string variable containing them.

GLE contains and extensive library of built in functions which may also be used to simplify a program. Many of the expressions are mathematical, including all the standard operations: '+' for add, '-' for subtract, '*' and '/' for multiply and divide. We may include Boolean operators (AND, OR) and relational operators (<, >, =>, <=, =, <>). There are also trigometric and exponential functions, a set of functions for manipulating strings of text, and functions that return the current date and time. A full listing of the available expressions is given in the appendix.

Wherever a string or numerical value is used, an expression can also be used. Equally, numerical and string variables can be substituted into an expression. The following lines of code will draw three identical lines:

```
rline 3 2

rline 9/3 sqrt(4)

a=9
b=3
c=1+1
rline a/b sqrt(2*c)
```

Expressions in GLE are delimited by white space and therefore we must not include any spaces within the expressions themselves. Thus, `rline 3*2 4` is a valid expression, whilst `rline 3 * 2 4` is not.

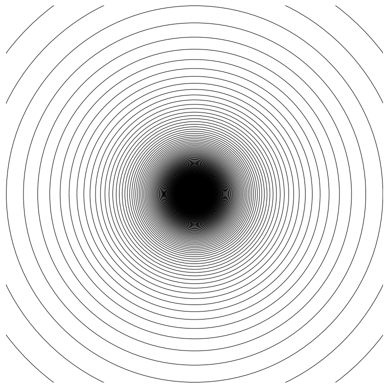
4.2 Conditional logic and programming loops

An important part of GLE programming is the use of programming loops and the logic that is needed to execute them. Suppose we have a repetitive task whose execution differs slightly each time. Instead of writing out each individual step we use a loop, a repeated piece of code that runs until the conditions specified at the top of the loop are satisfied.

4.2.1 The circles

We look at a simple example of the use of a loop

```
circles.gle
size 20 20
amove 10 10
for x = 0.01 to 100 step 0.01
        circle 1/x
        next x
```



Draws a series of concentric circles aproaching a limit at the center of the page. The loop is determined by the `for` command and the `next x` command, everything between these is part of the loop. The three qualifiers that follow this command determine the number and spacing of the loops, consider the loop,

```
for x = a to b step c
        (....)
        next x
```

On reaching this line of code GLE sets `x` equal to `a` and then repeats the following procedure,

- If `x` is greater that `b` then the loop is skipped and the command immediately following the loop is exectuted.
- The value `c` is added to `x`
- The code between the `for` and `next` commands is executed

There is nothing special about x, it can be any variable we wish. Notice, however, that the variable used in a loop is not a local variable as with a subroutine. If we define `x` in the code preceding the loop then in the following code the value of `x` will be the last value it took in the loop.

4.2.2 `If...then` statements

A similar command can be used to execute different code depending on whether a given statement is true or not.

```
if xpos()=3 then
        text We are at x=3
else
        text We are not at x=3
end if
```

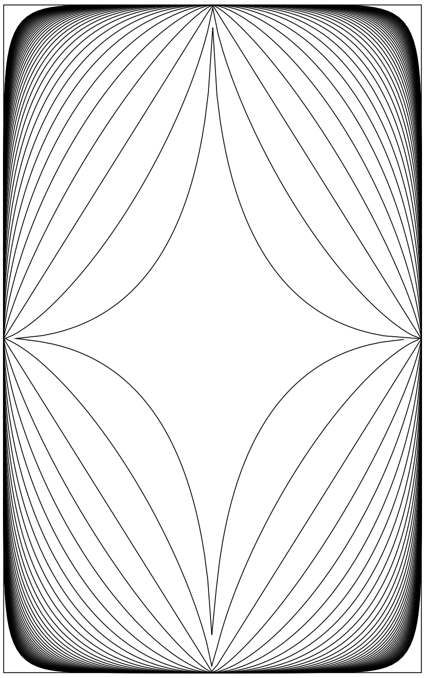
If the statement immediately following the `if` command is true then GLE executes the code between the `then` command and the `else` command, if it is false then GLE executes the code between `else` and `end if`. Either way GLE then moves onto the line of code imediatly following `end if`. In this case the program will print 'We are at x=3' if the x coordinate is 3, or 'We are not at x=3' if it is not.

4.2.3 The superellipse

We look at a more concrete example using both the loop and conditional logic, this illustrates the great advantage GLE has over traditional drawing programs. The following code draws Piet Hein's superellipse, a set of shapes that satify, for different values of n, the equation

$$(x/a)^n + (y/b)^n = 1 \tag{1}$$

For a discussion of the superellipses origin, and the diagram itself, see *J. Phys: Condensed Matter* 13, 2271 (2001).



```
!Superellipse.gle - Graphical representation of Piet Hein's superellipse

size 24 18

amove 10 9
begin origin            !Sets the origin of the ellipse at (10,9)

a=5                     !a and b represent the excentricity of the ellipse
b=8
n=1.5

amove -a -b             !Draws the box that represents the limit as n tends to
box 2*a 2*b             !infinity
amove 0 0
```

```
for j=0.5 to 10 step 0.25
n=j                      !n is the exponent of the ellipse

for i=0 to 360 step 1    !Calculates the radial distance from the x and y
ang=i*3.14/180
c=cos(ang)
if (c<0) then
  c=-c
end if
s=sin(ang)
if (s<0) then
  s=-s
end if
ax=(c/a)^n               !ax is the projection of the radial coordinate along
                         !the x axis
ay=(s/b)^n               !ay is the same along the y axis
zob=1/(ax+ay)
if (zob<0) then
  zob=-zob
end if
zub=zob^(1/n)
x=zub*cos(ang)
y=zub*sin(ang)
if i=0 then
 amove x y
else
 aline x y
end if

next i
next j
```

4.3 Subroutines

Although variables may be defined anywhere within a code it makes sense to place them at or near the start of the program. This allows the variables of the program to easily be changed if the need arises, and also, reduces the number of lines within the central body of a program. Reducing the size of the main programming loop makes the program easier to understand, and to check for errors. A similar technique applies to the writting of subroutines. A subroutine is a distinct section of code that perform a particular task. For example, within a program we may be required to draw a triangle,

```
amove 2 2
rline 2 0
rline -1 sqrt(3)
rline -1 -sqrt(3)
```

We can write this section of code at the beginning of the script using the `sub` and `end sub` commands.

```
sub triangle
        amove 2 2
        rline 2 0
        rline -1 sqrt(3)
        rline -1 -sqrt(3)
end sub
```

Indentation is optional but can make the program easier to understand. We may call the triangle at any point within the program body using the command,

```
@triangle
```

Where the `@triangle` commmand tells GLE to execute a subroutine with the name 'triangle' .

This has accomplished our goal of simplifying the program body by removing self contained code to the start of the program and giving it a logical call name. It would, however, be far more useful if we could specify the size and position of the triangle each time we draw it. We do this by specifying a series of local variables within the subroutine.

```
sub triangle x y size
        amove x y
        rline (size) 0
        rline -(0.5)*(size) (0.5)*(sqrt(3))*(size)
        rline -(0.5)*(size) -(0.5)*(sqrt(3))*(size)
end sub
```

Again we may call the triangle at any point in the program, however we must now specify the size and position of the triangle.

```
@triangle 2 3 1
@triangle 0.2 sqrt(3.4) 0.5
@triangle a b c-4
```

The variables used within a subroutine are entirly separate from the program body. They are forgotten as soon as the subroutine is finished; so the size variable in the above line of code is undefined in the program body. We are, of course, free to define a variable 'size' and use it within the program body. It will neither affect nor be affected by the execution of the subroutine.

Commonly used subroutines can be removed entirly from the programming script and placed in a separate file which can be made available to any number of GLE programs. The following subroutine substitutes a character (a snowflake) from one of the font tables as a marker for either a graph or diagram.

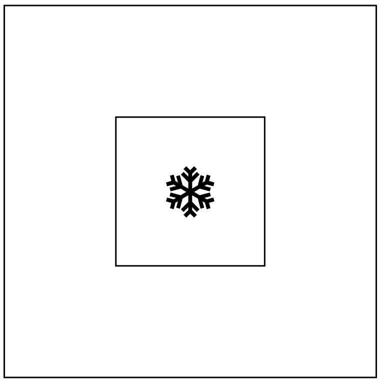
```
!subsnow.gle - Subroutine that prints a snowflake
sub subsnow size                !Size is the font size of the snowflake
gsave                           !Saves the initial graphic state
set font pszd hei size
t$ = "\char{102}"
rmove -twidth(t$)/2 -theight(t$)/2  !Centers the snowflake over the initial coordinates
```

```
write t$
grestore                        !Restores the saved graphic state
end sub
```

There are several new elements within this code. The `gsave` and `grestore` commands respectively save and restore the current graphic state. When the subroutine is called GLE remembers the initial coordinates and settings of the font height and type. Thus the set command that appears in the subroutine is then not allowed to affect the main program loop, an advantage if we wish to construct a piece of code that will fit into any program we choose.

The rest of the subroutine writes character 102 (a snowflake) from the postscript zapdingbat font at the center of the initial coordinates. The quotation marks around the definition of 't$' indicate to GLE that the string reference is to be taken as a special character and not a literal piece of text. The `twidth(t$)` and `theight(t$)` are special functions that give the width and height of the character in brackets, if it was printed with the current font size settings.

This piece of code is a perfectly valid subroutine and can be placed at the top of any GLE program. However, if we draw a lot of snowflakes we can reduce the complexity of our GLE script by calling the subroutine from within the program. The above code is placed in a separate file `snowflake.gle`. Notice that this is not a valid GLE file, it does not have a size declaration nor any program body. We can however tell GLE that this subroutine is to be compilied with a program using the `include` command. We write,



```
!Snowflake - produces a boxed snowflake
size 5 5
include subsnow.gle             !Subroutine for drawing a snowflake
define marker snow subsnow      !Defines a marker to be the snowflake produced
                                !by the subroutine
amove 2.5 2.5
box 2 2 just CC
marker snow 1                   !Draws a 1cm snowflake
```

The code defines a new marker `snow` that draws the snowflake in the subroutine in the same way as any other marker. The box is to show that the snowflake is centered on the initial coordinates.

We have looked at including subroutines in our program, but we can also use the `include` command to run any GLE script at any point within another program. Sometimes we may have a very large or complex program that we wish to use again in another script, in this case the `include` command can be used instead of copying out the entire file. If we wish to call a very complex program then we can also use the command `bigfile`, this is identical to the `include` command except the program will be read and compliled one line at a time. This can speed up the execution of complex programs, however there are certain restrictions: some complex multi-line commands will not work and we cannot define subroutines within the `bigfile` program.

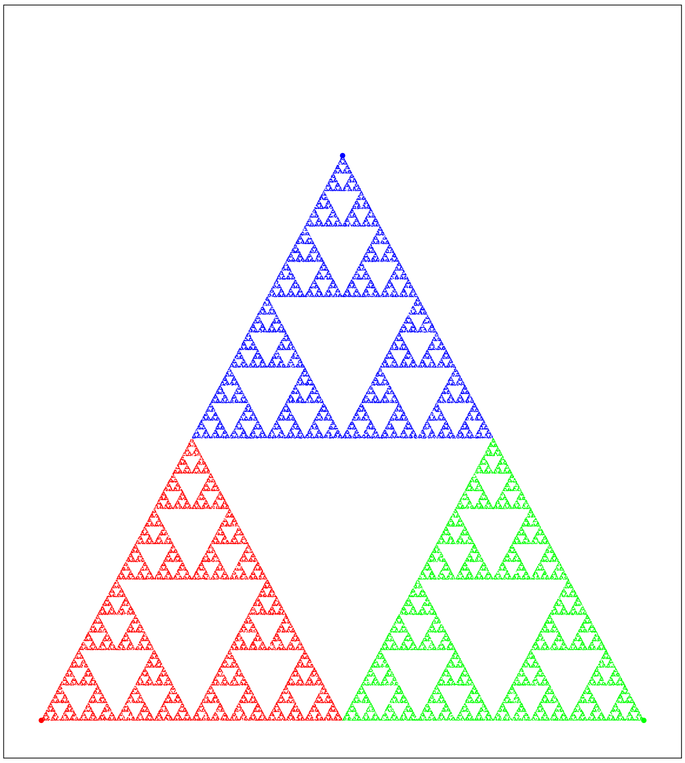
Frequently used subroutines can be stored in a separate user directory and called by any program. For example the following code will produce a shadowing effect on any given text.

```
sub ziptext tt$
gsave
for i = 1 to 0 step -0.05
set color (i)
write tt$
rmove -0.05 0.025
next i
set color white
write tt$
grestore
end sub
```

We call the subroutine at any point using the usual set of commands, we are able to pass any text string we wish onto the subroutine.


```
!glezip.gle - Demonstrates the ziptext command
size 7 5
include ziptext.gle

amove 2 1
set font rm hei 2
@ziptext "GLE"
```

We also look at a final example of the use of subroutines which gives an idea of the maximum processing speed that GLE is capable of. When using programming loops it is a good idea to start with only a small number of iterations, before building up to the number required to produce an accurate looking diagram. It is also important to remember that most postscript viewers are not designed to display very complex diagrams and will not do so with any haste.



```
!fractal.gle - Draws a fractal
size 18 20

rx=1            !Defines the position of the coloured markers
ry=1
bx=9
by=16
gx=17
gy=1

amove rx ry     !Draws the coloured markers
set color red
marker dot 0.4
amove bx by
set color blue
marker dot 0.4
amove gx gy
set color green
marker dot 0.4

                !This set of subroutines moves to halfway between the initial
                !coordinate and the coloured marker and draws a dot
sub green
set color green
```

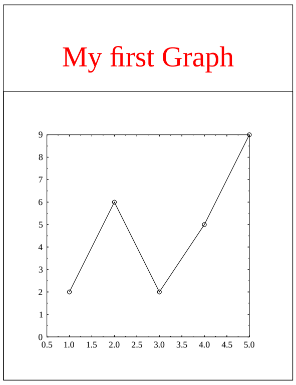
```
amove (gx+xpos())/2 (gy+ypos())/2
end sub

sub blue
set color blue
amove (bx+xpos())/2 (by+ypos())/2
end sub

sub red
set color red
amove (rx+xpos())/2 (ry+ypos())/2
end sub

amove 5 5

for n=1 to 100000 step 1        !Number of iterations defines the speed at
                                !which the program compiles

        r=rnd(1)                !Executes each of the subroutines randomly with
        if r<0.333 then         !the same probability
                @blue
                else
                if r<0.6666 then
                        @green
                        else
                        @red
                end if
        end if

if n>15 then            !Ignores the first few dots which are influenced by
marker dot 0.05         !initial start position
next n
else
next n
```

5 The graph module

GLE allows the user to produce a graph from data in a separate file or by defining a formula within a program script. The advantage of using GLE to produce graphs lies in its flexibility and the degree of control it allows over the final apperance. However, GLE does not include a dedicated spreadsheet. There is, of course, nothing to prevent us from preparing the data in a traditional spreadsheet and then producing the finished graph in GLE.

5.1 Graph basics

The graph module is a separate enitity within a GLE program. The commands `begin graph` and `end graph` define the limit of the module. Outside of these commands we are free to use any standard GLE command, between them there is a special command set appropriate for graph drawing. The following example draws a simple graph using data from a separate file.




```
!mygraph.gle - Some of the simple graph drawing commands
size 10 13                          !Sets the size of the paper
begin graph
        size 10 10                  !Sets the size of the graph
        data mydata.dat             !Reads data from mydata.dat
        title "My first Graph" hei 1 color red dist 2
        yaxis min 0                 !Sets the y axis to zero
        d1 line marker circle msize 0.2
end graph
```

The data file (`mydata.dat`), in the same directory as the source code, is:

```
1 2
2 6
3 2
4 5
5 9
```

The first command, `size`, sets the size of the paper we are working with. On this paper we can place any number of graphs or other drawings, though in this case we have only a single graph. The second size command, within the graph loop, sets the size of the graph within the page. To separate the graph from the rest of the page GLE automatically scales the graph to 70% of the given size and also adds a box. Entering the command `nobox` within the graph loop will remove the box. To change either or both of the default scaling we use the commands `vscale` and `hscale`. For example, inserting the commands:

```
size 10 10
nobox
vscale 1.2
hscale 1
```

would remove the box and scale the vertical axis to 12cm and the horizontal axis to 10cm. To set the graph size to 100% and remove the box we can use the shortcut command `fullsize` within the module.

Data is stored in GLE as a set of (x,y) coordinates, each with a name given by `dn` where `n` is an integer. The default for the first data set is `d1`, `d2` for the second, and so on. The `data` command tells GLE which file to use for the data set, and in the above example we use the default settings. If we wish we may specify any data set.

```
data mydata.dat d34
```

More usefuly if we have a data file with multiple columns, we can define a data set using any pair of columns.

```
data experiment.dat d1=c1,c2 d2=c1,c3
```

Which would read columns 1 and 2 of `experiment.dat` into `d1`, and columns 1 and 3 into `d2`.

The `title` command places, as might be expected, a title onto the graph. The title must be contained within quotation marks and may be followed by commands defining the height, colour, or font of the title. The `dist` command sets the distance of the title (in cm) from the top of the x-axis, the default setting is zero with positive and negative values also allowed.

Another default setting that is often altered is GLE's habit of taking the origin of the graph to be the lowest (x,y) data pair. To set the origin and the maximum we use the command,

```
xaxis min 0 max 5 dpoints 1
yaxis min 0 max 10 dpoints 1
```

Where `min` and `max` determine the maximum and minimum values of the x and y axis. `dpoints` sets the number of decimal places displayed in the data.

To plot the data we use the data label (d1, d2, d3, ...) followed by a set of commands that determine how the data is plotted. The line command tells GLE to join the points with a series of straight lines. We can apply the usual formatting options for lines by adding qualifiers after the main data command.

```
d2 line lstyle 2 lwidth 0.01 color red
```

The `d1 marker` command indicates that each point should be represented by a set of circular markers of size 0.2cm. The markers are identical to those used outside of the graph module: a list can be found in the appendix. By default neither lines or markers are drawn by GLE, and an error in this command can often be the reason behind blank graphs.

The dataset commands have the same syntax as the `set` function introduced in the previous section. The subcommands can appear on the same line, or on separate lines each with its own data label. The following script is identical to the single line command,

```
d1 line
d1 marker circle
d1 msize 0.2
```

Often we shall use the longer version when highlighting new commands or to make programs easier to understand. If we wish to format a group of data types at the same time then we can also use the `dn` command, which applies to all the defined data sets at the same time.

```
dn marker circle
d3 marker square
```

This will draw all of the data sets with circles except for the set d3 which will be drawn with squares. Notice that the dn command will overwrite any previous formatting commands, so if we had written:

```
d3 marker square
dn marker circle
```

All the data sets would be circles regardless of the `d3 marker square` command.

If a data point is missing (there is no value supplied in the data file) then, by default, GLE will not connect the points on either side of the blank. To connect all points, even if they are not consecutive, we can add the `nomiss` command, either in the same statement as the `line` command, or with its own separate data label.

5.2 Examples of formatting

To cover all the possible options for formatting both the axis and line styles we shall look at a series of more complex graphs. The first example is somewhat artificial, in that both graphs use the same (made up) data set, but it does serve to illustrate the degree of control it is possible to take over the final appearnce of a graph. This flexibility is clearly an advantage, but can be time consuming and sometimes unneccesary. In most cases GLE will do a reasonable job of producing a graph using the default settings.

```
!format.gle - Demonstrates the various format options for the axis of a graph
size 18 27
amove 1.3 14.5

begin graph
        size 16 12
        nobox                                  !Formats the axis
        xaxis min 0 max 20 dticks 4 dsubticks 1
        xplaces 4 8 12 16 20
        xnames "Sep 13" "Sep 23" "Oct 3" "Oct 13" "Oct 23"
        xticks length 0.2
        yticks length 0.2
        yaxis min 0 max 75 dticks 10 dsubticks 5 nofirst
        ytitle "Bud burst (%)"
        xtitle ""
        data test.dat


                !Draws the various data points
        d1 lstyle 1 marker wcircle msize 0.3 key "Shelter row"
        d2 lstyle 1 marker wsquare msize 0.3 key "Shelter row + H"
        d3 lstyle 2 marker wcircle msize 0.3 key "Middle row"
        d4 lstyle 2 marker wsquare msize 0.3 key "Middle row + H"

        key pos tl hei 0.4             !Draws a key
```

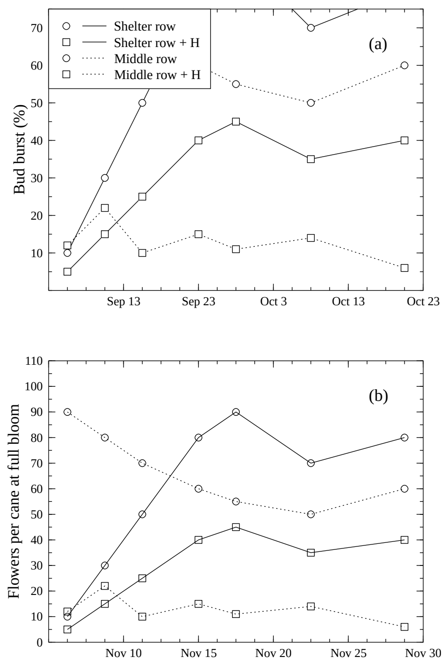


Figure 5: Influence of Hicane on the duration and timing of (a) bud burst and (b) flowering of kiwifruit. (Note: this data has been made up)

```
end graph

rmove 12 9                        !Labels graph (a)
set font rm hei 0.5
text (a)

amove 1.3 4
                                  !Draws the lower graph
begin graph
        size 16 12
        nobox                                     !Formats the axis
        xaxis min 0 max 20 dticks 4 dsubticks 1
        xplaces 4 8 12 16 20
        xnames "Nov 10" "Nov 15" "Nov 20" "Nov 25" "Nov 30"
        xticks length 0.2
        yticks length 0.2
        yaxis min 0 max 110 dticks 10 dsubticks 5
        ytitle "Flowers per cane at full bloom"
```

```
        xtitle ""
        data test.dat
        d1 lstyle 1 marker circle msize 0.3     !Draws the various data points
        d2 lstyle 1 marker square msize 0.3
        d3 lstyle 2 marker circle msize 0.3
        d4 lstyle 2 marker square msize 0.3
end graph

rmove 12 9                             !Labels graph (b)
text (b)

set hei 0.4                            !Adds an explanation of the data
amove 1.3 3
begin text width 16
Figure 5: Influence of Hicane on the duration and timing of (a) bud burst and (b)
flowering of kiwifruit. (Note: this data has been made up)
end text
```

5.2.1 Adding a simple key

There is a dedicated command module, similar to the graph module we are working with, that allow for the construction of keys. If, however, we only want to add a simple key then we can add a command from within the graph module. The `key` command has less formatting options available to it, but is simpler than the full key module. We indicate that we wish a dataset to be included in the key by placing the `key` qualifier in the data line followed by the title we wish the dataset to have,

```
d2 marker dot key "Observed values"
d3 line key "Predicted values"
```

We can change the position and format of the key by adding a subsequent `key` command,

```
key pos tl hei 0.4
```

This will draw the key in the top-left hand corner with text of height 0.4cm. We can remove the box that GLE adds by default with the qualifier `nobox`, for example,

```
key pos br nobox
```

5.2.2 Formatting the axis

A graph in GLE consists of a number of separate parts, four axes, their corresponding labels, and various titles. Each part has various formatting options that can be specified individually, or as a group. In this text we have tried to use all the possible formatting options for a given command. It is normally unnecessary to be so specific, as GLE will make reasonably sensible choices for most of the formatting.

The most important compontents, from the point of formatting, are the axes. These are referred to by the labels `xaxis, yaxis, x2axis, y2axis,` where the last pair refer to the upper and right hand axis. We can specify the formatting of these axis with the command,

```
xaxis color red lwidth 0.2 font texcmss hei 0.5
```

The syntax for the qualifiers is the same as for line styles and text outside the graph module. The other three axis can be specified in the same way,

```
y2axis color blue
```

By default all four axis will be drawn with labels only on the left and bottom. Changing the format of the `xaxis` will also change the formatting of the `x2axis`.

```
x2axis color blue
xaxis color red
```

Will still draw both the `xaxis` and the `x2axis` in red, though the y-axis will be unaffected. To produce a blue `x2axis` we would need to place the `x2axis` command after the `xaxis` one.

The axis commands control the overall format of the graph; they apply to the labeling, the title, and the appearance of the axis graduations which we call 'ticks' and 'sub-ticks'. Specifying a format option with the `xaxis` command sets it as the default for all following commands. So if we write,

```
xaxis color green
xtitle "Voltage"
```

The title 'Voltage' will appear in green as opposed to black.

We may turn off the `x2axis` and the `y2axis` with the commands,

```
x2axis off
y2axis off
```

In this way the `xaxis` and `yaxis` may also, if we wish, be removed. For the sake of completeness we mention that the command `xaxis on` will turn the x-axis back on again, though this can also be accomplished by not turning it off in the first place.

We may change the format options of the axis lines themselves using the command,

```
xside color blue lwidth 0.2
```

There are similar commands for the other three axis. Using this command the ticks, subticks, and the axes themselves will be changed, but the labels and title will remain intact. Again we may turn off the x-axis' lines, or any other lines, with the command,

```
xside off
```

And back on again with `xside on`.

The ticks and subticks, refering to the large and small graduations on the axes, can also be formatted. There are several commands that relate both to their positioning and to their appearence.

```
xticks lstyle 2 lwidth 0.01 length 0.2 color red
```

This will apply to the ticks and subticks on the top and bottom axes. We can specify the y axis as well.

```
y2subticks lstyle 2 lwidth 0.1 length 0.1 color black
```

This will apply only to the subticks on the right hand y-axis. Again the order of commands is important, as we may end up obliterating previous formatting by inadvertantly placing a `xaxis` command at the end of a graph module.

Ticks and subticks are draw on the inside of the graph, if we wish to move them to the outside then we simply specify a negative length.

```
x2subticks length -0.2
```

This will draw the top x-axis subticks on the outside of the axis.

The command `xlabels` formats both the data labels and the title of the x-axis, and the top x-axis. The converse is not true, `x2labels` formats only the top x-axis and leaves the bottom x-axis untouched.

```
xlabels color blue hei 0.3 font tt
x2labels hei 0.1
x2labels on
```

Where the `x2labels on` command tells GLE to add the `x2axis` labels which will, by default, not appear even if we attempt to format them. Again the `x2labels` format commmand must appear after the `xlabels` command if it is to have any effect. The `xlabels on` command may appear anywhere within the graph module, and we may include it in the format line: `x2labels hei 0.1 on`.

The ticks and subticks are allocated evenly accross the axes, but we can gain more control over the allocation using an `xaxis` command. We write,

```
xaxis dticks 3 dsubticks 1.5
```

Which will produce ticks every 3cm, with subticks every 1.5cm. Alternatively we can write,

```
xaxis nticks 5 nsubticks 20
```

Now the spacing of the ticks is changed such that there are 5 ticks and 20 subticks along the axis. We can combine the two commands by specifying a set number of ticks and a set distance between the subticks, but if we specify both a `nticks` command and a `dticks` command the set distance command will take precedence. Again the `xaxis` command will also set the format for the top x-axis.

When a graph is drawn the labels on the axes are assigned with each tick having an associated label, which should produce a graph with an easily readable scale. If for some reason it does not, or we wish to relabel the graph to draw attention to a particular region, then we may manually add the axes labels with the command `xplaces`. If, for example, we have a data set that is asymptotic to x=1 then, assuming we have sufficient resolution, we may write

```
xplaces 0 0.25 0.5 0.75 0.9 0.95 0.99
```

This will label the x-axis with the numbers 0, 0.25, 0.5, ..., 0.99, each number having a tick associated with it. These ticks will remain in position even if we set a tick spacing.

```
xplaces 0 0.25 0.5 0.75 0.9 0.95 0.99
xaxis nticks 6
```

The `xaxis nticks` command will not change the position of the labels on the bottom x-axis, we must remove the `xplaces` command to do that, the top x-axis will have 6 ticks along it.

If we are not satisfied with the default labels supplied by GLE then we can substitute them for ones of our own choosing.

```
xplaces 1 2 4 6 7
xnames "Monday" "Tuesday" "Thursday"
xnames "Saturday" "Sunday"
```

The `xnames` commands allows us to supply any text variable to replace the labels used in the text. It also, as in the example above, allows us to specify labeling with a non-uniform spacing. The `xplaces` command is not required, without it GLE will apply the labels to the default placings. If there is insufficient space on a single line for all the labels then separate `xnames` commands can be used for each individual label.

5.2.3 Fine tuning

There are a few remaining commmands that are useful for solving particular problems with the final graph.

The commands `xaxis nofirst` and `xaxis nolast` will remove the first or last labels on the x-axis, with similar commands for the y-axis. If the labels are overlapping then the use of this command can be easier than reformatting the entire axis.

`xaxis shift 0.2` will shift the labels on the xaxis by 0.2cm from the left to the right. This can be used when the labels refer to data between the labels rather than above them. Again an analogous command exists for the y-axis.

If we wish to add a grid to our graph then we may use a dedicated command `xaxis grid` to produce one. The command makes the x and y ticks as long and as tall as the graph itself. For example,

```
xticks lstyle 2
yticks lstyle 2
xaxis grid
yaxis grid
```

will produce a dotted grid underneath the data.

By default the text on the second y-axis will be written vertically upwards. If we wish to include the graph within another application then to stop text appearing upside down we may need to reverse this default setting. To do this we use the command `y2title "Axis title" rotate`, which will display the text written vertically downwards instead. The command is specific to the second y-axis.

5.2.4 Logarithmic scales

We can produce graphs with logarithmic scales on the x-axis, or the y-axis, or both. Logaritmic scales no longer have a constant spacing between the labels, so we should use the `xaxis nticks` command for changing the spacing between the ticks. There are also some slight differences in the use of smoothing routines with logarithmic scales.

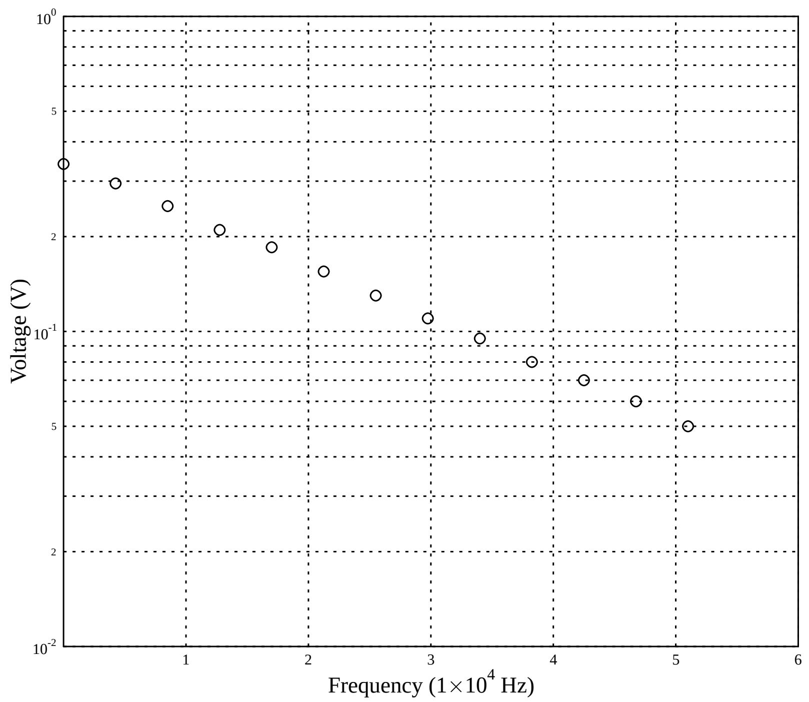


Figure 4b: Measured frequency dependence responce of target circuit

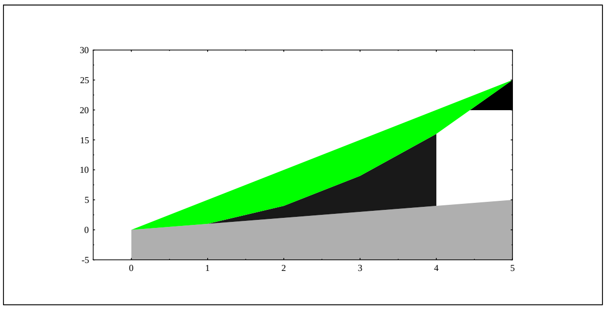
```
!log.gle - Demonstration of the use of logarithmic scales
size 14 12
amove 0 0

begin graph
nobox
size 14 12
data freq.dat                   !Reads data from freq.dat

yaxis log min 0.01 max 1 grid   !Formats the axis and adds a grid
xaxis min 0 max 0.0006 grid

xlabels hei 0.2
ylabels hei 0.2
xplaces 0.0001 0.0002 0.0003 0.0004 0.0005 0.0006
xnames "1" "2" "3" "4" "5" "6"  !Labels the axis

yticks lstyle 2                 !Changes the grid to a dotted line
ysubticks lstyle 2
xticks lstyle 2
```

```
xtitle "Frequency (1\times 10^{4} Hz)" hei 0.3
ytitle "Voltage (V)" hei 0.3

d1 marker circle msize 0.2      !Draws the data
end graph

amove 7 0.5                     !Adds a title
set just CC hei 0.3
text Figure 4b: Measured frequency dependence responce of target circuit
```

As we can see from above, the command `xaxis log` either on its own or with other x-axis commands will tell GLE to draw a logaritmic scale on the x-axis. A similar command, `yaxis log`, will produce a logarithmic scale on the y-axis.

5.2.5 Filling between the lines

Another possible formatting option is the ability to fill the areas between datasets, or between a dataset and an axis, with a solid block of colour.




```
!fill.gle - Demonstrates the options for filling between lines
size 16 8
amove 0 0
begin graph
      data fill.dat d1=c1,c2 d2=c1,c1
      fill x1,d2 color grey20
      fill d1,d2 color grey80 xmin 1 xmax 4
      fill y2,d1 color black ymin 20 ymax 25
      fill d1 color green
      end graph
```

The `fill` command instructs GLE to fill the lines immediately following the command, these can be either a dataset or one of the axes referred to with the labels, x1, x2, y1, and y2.

```
fill x2,d2 color green
```

This will fill the area between the dataset `d1` and the upper x-axis with a green colour. The fill style is clipped according to the line joining the points. At the moment this means that both the line and the filling appear as a jagged polygon; in later sections we will fit a smoothed line to the dataset, and in this case, the filling will also be smoothed. We can also fill between a dataset and the y-axis, though in this case, the filling will be clipped at the edge of the data and not necessarily at either y-axis. We can also specify a maximum and minimum range of values to fill between using the `xmin, xmax, ymin,` and `ymax` commands. It is normally a good idea to specify the limits, particularly when filling from the y-axis, as GLE may have trouble interpreting some of the fill commands if the function is undefined at any point.

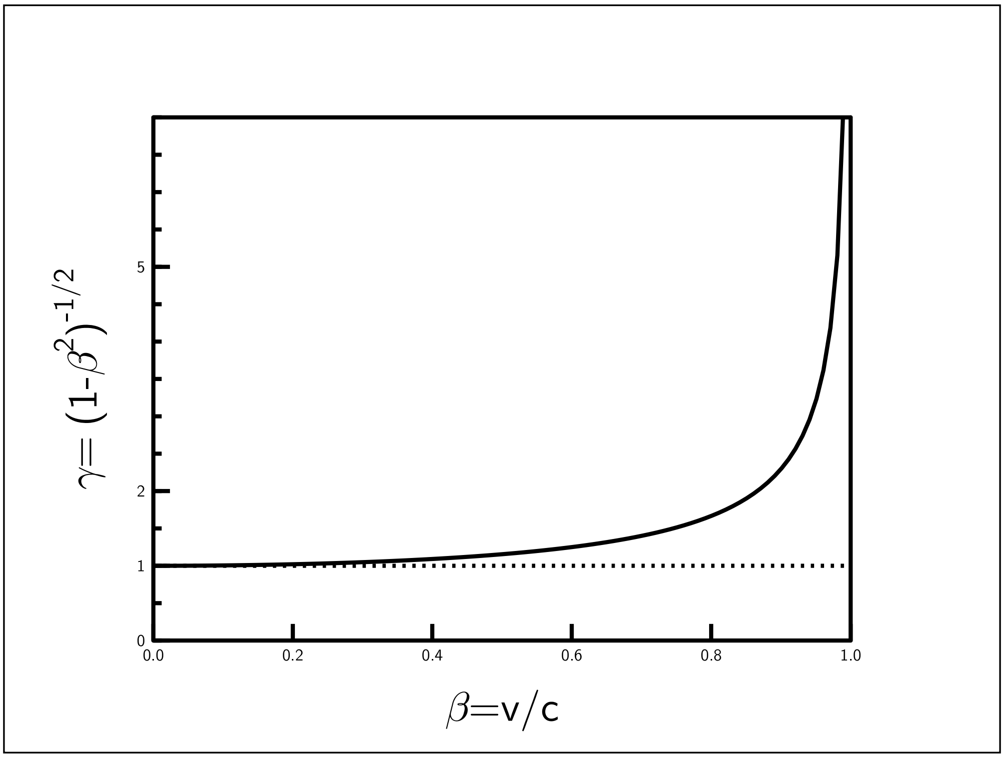
A final command allows us to fill between a data set itself,

```
fill d3 color blue xmin 4 xmax 40
```

In this case the maximum and minimum points are joined by a straight line to form one or more closed polygons. The fill colour is then applied to the inside of these shapes.

5.3 Plotting graphs from user defined formulae

Often it is necessary to reproduce the expected curve from a theory, or to demonstrate the behaviour of a function. For this GLE provides the ability to generate graphs from a given function as well as from sets of data.




```
!gamma.gle
size 12 9
```

```
set lwidth 0.05                  !Formats the text and line width
set font texcmss

begin graph
        size 12 9
        nobox                    !Defines data set from the value of gamma
        let d1 = 1/(sqrt(1-x*x)) from 0.001 to 0.999 step 0.01
        d1 line
        let d2 = 1.0 from 0 to 10 step 2        !Produces a comparison data set
        d2 line lstyle 2
        xaxis min 0.0 max 1.0 hei 0.8           !Formats the axis
        yaxis min 0.0 max 7.0 hei 0.8
        xplaces 0 0.2 0.4 0.6 0.8 1.0
        yplaces 0 1 2 5
        xlabels hei 0.2
        ylabels hei 0.2
        xtitle "\beta=v/c" hei 0.5 dist 0.3
        ytitle "\gamma = (1-\beta^2)^{-1/2}" hei 0.5 dist 0.3
        xticks length 0.2
        xsubticks off
        yticks length 0.2
        x2ticks off
        y2ticks off

end graph
```

We can define data sets from a mathematical function using the `let` command. We must also set the sample rate and range as we would do if we were defining a programming loop. GLE reads the data into the set at the rate and according to the same rules set out in the section on conditional logic. Any function that can be used in a programming loop can also generate data for a graph.

The graph itself is made up of a number of straight line segments, which is why the sampling rate must be set so high. In the next section we will look at ways of drawing smooth continuous curves through a section of points, though even this will require a high sample rate if we are to have an accurate graph.

We can also create data sets that are functions of other data sets. For example, if we wish to take the average of two data sets `d1` and `d2`,

```
data experiment.dat d1 d2
let d3 = (d1+d2)/2
d1 line lstyle 4
d2 line lstyle 4
d3 line lstyle 1
```

Which will draw both data sets `d1` and `d2`, and their average value `d3`.

Datasets can, and often are, created solely for the purpose of calculation and are never plotted on their own. A combination of data sets should be defined over the same range if they are to be successfully combined.

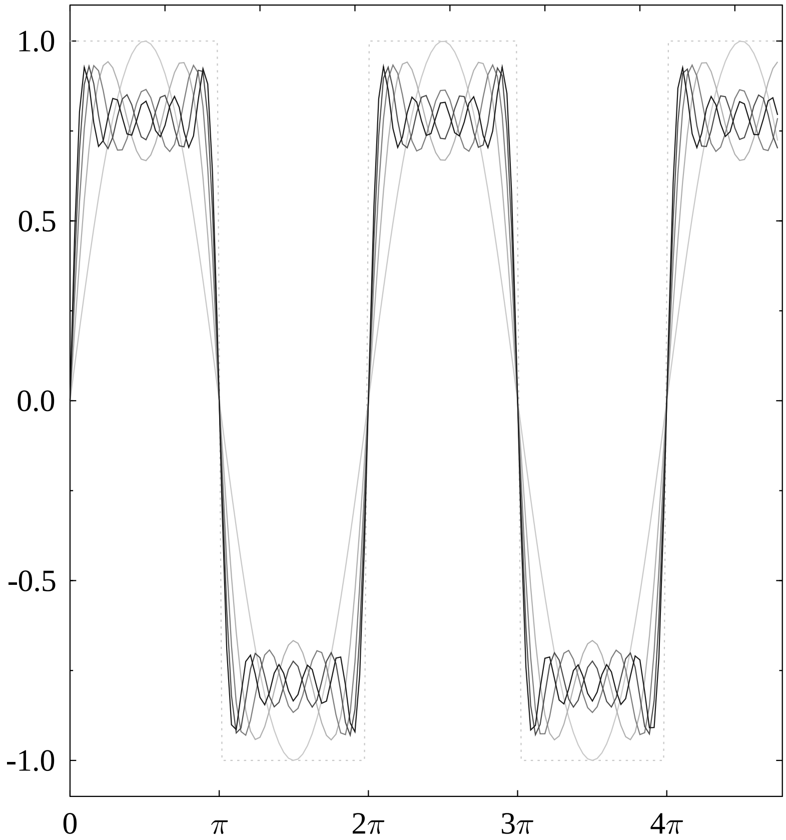


Figure 7b: Synthesis of a square wave from harmonic components

```
!Square.gle - Example of combining datasets

size 18 20
pi = torad(180)
amove 0 0

begin graph

yaxis max 1.1 min -1.1
xaxis max 15 min 0

let d1 = sin(x) from 0 to 15 step 0.1          !The fundamental sine wave
let d2 = sgn(d1)                               !The square wave
let d3 = sin(3*x) from 0 to 15 step 0.1        !Various harmonics over
```

```
let d4 = sin(5*x) from 0 to 15 step 0.1         !the fundamental
let d5 = sin(7*x) from 0 to 15 step 0.1
let d6 = sin(9*x) from 0 to 15 step 0.1

let d10 = d1+(1/3)*d3                           !We take linear combinations
let d11 = d10+(1/5)*d4                          !of the various frequencies
let d12 = d11+(1/7)*d5
let d13 = d12+(1/9)*d6

d1 line color grey10                            !These could also be plotted
d2 line lstyle 2                                !in different colors or line
d10 line color grey20                           !styles to make the difference
d11 line color grey40                           !clearer. A key may also help
d12 line color grey60
d13 line color grey80

xplaces 0 pi 2*pi 3*pi 4*pi                     !Labels the xaxis
xnames "0" "\pi" "2\pi" "3\pi" "4\pi"
xsubticks off

end graph

set just cc hei 0.3                             !Adds a title
amove 9 1
text Figure 7b: Synthesis of a square wave from harmonic components
```

Notice that we use the `sgn()` function to return a 1 if `d1` is positive and a -1 if `d1` is negative. We cannot define the function piecewise with the commands,

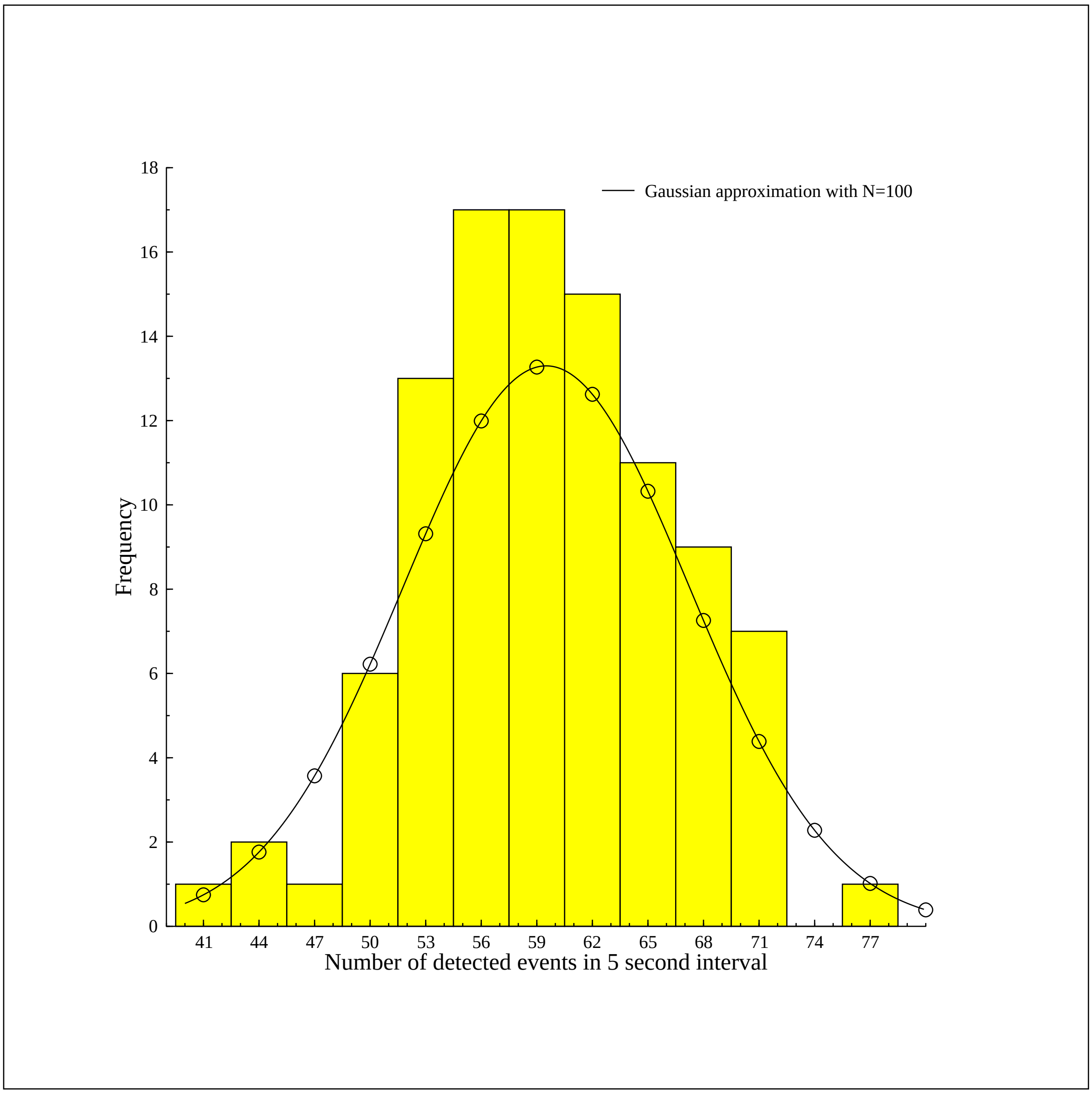
```
let d2 = 1 from 0 to pi step 0.001
let d2 = -1 from pi to 2*pi step 0.001
let d2 = 1 from 2*pi to 3*pi step 0.001
let d2 = -1 from 3*pi tp 4*pi step 0.001
```

Each use of the let command will wipe any data previously stored in the dataset, the above commands would produce a dataset that was defined to be -1 from 3{[pi]} to 4{[pi]} and undefined, that is zero, elsewhere. We could, of course, have defined four datasets each with part of the function and then added them together with the `let` command; though, in this case, we took a short cut and used the `sgn()` function.

5.4 Bar charts

The bar graph command is often complex, representing the number of possible options that we may choose to plot. It exists as a subset of the graph module so that we can plot both bar and line data on the same graph.

5.4.1 Combining lines and bars




```
!decay.gle - Demonstration of the bar command with data from radioactive decay

size 18 18
pi = 3.1415
mu = 59.51                !These are the parameters of the gaussian approximation
sigma = mu^0.5
N = 100                   !The number of timed intervals

amove 0 0
begin graph
nobox
                          !Sets our Gaussian into both "continuous" and
                          !discrete forms
let d1 = ((N/3)/(((2*pi)^(0.5))))*exp(-((x-mu)^2)/(2*(sigma^2))) from 40 to 80 step 0.1
```

```
let d2 = ((N/3)/(((2*pi)^(0.5))))*exp(-((x-mu)^2)/(2*(sigma^2))) from 41 to 90 step 3

data decay.dat d3=c1,c2         !Data taken from the experiment

d1 line                         !Plots the approxmation and the data
d2 marker circle
bar d3 width 3 fill yellow

key hei 0.3 pos tr nobox        !Adds a key
d1 key "Gaussian approximation with N=100"

                                !Formats the axis
xplaces 41 44 47 50 53 56 59 62 65 68 71 74 77
xaxis min 39 max 80 hei 0.3 dsubticks 1
yaxis min 0 max 18 hei 0.3
x2axis off
y2axis off

                                !Adds the title
xtitle "Number of detected events in 5 second interval"
ytitle "Frequency"
end graph
```

We have met much of the above before, the important command to notice is the `bar command`. The command instructs GLE to draw a set of bars of width 3 units, with the position and height determined by the values in dataset `d3`. The position is taken to be the midpoint of the graph so our data set must refer to the midpoint values. Unfortunately it is not possible to use a second data set for individual widths, so we must either settle for fixed widths or draw the chart as a combination of individual datasets, one for each different width.

The format commands that apply to the bar charts are fairly straightforward,

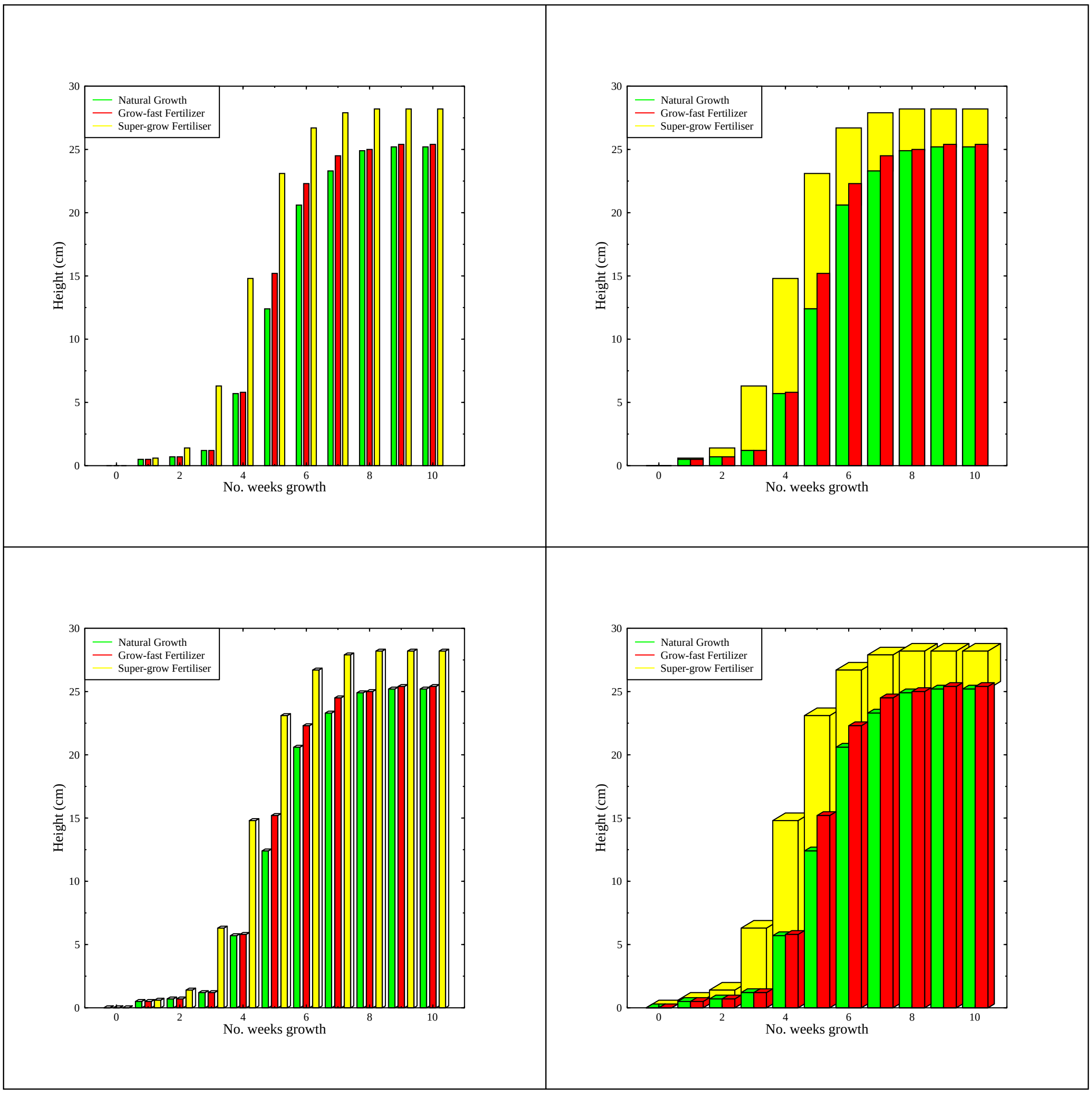
```
bar d1 width 2 fill blue color green
```

This will draw bars of width 2 with a green outline and filled with blue. In the following sections we shall see that there are other formatting commands that can produce more interesting effects.

5.4.2 Comparing data - grouped and 3D bar charts

We often need to present grouped data on a chart, to do this we can either display the bars side by side or stacked on top of each other. We look at an example which shows the various option available, including the ability to display a 3 dimensional representation of the data.

```
!bean.gle - Demonstrates the various formatting options for a bar chart
size 20 20

amove 0 10
```




```
begin graph
size 10 10
data bean.dat
                                !The first graph draws a set of bars side
                                !by side
bar d1,d2,d3 fill green,red,yellow

d4 line                         !These help with the generation of the key
d5 line
d6 line

xaxis hei 0.2
yaxis hei 0.2 min 0 max 30
```

```
xtitle "No. weeks growth"
ytitle "Height (cm)"

key pos tl hei 0.2              !The key uses the blank data sets - we can
                                !create more advanced keys using the key module
d4 key "Natural Growth" color green
d5 key "Grow-fast Fertilizer" color red
d6 key "Super-grow Fertiliser" color yellow
end graph


                        !The three following graphs have the same format and
                        !data as the previous - the only difference is the
                        !way the bars are plotted
amove 10 10
begin graph

size 10 10
data bean.dat
                        !This stacks one of the data sets ontop of the
                        !other two
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4
bar d3 from d1 fill yellow width 0.8
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4

d4 line
d5 line
d6 line

xaxis hei 0.2
yaxis hei 0.2 min 0 max 30

xtitle "No. weeks growth"
ytitle "Height (cm)"

key pos tl hei 0.2
d4 key "Natural Growth" color green
d5 key "Grow-fast Fertilizer" color red
d6 key "Super-grow Fertiliser" color yellow
end graph

amove 0 0

begin graph
size 10 10
data bean.dat
```

```
                        !This changes the bars into 3D blocks
bar d1,d2,d3 fill green,red,yellow  width 0.2,0.2,0.2 dist 0.3 3d 0.5 0.3

d4 line
d5 line
d6 line

xaxis hei 0.2
yaxis hei 0.2 min 0 max 30

xtitle "No. weeks growth"
ytitle "Height (cm)"

key pos tl hei 0.2
d4 key "Natural Growth" color green
d5 key "Grow-fast Fertilizer" color red
d6 key "Super-grow Fertiliser" color yellow
end graph

amove 10 0
begin graph
size 10 10
data bean.dat

                        !We can also stack a 3D bar graph
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4 3d 0.5 0.3 side green,red
bar d3 from d1 fill yellow width 0.8 3d 0.5 0.3 side yellow top yellow
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4 3d 0.5 0.3 side green,red top green,red

d4 line
d5 line
d6 line

xaxis hei 0.2
yaxis hei 0.2 min 0 max 30

xtitle "No. weeks growth"
ytitle "Height (cm)"

key pos tl hei 0.2
d4 key "Natural Growth" color green
d5 key "Grow-fast Fertilizer" color red
d6 key "Super-grow Fertiliser" color yellow
end graph
```

To group bars together on the same line we list the appropriate datasets under the same `bar` command.

```
bar d1,d2 fill green,red width 1,2 color black,red
```

The datasets are separated by a single comma (without space). Each option that may be specified for a single bar may be specified for grouped bars, but we separate the options with a comma and in the same order as the datasets are listed. The example above fills the first dataset, `d1`, in green with a black outline and a width of 1 unit on the x-axis.

We can also stack single bars on top of each other, and if we wish, stack grouped bars on top of each other too.

```
bar d1,d2
bar d3,d4 from d1,d2
bar d5,d6 from d3,d4
```

This will draw the datasets `d1` and `d2` next to each other, then `d3` and `d4` on top of these two, and finally, `d5` and `d6` at the very top. The bars are plotted sequentially so GLE will overwrite anything that has already been plotted. In the program `bean.gle` we wrote,

```
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4
bar d3 from d1 fill yellow width 0.8
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4
```

This will plot the datasets `d1` and `d2`, then partially obscure `d2` as `d3` is drawn (if you have a slow enough postscript viewer then you can watch this happening). We need to redraw the first two data sets if we keep the commands in this order. It is not however necessary to draw the two data sets first; we can remove the first bar command to leave,

```
bar d3 from d1 fill yellow width 0.8
bar d1,d2 fill green,red width 0.4,0.4 dist 0.4
```

This will accomplish the same thing, although in a slightly less intuitive order.

3D bar charts are subsection of the `bar` command; to use these we tell GLE that we require a 3D plot by adding the `3D` command on the same line as the `bar` command.

```
bar d1 3d 0.5 0.3
```

The two numbers following the `3D` command define the x and y vectors used to draw the receding lines, given in terms of a fraction of the width of one bar. The pair of vectors control how '3D' the graph appears: small fractions give a very subtle effect, larger ones can cause the bars to overlap.

Charts with 3 dimensional bars can be stacked, grouped, and formatted in exactly the same way as the ordinary 2 dimensional variety. Some of these effects are best used with caution since sometimes the results can be confusing, both for the programmer and for the final audience. In addition to these, there are several formatting options that apply specifically to the 3 dimensional case. A typical command may have the form,

```
bar d2,d3 3d 0.2 0.2 side green,blue top green,blue fill green,blue
```

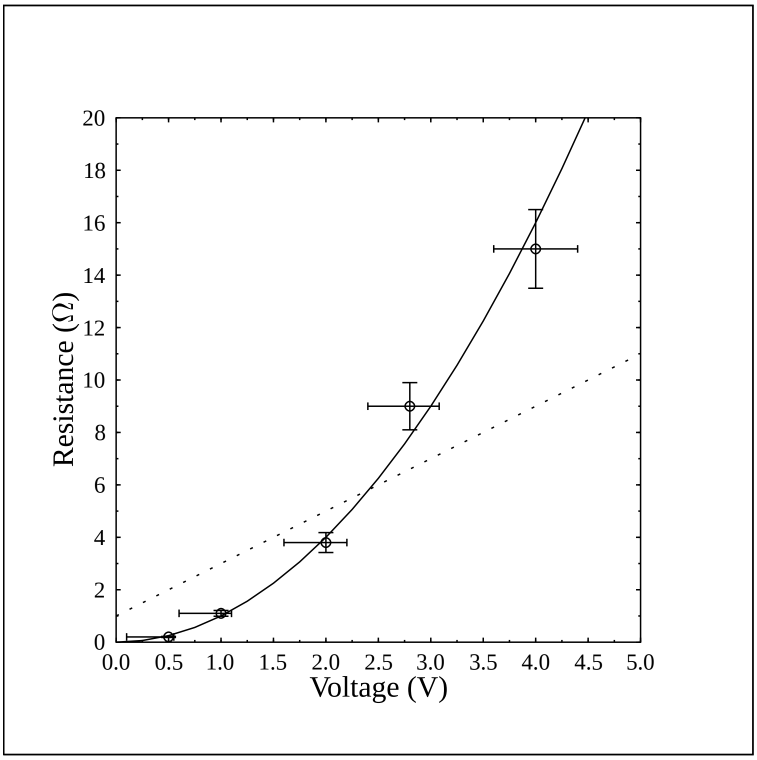
The `side` command gives the colour to be used for the sides of the bars, if there are more than one then we list the various colours separated by a comma as in the 2 dimensional case. Similarly the `top` command gives the colours to be used for the tops of the bars. We can replace the `top` command with the `notop` command which leaves the top of the bar open for use with stacked bar 3D bar charts.

5.5 Scientific graphs

GLE is ideal for producing technical graphs for inclusion in reports and papers, with this in mind we look more closely at some of the graph commands that are useful to scientists.

5.5.1 Treatment of errors

GLE allows for the errors in a set of data-points to be displayed with error bars that are taken from a second dataset, or from a certain fractional or absolute error in each data point.




```
!error.gle - Example of the use of error bars
size 10 10

begin graph
data error.dat
let d2 = x^2 from 0 to 50       !Two different possible models
let d3 = 2*x+1 from 0 to 5
d3 lstyle 4
d2 line smooth
d3 line smooth

xaxis min 0 max 5
yaxis min 0 max 20
xtitle "Voltage (V)"
ytitle "Resistance (\Omega)"    !Notice the difference between Omega and omega
```

```
                                !Adds the markers and error bars
d1 marker circle err 10% errwidth 0.2  herrleft 0.4 herrright 10%

end graph
```

There are two separate error bars, horizontal and vertical, which can can be formatted individually and independently. We shall look mainly at the command options for the vertical bars, but the horizontal bars can be formatted by very similar commands.

The simplest possible case would be if we had an absolute error in the y coordinate for each data point. We assume, to start with, that the error is symmetrical such that the mean value is at the centre of the two error bars.

```
d1 err 10%
```

This will produce an error bar that extends above the data point for 10% of the absolute y value, and a similar distance below. The command `herr 10%` will do the same for the horizontal bar, except the error will be 10% of the absolute x value.

If we do not have a symmetric distribution of errors then we may specify individual errors for the up and down error bars.

```
d1 errup 10% errdown 5%
```

This will take 10% of the y value up, but only 5% down. For the horizontal bar we use the commands `herrleft 5%` and `herrright 10%`.

We can also specify absolute errors using the same set of commands,

```
d1 err 0.2
d1 errup 0.2 errdown 0.2
```

These are commands that accomplish the same thing, the error bars extend for 0.2 y units above and below the actual data point.

If we wish to we can control the size of the lines that terminate the error bars. The commands `errwidth 0.5` and `herrwidth 0.5` will set both the horizontal and vertical lines to 0.5 in their respective units.

Errors are an integal part of many technical graphs, and there are occasions when we need more control over the displayed errors. We may, for example, wish to use a separate program to calculate the error on each individual data point. If this is the case then we can use a second dataset to store the error values and produce error bars that are unique to each point.

```
!Planck.gle - Demonstration of errors defined by separate dataset
size 12 12

amove 0 0
begin graph

yaxis min 6.623 max 6.6265 hei 0.2
```

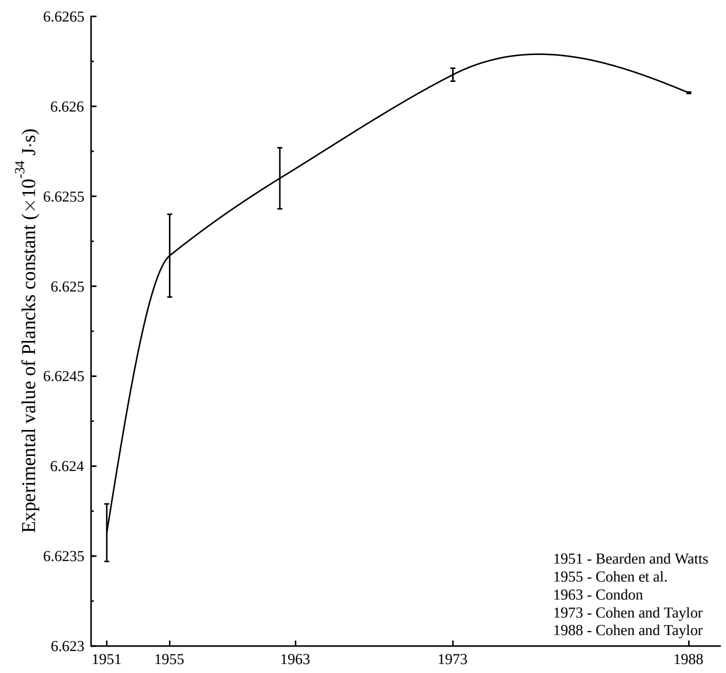


Fig 4: How constant is constant?

```
xaxis hei 0.2

data planck.dat d1=c1,c2 d2=c1,c3       !Reads the data and errors into
                                        !two different data sets

d1 line smooth err d2                   !Uses the second data set to
                                        !Define the error on the first

ytitle "Experimental value of Plancks constant (\times 10^{-34} J\cdot s)"
xplaces 1951 1955 1963 1973 1988
xsubticks off
x2axis off
y2axis off

key pos br nobox hei 0.2                !We use undefined datasets to add
d3 key "1951 - Bearden and Watts"       !a key - Notice that the order is
d4 key "1955 - Cohen et al."            !the same as the number of the
d5 key "1963 - Condon"                  !dataset, not the order on the page
d6 key "1973 - Cohen and Taylor"
d7 key "1988 - Cohen and Taylor"
```

```
end graph

amove 3 0.75                          !Adds a title
set hei 0.3 font rmb
text Fig 4:
amove xend() yend()
set font rm
text \glass How constant is constant?
```

This is the separate data file, the figure given in the third column is the absoulte error in each measurement. For example, 6.2363 ± 0.00016 Js.

```
1951  6.62363    0.00016
1955  6.62517    0.00023
1962  6.62560    0.00017
1973  6.626176   0.000036
1988  6.6260755  0.0000040
```

All the error commands can be used with a separate data set, except instead of specifying an absolute or fractional error in every point we can now specify the individual errors.

```
d1 errup d2 errdown d3 herr 10%
```

This will give a 10% error in the x coordinate and an error in the y coordinate dictated by the datasets d2 and d3.

5.5.2 Curve fitting

GLE contains two separate routines that allow for a discrete set of data to be connected by a smooth curve. In the simplest possible case GLE will fit a third degree polynominal piecewise to a set of data points. To do this we add the `smooth` command.

```
d3 line smooth
```

The command is useful if we either do not know the simple form of the equations that produced the data points, or we know that a simple form does not exist. By using a high sampling rate and the `smooth` command we can approximate most functions to an arbitarily high degree. GLE also provides a number of fitting routines for commonly used functions. To use the routines we must define another dataset which contains the approximation data; we can then plot the fitted dataset just as we would with any other set of points.

```
let d4 = logefit d3 from 0 to 100
```

This will create data in `d4` that best represents a logarithmic (base e) fit of the data in `d3`. There are three other fitting routines available, `log10fit` uses a base 10 logarithm, `linfit` a straight line, and `powxfit` a power law.

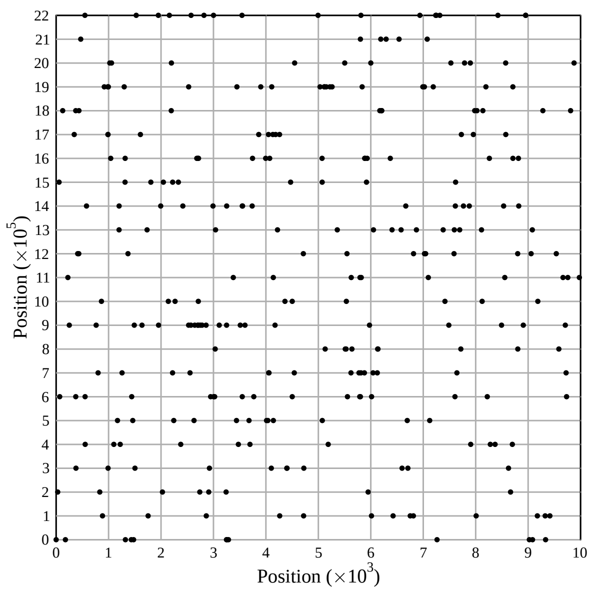
The range of the data can either be defined explicitly as above, or we can choose the same range as the original dataset with one of the following commands,

```
let d4 = linfit d3 limit_data
let d4 = linfit d3 limit_data_x
let d4 = linfit d3 limit_data_y
```

`limit_data_x` gives the fitted data, `d4`, the same range as the x values in `d3`, `limit_data_y` gives the fitted data the range of the y values in `d3`. `limit_data` will give `d4` a range that can extend from the minimum x or y value to the maximum x or y value, whichever option gives the greatest range.

5.5.3 Dealing with very large data sets

Although the steady evolution of computer hardware has rendered it less and less important GLE provides a number of commands for dealing with very large data sets where the physical memory of the computer becomes an issue.



Distribution of palendromic sequences in viral DNA

```
!dna.gle - Demonstrates the use of the bigfile command for use with large datasets
size 10 10

begin graph             !dna.dat is a list of the positions of palendromic
                        !sequences in a viruses DNA
d1 bigfile "dna.dat,1,2" marker dot msize 0.2

                        !The position is represented by two columns, one is
                        !the position single units, the other in tens of
                        !thosands
xaxis min 0 max 10000 grid hei 0.2
```

```
yaxis min 0 max 22 grid dticks 1 hei 0.2
yticks color grey20
xticks color grey20
ysubticks off
xnames "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
xtitle "Position (\times 10^{3})"
ytitle "Position (\times 10^{5})"
end graph

amove 5 0.5
set just cc hei 0.3
text Distribution of palendromic sequences in viral DNA
```

The data file has the form,

```
(...)
5151 19
5221 19
5262 19
5835 19
6992 19
7022 19
7191 19
8195 19
8709 19
1023 20
1056 20
2198 20
4548 20
(...)
```

The first column represents the position in single units the second in tens of thousands so that the first sequence is 195151.

The dataset in the example is only a few thousand lines long, this may have caused trouble a few years ago but modern PCs are perfectly capable of dealing with datasets far in excess of this. Nevertheless it is occasionally necessary to deal with extremely large data sets. If the DNA above was taken from a human and not a virus we would have wanted to reduce the pressure on computing power as much as possible. The `bigfile` command used above saves on memory by plotting the data point by point straight from the file instead of being reading the data into memory and then plotting.

```
d1 bigfile "data.dat,2,3" autoscale marker circle line
```

This will plot columns 2 and 3 from the file `data.dat` with circular markers and joined with straight lines. The `autoscale` command pre-reads the data and scales the axes, before re-reading it and plotting. This can be slow, but is useful if we do not know the exact extent of the data. Without the command we must specify the minimum and maximum values of the x and y axes.

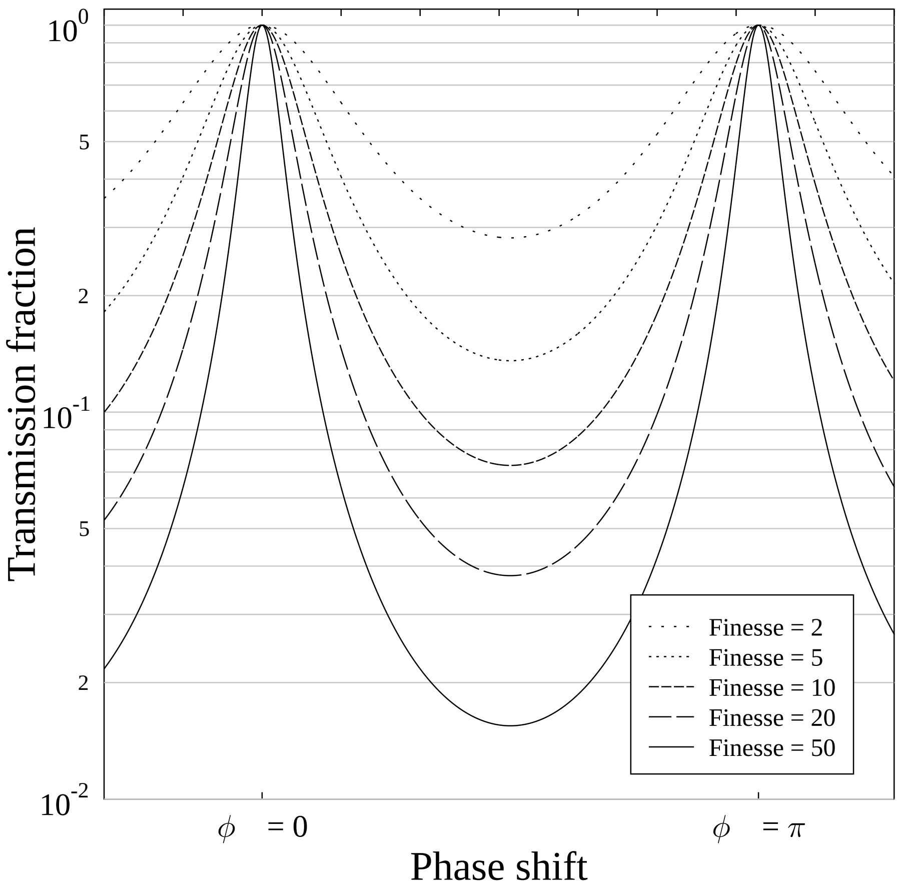
Virtually all of the graph module commands will work with the `bigfile` option. It is, however, impossible to draw bar charts using the command, or to use the `let` command to create new datasets.

5.6 The key module

Although it is perfectly possible to draw a key from within the graph module, GLE provides a dedicated set of commands that allow the user to draw up a key of points for a graph.

5.6.1 A basic key

Each key follows the graph module to which it refers; we begin the module with a `begin key` command, and end it with an `end key` command. In between these two lines a special command set applies.



Transmission fraction of the Fabry-Perot Etalon

```
!etalon.gle - Transmission function of the Fabry-Perot Etalon to demonstrate
                !the use of the key module
size 18 18
pi = 3.1415
```

```
f1 = 2          !These define the finesse of each line
f2 = 5
f3 = 10
f4 = 20
f5 = 50

amove 0 1
begin graph
nobox
size 18 18

                !This is the transmission function for the etalon
let d1 = 1/(1+(4*(f1)/pi)*((sin(x))^2)) from -2*pi to 2*pi step 0.001
let d2 = 1/(1+(4*(f2)/pi)*((sin(x))^2)) from -2*pi to 2*pi step 0.001
let d3 = 1/(1+(4*(f3)/pi)*((sin(x))^2)) from -2*pi to 2*pi step 0.001
let d4 = 1/(1+(4*(f4)/pi)*((sin(x))^2)) from -2*pi to 2*pi step 0.001
let d5 = 1/(1+(4*(f5)/pi)*((sin(x))^2)) from -2*pi to 2*pi step 0.001

                !We use a logarithmic scale to increase the contrast between
                !the different values of finesse
yaxis min 0.01 max 1.1 log grid
xaxis min -1 max 4

xsubticks off
yticks color grey10
ysubticks color grey10
xaxis hei 0.5
yaxis hei 0.5
xplaces 0 pi
xnames "\phi \: = 0" "\phi \: = \pi"
xtitle "Phase shift"
ytitle "Transmission fraction"


d1 line lstyle 4        !These commands draw the various lines
d2 line lstyle 2
d3 line lstyle 3
d4 line lstyle 5
d5 line lstyle 1
end graph

set just CC hei 0.5     !Adds a title
amove 9 1
text Transmission fraction of the Fabry-Perot Etalon

amove 11 4              !We move to where we want to draw the key
```

```
begin key               !The key module
hei 0.4
offset 0.1 0.1
text "Finesse = 2" lstyle 4
text "Finesse = 5" lstyle 2
text "Finesse = 10" lstyle 3
text "Finesse = 20" lstyle 5
text "Finesse = 50" lstyle 1
end key
```

The key module is an extension of the `key` command we have used previously. Within the module we may specify any formatting options that we used with the single line command, plus a few extra ones that we could not.

```
begin graph
...
end graph
begin key
hei 0.5
positon tr
text "This is the first title" marker dot msize 0.2 mscale 0.5 color red fill blue lstyle 2
end key
```

This will produce a key in the top-right hand corner of the previous graph. The `text` command uses the quoted text string as first line of the key, the qualifiers following this title represent the format of the line. If we add a `marker command` then it will be displayed next to the title, with a size defined by the `msize` command. The `mscale` command is similar except the size is taken to be a fraction (0.5 in the example above) of the default size. If we define a line style with the text command then a line in the appropriate style will also appear next to the title. The `fill` command adds a coloured box next to the title. The colour of the text, the line, and the marker are all set by the `color` command.

5.6.2 The key syntax

The syntax of the key module is a little different to the rest of the GLE commands we have dealt with. We shall classify commands into two types: those which relate to global features of the key module, and those which relate to a specific dataset. The global commands in the key module are,

`offset, nobox, hei, pos`

Within the key module these can appear anywhere except on the start of a line that defines a new dataset.

```
begin key
        offset 2 3 hei 2
        text "abc" nobox
        nobox text "abc" !This line is not valid
end key
```

The single dataset commands are,

`text, marker, msize, mscale, color, fill, lstyle, line, lwidth`

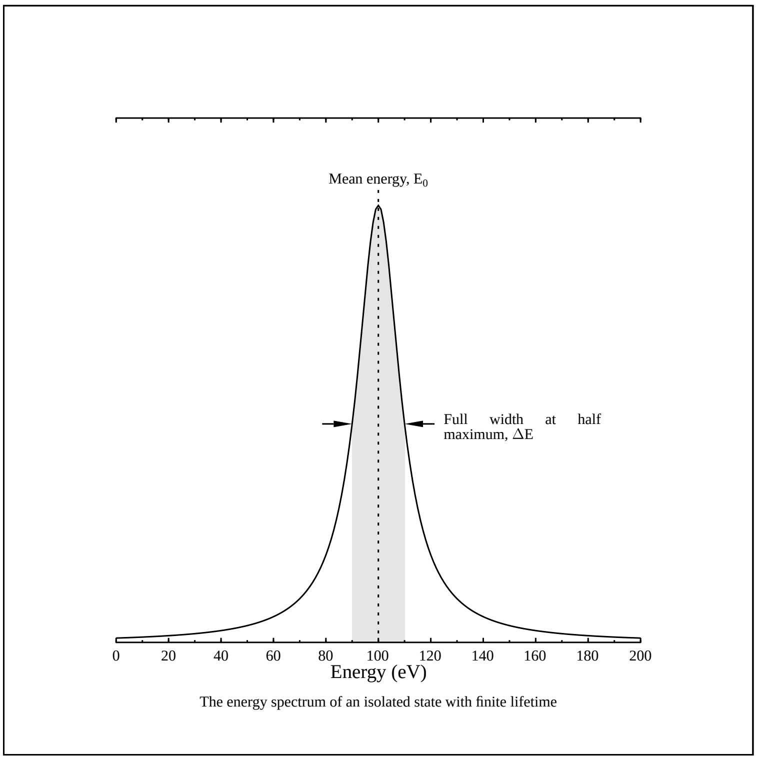
These must appear together on a single line corresponding to a single dataset, within this line the order is not important.

```
begin key
        marker circle text "abc" lstyle 2
        lstyle 2 marker square
end key
```

The (entirely arbitrary) convention is to put the global commands at the start of the module, and follow them by a single line for each line in the key.

5.7 Integrating graphs and diagrams

A single graph may form part of a larger diagram, even if that diagram contains only a few additional notes or titles. To make the transition between lines drawn inside the graph module and those drawn outside as seamless as possible we may use two commands, `xg()` and `yg()`, that refer to a point on a graph while we are outside the main graph module.




```
!lorentzian.gle - Demonstrates the xg() and yg() commands
size 10 10
E = 100                 !This is the energy, E, and width, W, of the state
W = 20

begin graph
```

```
!These two datasets produce the Lorentzian profile
let d1 = (x-E)/(W/2) from 0 to 200 step 2
let d2 = 1/(1+((d1)^2))

                                                !This formats the axis
yaxis min 0 max 1.2 hei 0.2
yaxis off       !Remove this line to add the y-axis
xaxis min 0 max 200 hei 0.2
yplaces 0 0.5 1
xtitle "Energy (eV)"
title "The energy spectrum of an isolated state with finite lifetime" hei 0.2 dist -8

d2 line smooth          !Draws the profile and fills in the width at half maximum
fill x1,d2 xmin 90 xmax 110

end graph

set hei 0.5                             !Labels the FWHM
amove xg(90) yg(0.5)
rline -0.4 0 arrow start
amove xg(110) yg(0.5)
rline 0.4 0 arrow start
set hei 0.2

amove xg(125) yg(0.5)
begin text width xg(190)-xg(130)
Full width at half maximum, \Delta E
end text
                                        !Labels the mean energy of the state
set just center lstyle 2
amove xg(100) yg(0)
aline xg(100) yg(1.05)
text Mean energy, E_{0}
```

The commands `xg(a)` and `yg(b)` return the absolute x and y coordinates of a point (a,b) measured in the x and y units of the most recently drawn graph. We may use these two functions to label points of interest on a graph.

```
begin graph
  (...)
end graph
amove xg(80.6) yg(1.2)
rline 0.5 0.25
text \glass \: Production of W^{+}
```

This will move to the point (80.2,1.2) on the graph, where the coordinates are in the same units units the x- and y-axis, and add a label. We may also join two points on a graph using these commands, or use them to format text. One of the uses may be to add complex mathematical curves, particularly those in cylindrical coordinates, to a set of axes.

6 More complex examples

Having covered the majority of the commands needed to use successfully GLE we now turn to some of the more specialized commands that may be useful in some applications.

6.1 Adding existing graphics to a GLE script

In many cases we may wish to include an existing graphic within a GLE file. It may be a supplied company logo or advertisement, or it may be a file that has been drawn previously with GLE or another graphics package. GLE can add graphics at any point on the page, provided the graphic files are in one of two common formats, Postscript or TIFF.

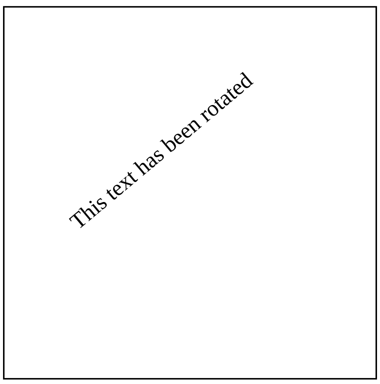
6.2 Coordinate transforms

Within a GLE program it is often possible to denote a section of code that is referred to a different coordinate system to take advantage of a symmetry within the diagram. The coordinate transforms all begin with a `begin` command and end with an `end` command, everything in between these two lines is treated within the separate system. At the `end` command GLE restores both the reference point and graphic state, including line styles and font heights, that was present before the `begin command`.

We have already met the most basic coordinate transform, that of a translation. We used the `begin origin` transform in the super-ellipse, it allows us to refer to an origin that differs from the origin of the page. A useful command when we have a subroutine or need to draw something with `amove` and `aline`, we can reposition the object just by moving to a different point before the `begin origin` command.

6.2.1 Rotations

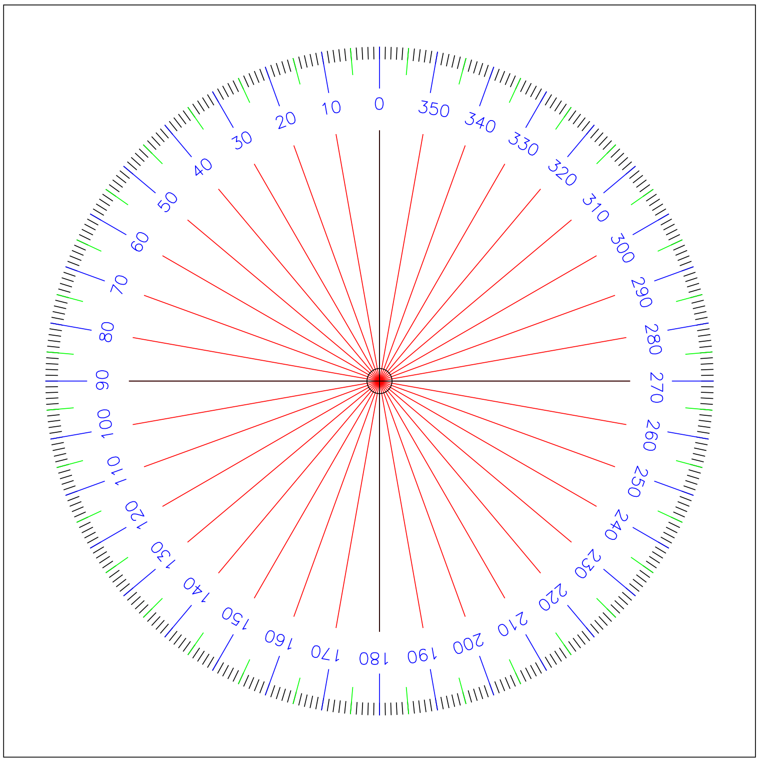
The `begin rotate` command allows us to draw an object then rotate it by an angle.



```
!rotate.gle - Rotates some text
size 5 5
set hei 0.3
amove 1 2
begin rotate 40
        text This text has been rotated
end rotate
```

This will rotate the given text by an angle of 40° anti-clockwise.

We look at a more concrete example that incorporates both of the above techniques and also produces a useful image.




```
!protrac.gle - Demonstrates the begin rotate and origin commands

size 18 18

set font plsr   !Defines the variables for easy alterations
set hei 0.4
a=8
b=0.3
c=1
d=0.65
f=6
g=6
```

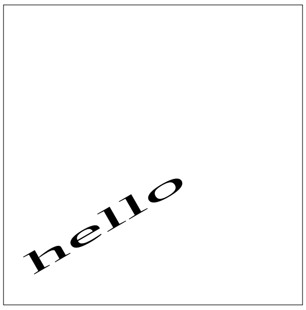
```
amove 9 9

begin origin               !Sets the origin to 9 9

        for i=0 to 359     !Draws the small unit angle ticks
        begin rotate i
                amove 0 a
                rline 0 -b
        end rotate
        next i

             !Draws the longer marks and labels every 10{deg}        set color blue
        for i=0 to 350 step 10
        begin rotate i
                amove 0 a
                rline 0 -c
                if i>0 then
                        rmove -0.2 -0.5
                        else
                        rmove -0.1 -0.5
                        end if
                write i
                 end rotate
         next i

             !Draws the small 5{deg} marks
        set color green
        for i=5 to 355 step 10
        begin rotate i
                amove 0 a
                rline 0 -d
                end rotate
        next i

             !Draws the long central lines every 10{deg}        set color red
        for i=0 to 350 step 10
        begin rotate i
                amove 0 0
                aline 0 g
                end rotate
        next i

             !Draws a black circle in the middle
        set color black
        amove 0 0
        circle 0.3
        amove -f 0
```

```
        aline f 0
        amove 0 -f
        aline 0 f
        amove 0 0

        end origin
```

We can see that the use of the rotate command greatly simplifies the drawing of the protractor - it is only neccessary to draw the lines and labels on a single straight line, the rotate command takes care of the rest of the shape.

6.2.2 Scaling

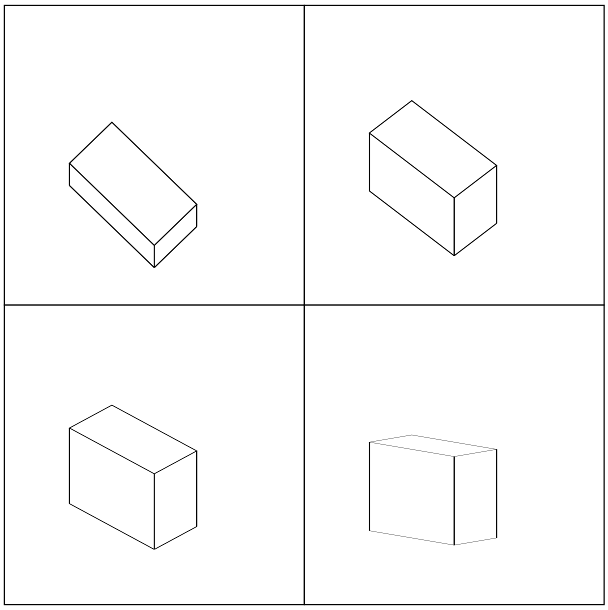
In the same way as we rotated objects we may also scale them in the x and y directions,




```
!scale.gle - demonstrates the scale command
size 10 10
amove 1 1
begin scale 3 1
        begin rotate 30
                text hello
                end rotate
        end scale
```

The numbers following the `scale` command represent the scaling factors in the x and y directions. A value of 1 will leave the object unchanged, greater than 1 will increase the size, less than 1 will decrease it. The example above rotates the text by 30°and then stretches it by a factor of 3 in the x direction. Notice that the scale and rotate operations are entirely independent of each other, so we can only scale a rotated object along its original axes. In the example above we talk of stretching along the 'x direction', but this direction is actually orietated at 30° to the x axis.

We can use the `scale` command to create the impression of a three dimensional object. The geometry behind these images, particularly if we wish to view an object from an arbritrary choice of angles, can get quite complex, and the independence of the `scale` and `rotate` commands can make seemingly simple shapes quite difficult to draw. The example below demonstrates a simple use of the `scale` command applied to a simple cuboid. Notice that to simplify the mathematics we restrict the viewing to a single plane along the diagonal of the object. More complex examples can be achieved but it must be said a dedicated 3-dimensional CAD package may, in this case, prove more fruitful.



```
!cube.gle - Demonstrates the use of the scale command to generate 3D objects
size 10 10
pi=3.1415

sub top a b                     !This draws the top of each cuboid
        gsave
        x1=a*cos(pi/4)
        y1=a*sin(pi/4)
        x2=b*cos(pi/4)
        y2=b*sin(pi/4)
        save top1
        rline x1 y1
        save top2
        rline -x2 y2
        rline -x1 -y1
        save top3
        rline x2 -y2
        grestore
end sub

sub bottom a b                  !This draws the bottom of each cuboid
        gsave
        x1=a*cos(pi/4)
        y1=a*sin(pi/4)
        x2=b*cos(pi/4)
        y2=b*sin(pi/4)

        gsave
        save bottom1
```

```
        rline x1 y1
        save bottom2
        grestore
        rline -x2 y2
        save bottom3
        grestore
end sub

sub side                        !This joins the top and bottom of a cuboid
        join top1 - bottom1
        join top2 - bottom2
        join top3 - bottom3
end sub

sub drawcube ang        !This combines the cuboid drawing into a single routine
                        !viewed from an angle theta above the bottom plane
begin scale 1 cos(ang)

@top 1 2
rmove 0 -1.5*tan(ang)
@bottom 1 2
end scale
@side

end sub


amove 0 0                       !Draws the cuboid from various angles
box 5 5
rmove 2.5 0.5+2*sin(1)
@drawcube 1
amove 0 5
box 5 5
rmove 2.5 0.5+2*sin(0.25)
@drawcube 0.25
amove 5 0
box 5 5
rmove 2.5 0.5+2*sin(1.4)
@drawcube 1.4
amove 5 5
box 5 5
rmove 2.5 0.5+2*sin(0.70)
@drawcube 0.70
```

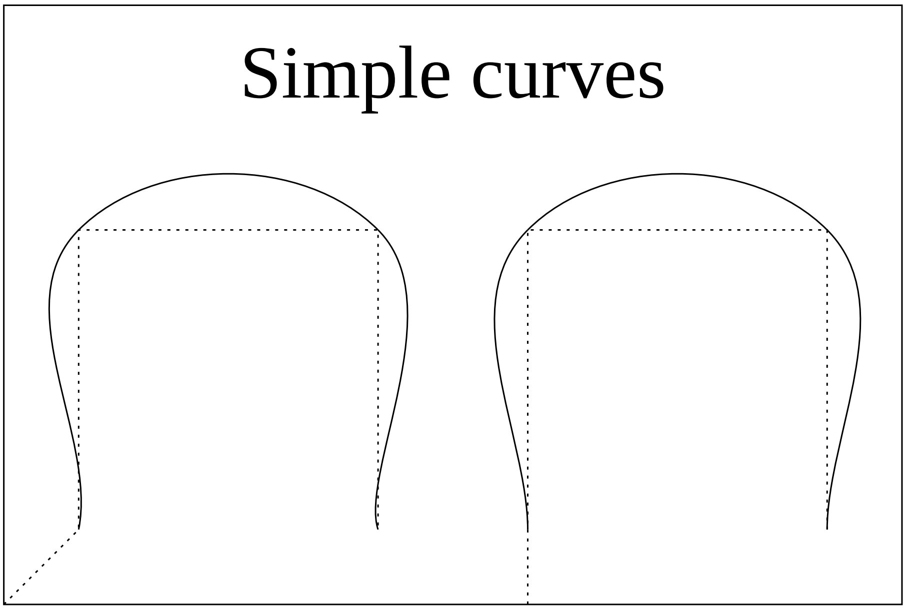
Generally, the apperance of 3D images such as this can be improved by judicious use of colours and shading. To add shading to anything other than a rectangle we need to know how to define an object, this is covered in the section on path drawing.

6.3 Mathematical curves

Technical drawings may require curves that have a well defined mathematical shape. GLE has a set of commonly used curves for this purpose, although it may sometimes be neccessary to add a particularly unusual curve using the graph module.

6.3.1 A simple curve

The `curve` command will draw a smooth line through any number of points, with an initial and final gradient that must also be supplied by the programmer.



```
!curve.gle - demonstrates the use of the curve function
size 12 8

set lstyle 2

aline 1 1       !Draws the points the curve follows
rline 0 4
rline 4 0
rline 0 -4

set lstyle 0
amove 1 1       !The first and last two numbers represent the gradient,
                !the others are the relative position of the guide points
curve 1 1 0 4 4 0 0 -4 1 1

amove 7 0
set lstyle 2
```

```
rline 0 5
rline 4 0
rline 0 -4

set lstyle 0
amove 7 1       !Another curve with different initial and final gradients
curve 0 1 0 4 4 0 0 -4 0 -1

amove 6 7
set just cc
text Simple curves
```

The curve command has the following syntax,

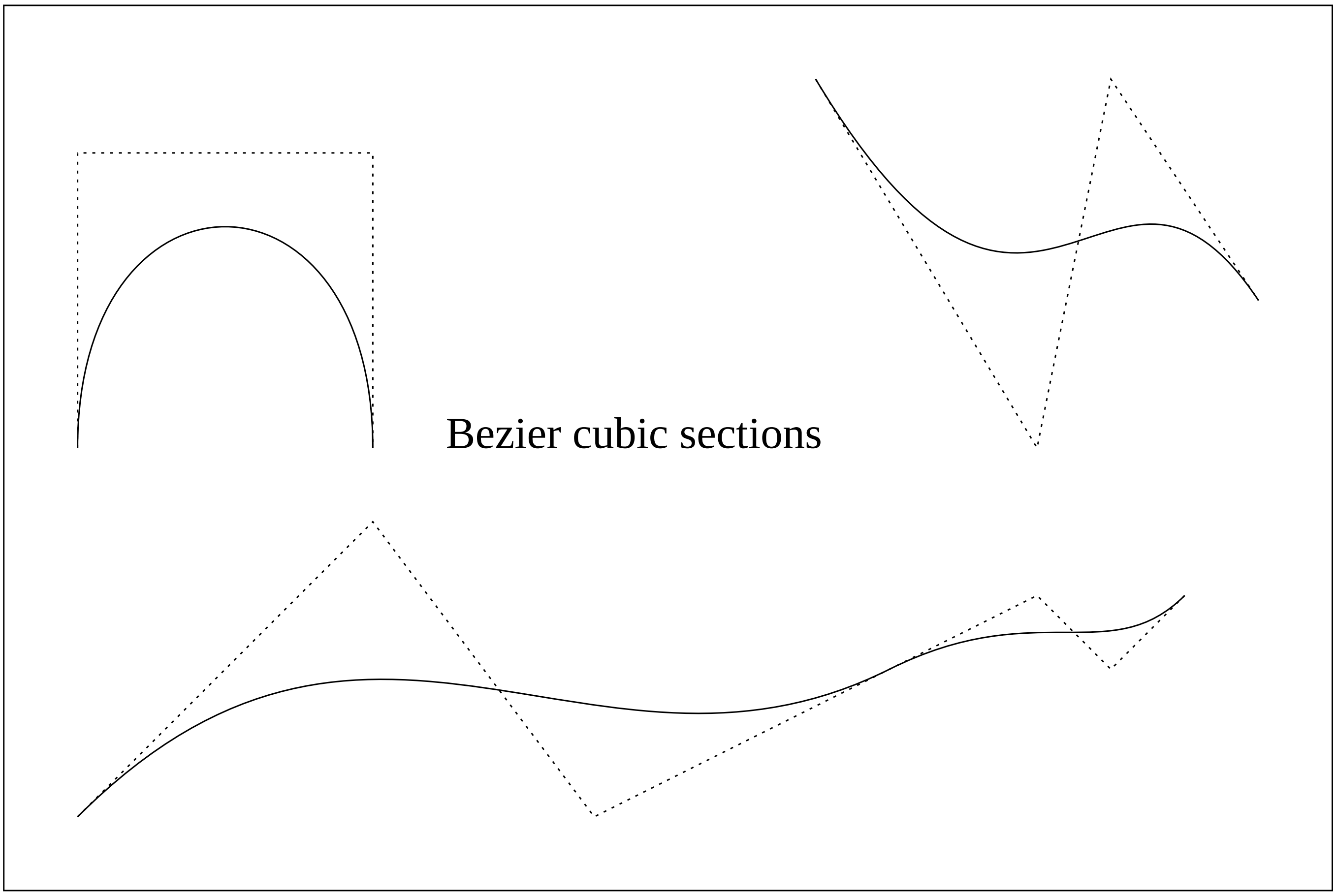
```
curve 2 1 1 1 1 2 -1 -1
```

The curve is drawn starting from the initial coordinates with a gradient defined by the first two values following the `curve` command. In this case (2,1) represents an initial gradient of two units along the x axis and 1 unit up the y axis or an elevation of arctan(1/2) above the x axis. With the exception of the final pair, the numbers following the initial gradient represent the points the curve must pass through. They are given in terms of relative position vectors from the last point so (1,0), (0,1), (-1,0), and (0,-1) would trace out a square. The final pair of numbers gives the gradient the curve is to have as it passes through the final point.

6.3.2 Bezier cubic sections

Bezier sections are mathematical curves that were first used for car bodywork design in the 70s. They have a number of neat mathematical properties that make them particularly appropriate for computer graphics. Most of the curve drawing commands in graphics packages are based around some sort of Bezier section, and the design of computer fonts also relies heavily on them. GLE has a command that will draw a cubic (3rd degree) Bezier curve to a set of four control points.

```
!bezier.gle - Demonstrates Bezier cubic sections
size 18 12

amove 1 6
set lstyle 2
aline 1 10
aline 5 10
aline 5 6

amove 1 6
set lstyle 0
bezier 1 10 5 10 5 6
```




```
amove 11 11
set lstyle 2
aline 14 6
aline 15 11
aline 17 8

amove 11 11
set lstyle 0
bezier 14 6 15 11 17 8

amove 1 1
set lstyle 2
rline 4 4
rline 3 -4
rline 4 2
rline 2 1
rline 1 -1
rline 1 1

amove 1 1
```

```
set lstyle 0
rbezier 4 4 7 0 11 2
rbezier 2 1 3 0 4 1

amove 6 6
set hei 0.6
text Bezier cubic sections
```

There are two commands that can be used to draw bezier sections, the first, `bezier`, is followed by three sets of absolute coordinates,

```
bezier 1 2 2 4 3 6
```

This will draw a third degree polynomial through the starting point and the final coordinate, the other two control points define the form of the rest of the curve. The second command we can use is `rbezier`, this performs the same operation except the points are now defined in terms of relative coordinates from the start point.

```
rbezier 1 2 3 6 6 12
```

This will draw the same curve as the example above, but with relative coordinates. Notice that this is different from the `curve` command, where the relative points were taken from the last point instead of the overall start position.

A discussion of the mathematics behind Bezier sections is beyond our present purposes, an explanation can be found in the Postscript Language Reference Manuel or in texts on computer graphics. With experience it is fairly easy to see roughly where the final curve will lie. Because of the applications to computer graphics there are several sites on the internet that offer interactive drawing routines to play with, for example try `http://www.cstc.org/data/resources/47/bezier.html` .

Mathematically we can define a Bezier section with any number of control points, for each control point the fitted polynomial gains an extra degree - 6 control points will give a 5th degree polynomial. In practice we normally only use cubic sections (or parabolic sections) for most applications. More complex curves are built up from a series of cubic sections joined together. This saves on processing power and also allows small corrections to be made to a single point without affecting the entire curve.

We demonstrate the joining of two bezier sections in the last curve of the `bezier.gle` example. A continuous curve is trivially made by starting a new bezier from the end point of the last one. To match the first derivatives as well we note that the gradient at the beginning of the curve is given by the line joining the first two control points, and that the gradient at the end is given by the line joining the last two control points. The first derivative will then be continuous across the two Bezier sections if all three control points are collinear. It is not generally possible to ensure continuity of the second derivative across a curve boundary.

6.3.3 Circular arcs

We have already looked at the simple example of the circle, we may extend this to draw a small arc of a circle. GLE 3.5 (not 3.3) also supports a similar set of commands for drawing all, or portions, of an ellipse.

To draw a portion of a circle we use the `arc` command, where we must now specify the radius of the arc and the angles that we wish to draw it between. For example,

```
arc 2 10 20
```

which will draw an arc of radius 2cm in an anti-clockwise direction between 10° and 20°. The angular convention is as polar coordinates, with 0° at 3 o'clock and 90° at 12 o'clock. Angles can be given that are greater than 360°, and can be either positive or negative.

To draw an arc in the opposite direction, which can simplify some diagrams and is important when constructing paths, we use the `narc` command.

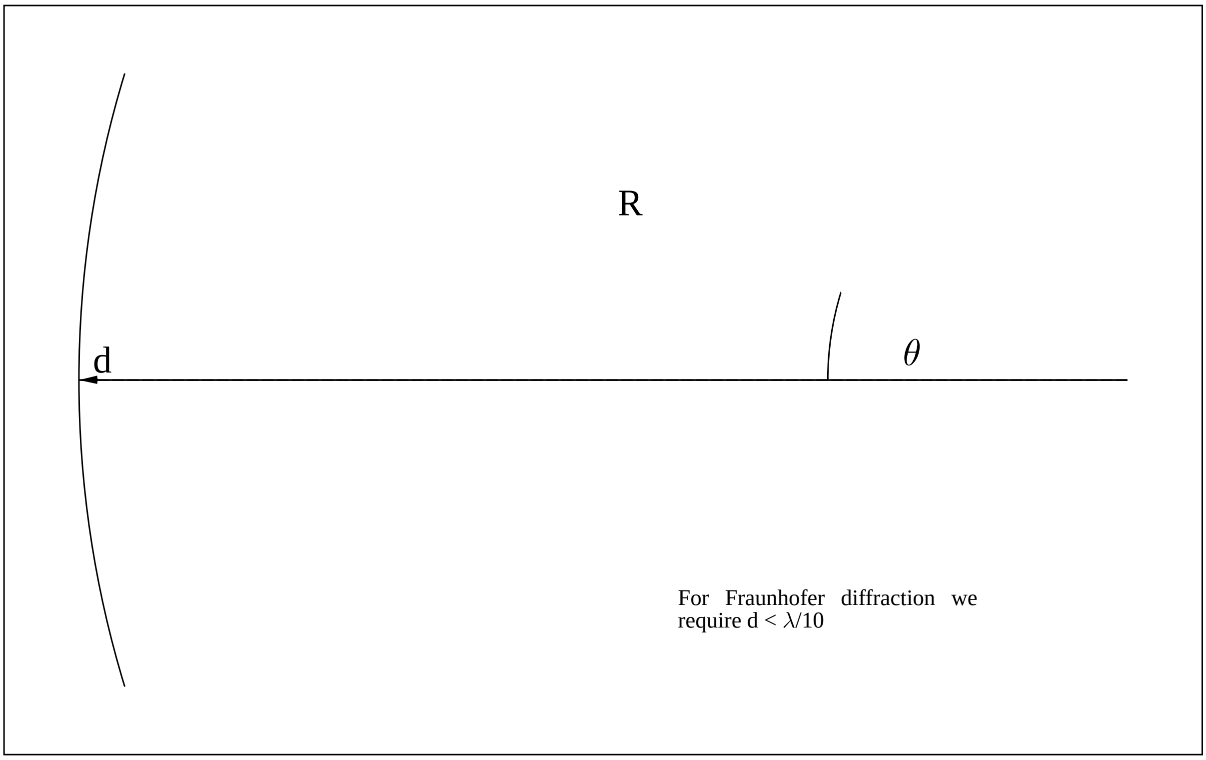
```
narc 1 10 20
amove xend() yend()
```

This will draw a 1cm arc from 20° to 10°, then move to the end of the arc at 10°. Notice that the command,

```
arc 1 20 10
amove xend() yend()
```

would have draw the arc in the anti-clockwise direction all the way around the origin.

We demonstrate the use of the arc command with an example showing the conditions for Frauenhofer diffraction.




```
!Frauenhofer.gle - The condition for frauenhofer diffraction
size 16 10

ang=17
radius=14
```

```
amove 15 5

arc 4 180-ang 180
narc radius 180+ang 180-ang
rline -radius*cos(torad(ang)) radius*sin(torad(ang))

set lstyle 2
rline 0 -radius*sin(torad(ang))
save point1
rline 0 -radius*sin(torad(ang))
set lstyle 0
aline 15 5
set lstyle 3
rline -radius 0
save point2

set hei 0.5
join point1 <-> point2
rmove 0.2 0.1
text d

rmove radius/2 2.1
text R

amove 12 5.2
text \theta

amove 9 2
set hei 0.3
begin text width 4
For Fraunhofer diffraction we require  d < \lambda/10
end text
```

Unfortunately GLE is not consistent in its use of radians or degrees. Generally, mathematical expressions such as `sin` or `tan` require the input angle to be in radians, and the corresponding inverse hyperbolic functions output angles in radians. Fractions of circular and elliptical arcs are, however, specified in degrees. There are two useful functions that will convert between the two units, `todeg` and `torad`. Thus to find the sine of a 10° angle we write,

```
sin(torad(10))
```

And to find an angle in degrees,

```
angle = todeg(acot(3/4)
```

will give the variable `angle` the value in degrees of `arccotan`(34)/;.

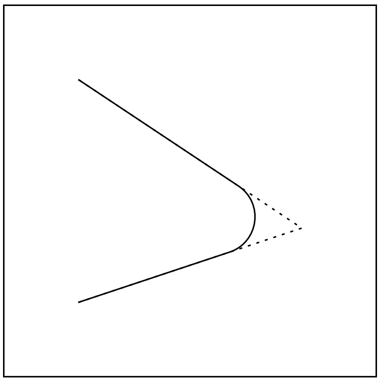
In the example above we choose to work mainly in degrees, the conversion ratios being forced upon us when we use a combination of trigonometric ratios and `amove` commands. Such a combination may seem slightly clumsy. Why, for example could we not have written,

```
narc radius 180+ang 180-ang
rline xend() yend()
```

The answer is, of course, is that it will not work. GLE does not, unfortunately, support the use of the `xend()` and `yend()` command with either circular or elliptical arcs. On this occasion we are forced to resort to trigonometry to produce the effect we require. It is important to be aware of the limitations of some of the commands in GLE, as identifying unexpected errors caused by the failure of a command can be time consuming.

6.3.4 The `arcto` command

An adaptation of the `arc` command performs a commonly used routine: the connecting of two points by an arc of a given radius. We look at a simple example,



```
!arcto.gle - Demonstrates the arcto command
size 5 5

amove 1 1
set lstyle 2
aline 4 2
aline 1 4

amove 1 1
set lstyle 0
arcto 3 1 -3 2 0.5
```

This compares the straight (dotted) lines drawn with the curved arc. The command uses relative movement, with the qualifiers taking the form,

```
arcto x1 y1 x2 y2 rad
```

This draws a line from the current point to a point a vector (x1,y1) away, then another line a further vector (x2,y2) away. The lines are then replaced with a partial arc of radius `rad` instead of the vertex at (x1,y1).

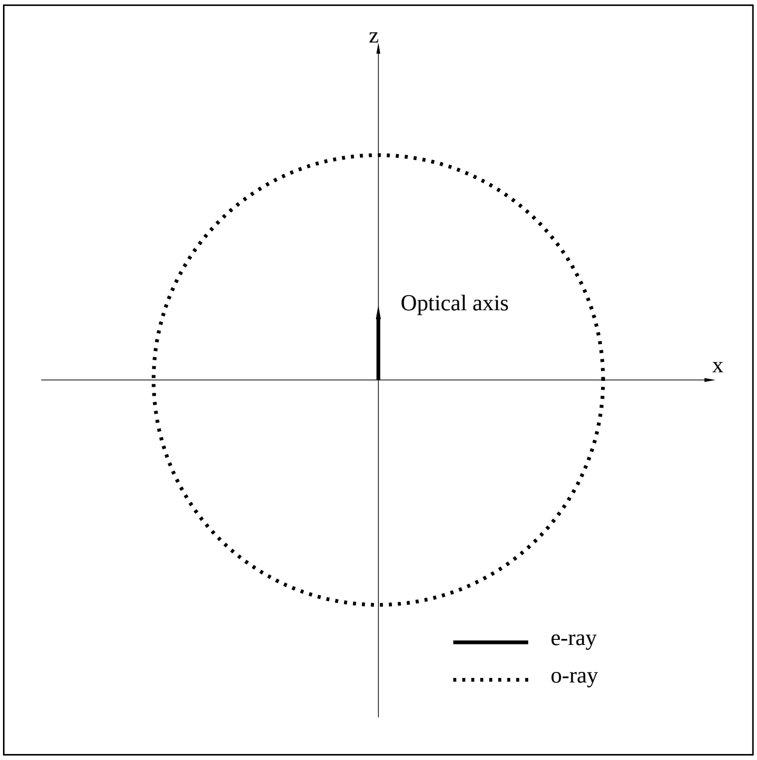
6.3.5 Ellipses

The `ellipse` command is very similar to the circle command, except we must now specify the eccentricity of the ellipse. The syntax is,

```
ellipse 2 1 fill red
```

which will draw a red ellipse with major axis 2 and minor axis 1.

We look at a simple example of the use of ellipses that leads on to a more complex program in the following section.




```
!light.gle - demonstrates the use of ellipses and circles in the propagation of light
size 10 10

set just br lwidth 0.01 hei 0.3
amove 5 0.5                   !Draws a set of axis
aline 5 9.5 arrow end
text z
rmove 0.1 0.1
amove 0.5 5
aline 9.5 5 arrow end
rmove 0.1 0.1
text x
amove 5 5
```

```
set lwidth 0.05                 !Draws the ellipse and circle that represent the
ellipse 4 3                     !propagation of the o-ray and e-ray.
set lstyle 2
circle 3

set just cl                     !Adds the direction of the optical axis
amove 5 5
set lstyle 0
rline 0 1 arrow end
text \glass \: Optical axis

amove 6 1.5                     !Adds a key
rline 1 0
text \glass \: e-ray
amove 6 1
set lstyle 2
rline 1 0
text \glass \: o-ray
```

The diagram shows the propagation of wavefronts in a birefringent material, the o-ray represents the component of the polarisation vector parallel to the y-axis (out of the page), and the e-ray represents the remaining component perpendicular to both the direction of propagation and the y axis. Notice that for light propagating along the optical axis the two polarisations are identical, but if there is a component of the electric field along the optical axis then the propagation speed is (in our example) greater.

6.3.6 Elliptical arcs

To illustrate the commands we look at a diagram to illustrate the different directions of propagation of light within a birefringent crystal. Such crystals have a directional anisotropy in the crystal binding forces, which produces a similar anisotropic effect in the rate of propagation of light through the crystal - light that has its polarisation vector in the direction of the anisotropy (the optical axis) will travel at a different speed (faster in our example) than light polarised orthogonal to the optical axis. Using Huygens construction we can represent the propagation of light through the crystal as a series of ellipses or circles. In practice, we illuminate the crystal with natural light containing both polarisations, the birefringent material will split the incoming light into two independent beams, one polarised along the optical axis and the other against it.

```
!oray.gle - Example of the use of ellipses and arcs

size 22 9.5

inclination=5           !Sets the inclination of the optical axis to the
                        !crystal clevage plane

sub arcdraw n space ang
gsave                   !Subroutine to draw an arc of a circle
set lwidth 0.001
```

```
rmove 0 (n-1)*space/2
for i = 1 to n step 1
marker dot 0.2
arc 1 -ang ang
rmove 0 -space
next i
grestore
end sub

sub ellipsedraw n space ang inc
gsave
rmove 0 (n-1)*space/2   !Subroutine to draw an arc of an ellipse
set lwidth 0.001
begin rotate inc
for j = 1 to n step 1
marker dot 0.2
elliptical_arc 1 0.6 -ang-10 ang+10
rmove 0 -space
next j
end rotate
grestore
end sub

sub drawcrystal         !Subroutine to draw the crystal rhombus
set lwidth 0.05
rline 0 7
rline 5 2
rline 0 -7
rline -5 -2
rline 0 7
end sub
```

```
sub move inc            !Subroutine to move along the crystal in a direction
a=cos(torad(inc))       !perpendicular to the optical axis
b=sin(torad(inc))
set lwidth 0.01
rline a b
set lwidth 0
end sub

sub drawaxis size                                !Subroutine to draw a set of
gsave                                            !xyz axis, this is a commonly
set lstyle 1 lwidth 0.01 hei 0.3 just bl         !used routine and could
rline size 0 arrow end                           !probably be put in file
text x                                           !on its own
rmove -size size
text z
rline 0 -size arrow start
rline size*((sqrt(3))/2) size/2 arrow end
text y
grestore
end sub

amove 3 0.25            !Draws the first o-ray crystal
@drawcrystal

amove 0 4.5             !Draws the incoming light for the first crystal
set lwidth 0.05
rline 2 0 arrow end
rline 1 0
set lwidth 0.01

@arcdraw 3 1 90         !Draws the spherical wavelets for the o-ray
rline 1 0
@arcdraw 6 0.5 45
rline 1 0
@arcdraw 12 0.25 30
rline 1 0
@arcdraw 24 0.125 15
rline 2 0

set lwidth 0.05
rline 2 0 arrow end     !Draws the outgoing wave for the o-ray crystal

amove 15 0.25
@drawcrystal            !Draws the second (e-ray) crystal

amove 12 4.5            !Draws the second incoming ray
```

```
set lwidth 0.05
rline 2 0 arrow end
rline 1 0

set lstyle 2            !Draws the expected direction of propagation for an
rline 5 0               !isotropic crystal
rmove -5 0
set lstyle 1 lwidth 0.01

@ellipsedraw 1 1 70 inclination         !Draws the actual propagation
@move inclination                       !using elliptical wavelets
@ellipsedraw 4 1 45 inclination
@move inclination
@ellipsedraw 8 0.5 40 inclination
@move inclination
@ellipsedraw 16 0.25 15 inclination
@move inclination
@move inclination

set lwidth 0.05         !Draws the outgoing ray
rline 2 0 arrow end

amove 15.5 1.25         !Adds the direction of optical axis to both crystals
rline -sin(torad(inclination)) cos(torad(inclination)) arrow end
amove 3.5 1.25
rline -sin(torad(inclination)) cos(torad(inclination)) arrow end

amove 15.6 1.25         !Adds the label to the optical axis
set just left hei 0.3
text optical axis
amove 3.6 1.25
text optical axis


set just tr

amove 2.8 4.2           !Adds labeling to the incoming and outgoing rays
begin text width 2.7
Incoming light polarised in the y direction
end text

amove 14.8 4.2
begin text width 2.7
Incoming light polarised in the z direction
end text

amove 9.9 4.2
```

```
begin text width 1.5
o-ray (polarised)
end text

amove 21.9 4.2
begin text width 1.5
e-ray (polarised)
end text

amove 0 4.6             !Draws two axis for both the rays
@drawaxis 1
amove 12 4.6
@drawaxis 1

amove 11 0              !Separates the two crystals with a solid line
set lstyle 1 lwidth 0.1
aline 11 10
```

6.4 Path drawing

Within GLE we can draw a set of lines that makes up an arbitrary shape, then use the `begin path` and `end path` to define the shape and apply fill shades and line styles. The lines of code within the two commands will be stored by GLE and used to draw the entire shape at once. Commands that refer to different coordinate points will be linked by a line, for example if we were to include the code,

```
rline 1 0
rmove 0 1
rline 1 0
```

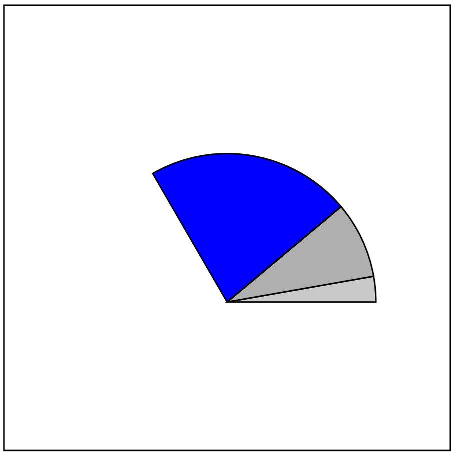
The `rmove` along the y-axis would be considered part of the boundary of the shape, though whether a line is actually drawn connecting the two points is left as an option for the user.

Some of the commands used in conjunction with the path drawing routine are device sensitive, they can only be guaranteed on postscript devices. Any other program may successfully interpret the postscript commands, but it is usually best to avoid drawing diagrams that are heavily dependent upon these commands.

6.4.1 A simple path

There is no support within GLE for drawing pie-charts; the following subroutine file, and an example of its use, can draw a pie wedge filled with any given colour.

```
!piesub.gle - Subroutine to draw a pie wedge between two supplied angles
sub pie ang1 ang2 radius colour$
begin path fill colour$ stroke
rmove 0 0                   !The rmove command is neccesary to set the beginning
arc radius ang1 ang2        !of the path
closepath
```



```
end path
end sub
```

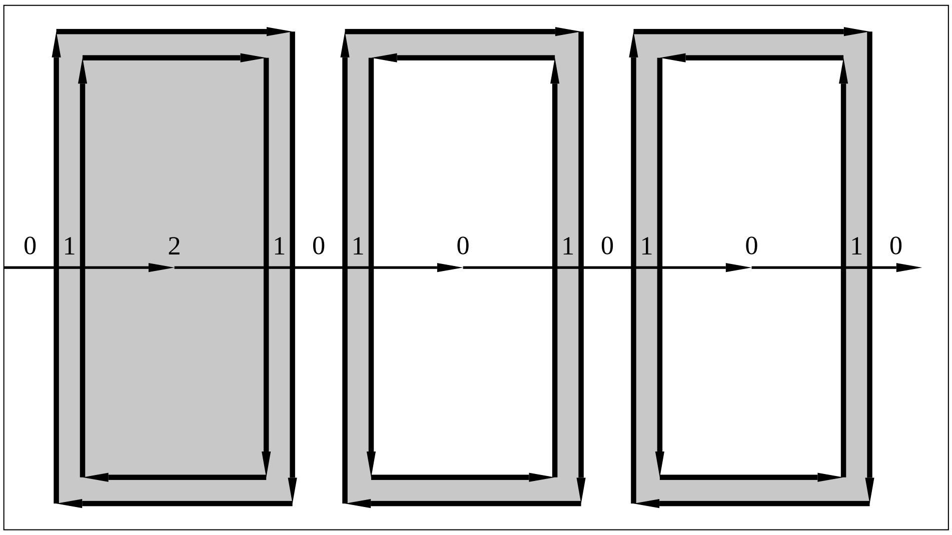
```
!drawpie.gle - Example of the use of the pie wedge routine
size 6 6
include piesub.gle

amove 3 2
@pie 0 10 2 "grey10"
@pie 10 40 2 "grey20"
@pie 40 120 2 "blue"
```

A path within GLE must contain the elements `begin path` and `end path`. To the `begin path` command we can affix qualifiers that determine the qualities of the finished shape. In the example above the `stroke` command tells GLE to draw the lines that make up the shape in the currently selected line style and width. The `fill` command fills the 'inside' of the shape with the selected fill pattern, which may take the form of a text string as in the example above. What we mean by 'inside' the shape may seem elementary in the example above, and indeed it is, but a more formal definition, for use with complex paths, will be given in the following section. A further option, the `clip` command, will also be covered in the section on clipping.

The path drawn will be defined from the start of the first object drawn following the `begin path` command. This is not, neccessarly, the same as the initial coordinate point. In the example above the `rmove 0 0` commands defines the start of the path to be at the current coordinate point, without it the start of the path would be at the beginning of the arc. The `closepath` commands returns to the beginnning of the shape; useful, and often necessary, when drawing complex shapes, though in the example above there is nothing to stop us completing the path manually using `rmove` command.

6.4.2 The zero winding rule

When filling complex shapes using the path commands we may often run into problems deciding which part of the shape is to be filled and which is to be left blank. GLE uses the 'Zero winding rule' to decide, and also provides a command that allows us to define for ourselves which the boundaries of the path.




```
!Zero.gle - Program to demonstrate the zero winding rule
size 18 10

sub drawbox a b         !These subroutines draw the various boxes to size.
        rline 0 b       !Because the path command does not recognise the
        rline a 0       !arrowed lines we must draw the boxes twice, once
        rline 0 -b      !for the path and once for the arrows.
        rline -a 0
        end sub

sub antidrawbox a b
        rline a 0
        rline 0 b
        rline -a 0
        rline 0 -b
        end sub

sub drawarrowbox a b
        rline 0 b arrow end
        rline a 0 arrow end
        rline 0 -b arrow end
        rline -a 0 arrow end
        end sub

sub antidrawarrowbox a b
        rline a 0 arrow end
        rline 0 b arrow end
        rline -a 0 arrow end
        rline 0 -b arrow end
        end sub
```

```
amove 1 0.5
set lwidth 0.1

begin path fill grey10          !This draws two boxes in the same direction
        @drawbox 4.5 9
        rmove 0.5 0.5
        @drawbox 3.5 8
        end path
@drawarrowbox 4.5 9
rmove 0.5 0.5
@drawarrowbox 3.5 8

amove 6.5  0.5                  !This draws two boxes in opposite directions
begin path fill grey10
        @drawbox 4.5 9
        rmove 0.5 0.5
        @antidrawbox 3.5 8
        end path
@drawarrowbox 4.5 9
rmove 0.5 0.5
@antidrawarrowbox 3.5 8

amove 12 0.5

begin path fill grey10          !This draws two boxes in the same direction
        @drawbox 4.5 9          !but uses the reverse command to reverse the
        rmove 0.5 0.5           !path direction
        reverse
        @drawbox 3.5 8
        end path
@drawarrowbox 4.5 9
rmove 0.5 0.5
@antidrawarrowbox 3.5 8

amove 0 5                       !The following adds some labels
set just cb hei 1.0 lwidth 0.05
rline 3.25 0 arrow end
rline 5.5 0 arrow end
rline 5.5 0 arrow end
rline 3.25 0 arrow end
set hei 0.5
amove 0.5 5.25
text 0
rmove 0.75 0
text 1
rmove 2 0
text 2
```

```
rmove 2 0
text 1
rmove 0.75 0
text 0
rmove 0.75 0
text 1
rmove 2 0
text 0
rmove 2 0
text 1
rmove 0.75 0
text 0
rmove 0.75 0
text 1
rmove 2 0
text 0
rmove 2 0
text 1
rmove 0.75 0
text 0
```

The zero winding rule is applied as follows: An imaginary line is drawn across the object. Every time a line of the path crosses the from left to right, one is added to a counter (initially set to zero); every time a line of the object crosses from right to left, one is subtracted from the same counter. Everywhere the counter is non-zero GLE assumes to be inside the object, everywhere else is considered to be outside.

When creating a complex filled shape from a path we must be aware of the direction we are drawing in. To change direction we can either deliberately draw in the opposite direction to the rest of the shape, or else use the `reverse` command which reverses the direction of the path of all the commands following it. The latter is useful if we wish to reuse a subroutine, as we show above.

GLE treats the commands that constitute a path as an entirely separate entity from the rest of the program. Notice that if we had not drawn the extra boxes on top of the path then there would have been no outline to the box. If we do not add a `stroke` qualifier to the `path` command then any lines that are drawn in the path will be ignored.

A problem also occurs in the use of arrow in path commands. If we specify any line or curve to begin or end with an arrow then the fill style and stroke commands will not apply, that is the path commands will fail.

6.4.3 Clipping

Clipping is a technique that will display only a portion of a diagram. The displayed portion being outlined using the path commands that we used in the previous section. Clipping is device sensitive and only works with postscript devices.

```
!Clip.gle - Demonstrates the useage of the path clip command
size 10 10              !Formats the text
set hei 4
```


```
amove 2 2
begin clip
        begin path clip
                text GLE
                amove 2 6
                box 5 2
                end path
        amove 2 2
        for x = 0.01 to 100 step 0.005
                circle 1/x
                next x
        end clip
amove 5 5
rline 10 10
```

We can draw an arbitrary clipping region by defining a closed path within the commands `begin path clip` and `end path`. We can use any arbitrary shape, but only Postscript fonts will currently work for this purpose.

The example defines a box and a line of text as the clipping region. We also draw a series of concentric circles centered on (2,2), only the areas that are defined as being within the clipped path will show through. The `begin clip` and `end clip` commands signify the end of the clipping region.

It is useful to define a clipping region when we wish to contain a set of commands to a given area. If we look at the diagram of the propagation of light in the birefringent material then, if we were producing a diagram for inclusion in a paper or thesis, it would have helped to define the crystal boundaries as a clipping region. This would have prevented the intrusion of the wavelets into the area beyond the limit of the crystal. Of course this would of added an extra layer of complexity to the program, one which we did not need at the time. It is important to balance the neatness of the clipping command with the added time taken and the lack of compatibility with non-postscript devices.

6.5 File handling

It is possible to read and write to files using GLE using various file handling commands that look rather like those used in C. `fopen` is used to open a file, `fclose` is used to close a file. This is perhaps best illustrated by an example. Imagine that you have a datafile which contains 5 columns of numbers. You want to make a new datafile with GLE that contains three columns of data (let's call them x, y and yerr) according to a particular algorithm (in this case, we ignore the first column of numbers, set x to be the second column, set y to be the quotient of the fifth and third column, and work out yerr as the error assuming that column 4 is the error for column 3, and column 6 is the error for column 5). Both the file to be read and the file to be written must be assigned a channel. In the example below, we use `inchan` as the channel for the input file (called "measured.dat") and `outchan` as the channel for the output file (called "calculated.dat").

```
fopen "measured.dat" inchan read
fopen "calculated.dat" outchan write

until feof(inchan)
    fread inchan a1 a2 a3 a4 a5 a6
    x=a2
    y=a5/a3
    yerr=y*sqrt((a6/a5)^2+(a4/a3)^2)
    fwriteln outchan x y yerr
next

fclose outchan
fclose inchan
```

This code has the desired effect and illustrates the use of the command `fopen`, `fclose`, `fread`, `fwrite`, `fwriteln`, `feof`, and the `until next` loop.

7 Appendices

7.1 List of available functions

7.1.1 Text functions

```
time$                   Returns current time.
date$                   Returns current date.
left$(string$,4)        Returns the first three characters of expression ‘str$’.
right$(str$,4)           Returns the characters of ‘str$’ from the fourth letter.
seg$(string$,2,5)       Returns the characters from the second to the fifth.
num$(4)                 Text representation of ‘4’.
```

num1$(4)	As above but with no spaces.
val(string$)	Value of string ‘string$’.
pos(string1$,sting2$,exp)	Position of ‘string2$’ in ‘string1$’ from position ‘exp’.
len(string$)	Returns the length of ‘string$’.

7.1.2 Mathematical functions

abs(exp)	Absolute value of ‘exp’.
acosh(exp)	Inverse hyperbolic cosine of ‘exp’.
acot(exp)	Inverse cotangent of ‘exp’.
acoth(exp)	Inverse hyperbolic cotangent of ‘exp’.
acsc(exp)	Inverse cosecant of ‘exp’.
acsch(exp)	Inverse hyperbolic cosecant of ‘exp’.
asec(exp)	Inverse secant of ‘exp’.
asech(exp)	Inverse hyperbolic secant of ‘exp’.
asinh(exp)	Inverse hyperbolic sine of ‘exp’.
atn(exp)	Arctangent of ‘exp’.
atanh(exp)	Inverse hyperbolic tangent of ‘exp’.
cos(exp)	Cosine of ‘exp’.
cosh(exp)	Hyperbolic cosine of ‘exp’.
cot(exp)	Cotangent of ‘exp’.
coth(exp)	Hyperbolic cotangent of ‘exp’.
csc(exp)	Cosecant of ‘exp’.
csch(exp)	Hyperbolic cosecant of ‘exp’.
exp(exp)	Exponent of ‘exp’.
fix(exp)	‘exp’ rounded towards zero.
int(exp)	Interger part of ‘exp’.
log(exp)	Log to the base e of ‘exp’.
log10(exp)	Log to the base 10 of ‘exp’.
sec(exp)	Secant of ‘exp’.
sech(exp)	Hyperbolic secant of ‘exp’.
sgn(exp)	Returns +1 if ‘exp’ is positive and -1 if ‘exp’ is negative.
sin(exp)	Sine of ‘exp’.
sinh(exp)	Hyperbolic sine of ‘exp’.
sqr(exp)	‘exp’ squared.
tan(exp)	Tangent of ‘exp’.
tanh(exp)	Hyperbolic tangent of ‘exp’.
todeg(exp)	Converts radians to degrees.
torad(exp)	Converts degrees to radians.
not(exp)	Logical not of ‘exp’.
rnd(exp)	Random number generated from seed ‘exp’.
sqrt(exp)	Square root of ‘exp’.

7.1.3 Graphic functions

xend()	The x coordinate of the end of a drawn text string.

yend()	The y coordinate of the end of a drawn text string.
xpos()	The current x coordinate.
ypos()	The current y coordinate.
twidth(string$)	The width of ‘string$’ in the current font.
theight(string$)	The height of ‘string$’ in the current font.
tdepth(string$)	The depth of ‘string$’ in the current font.
xg(xexp)	The x coordinate ‘xexp’ on the last graph.
yg(xexp)	The y coordinate ‘xexp’ on the last graph.

7.2 Special LaTeX characters

^{}	Superscript
_{}	Subscript
\\\\	Forced Newline
_	Underscore character
\\,	.5em (em = width of the letter ‘m’)
\\:	1em space
\\;	2em space
\char{22}	Any character in current font
\chardef{a}{hello}	Define a character as a macro
\def\v{hello}	Defines a macro
\movexy{2}{3}	Moves the current text point
\glass	Makes move/space work on beginning of line
\rule{2}{4}	Draws a filled in box, 2cm by 4cm
\setfont{rmb}	Sets the current text font
\sethei{.3}	Sets the font height (in cm)
\setstretch{2}	Scales the quantity of glue between words
\lineskip{.1}	Sets the default distance between lines of text
\linegap{-1}	Sets the minimum required gap between lines

The remaining characters are shown in the diagrams below

7.3 Font list

rm	Roman
rmb	Roman Bold
rmi	Roman Italic
ss	San Serif
ssb	San Serif Bold
ssi	San Serif Italic
tt	Typewriter
ttb	Typewriter Bold
tti	Typewrite Italic
texcmb	Computer Modern Bold

texcmex	Computer Modern Extensible
texcmitt	Computer Modern Italic Typewriter
texcmmi	Computer Modern Maths Italic
texcmr	Computer Modern Roman
texcmss	Computer Modern Sans Serif
texcmssb	Computer Modern Sans Serif Bold
texcmssi	Computer Modern Sans Serif Italic
texcmsy	Computer Modern Symbol
texcmti	Computer Modern Text Italic
texcmtt	Computer Modern Typewriter Text
plba	Block Ascii
plcc	Complex Cartographic
plcg	Complex Gothic
plci	Complex Italic
plcr	Complex Roman
plcs	Complex Script
pldr	Duplex Roman
plge	Gothic English
plgg	Gothic German
plgi	Gothic Italian
plsa	Simplex Ascii
plsg	Simplex German
plsr	Simplex Roman
plss	Simplex Script
plsym1	Symbols one
plsym2	Symbols two
plti	Triplex Italic
pltr	Triplex Roman
psagb	AvantGarde-Book
psagbo	AvantGarde-BookOblique
psagd	AvantGarde-Demi
psagdo	AvantGarde-DemiOblique
psbd	Bookman-Demi
psbdi	Bookman-DemiItalic
psbl	Bookman-Light
psbli	Bookman-LightItalic
psc	Courier
pscb	Courier-Bold
pscbo	Courer-BoldOblique
psco	Courier-Oblique
psh	Helvetica
psgb	Helvetica-Bold
pshbo	Helvetica-BoldOblique
psho	Helvetica-Oblique
pshc	Helvetica-Condensed

pshcb	Helvetica-Condensed-Bold
pshcbo	Helvetica-Condensed-BoldOblique
pshn	Helvetica-Narrow
pshnb	Helvetica-Narrow-Bold
pshnbo	Helvetica-Narrow-BoldOblique
pshno	Helvetica-NarrowOblique
psncsb	NewCenturySchlbk-Bold
psncsbi	NewCenturySchlbk-BoldItalic
psncsi	NewCenturySchlbk-Italic
psncsr	NewCenturySchlbk-Roman
pspb	Palatino-Bold
pspbi	Palatino-BoldItalic
pspi	Palatino-Italic
pspr	Palatino-Roman
pstr	Times-Roman
psti	Times-Italic
pstb	Times-Bold
pstbi	Times-BoldItalic
pszcmi	ZapfChancery-MediumItalic
pszd	ZapfDingbats
pssym	Symbol

7.4 Command index

7.4.1 Basic commands

`! comment`

Indicates the start of a comment within a program script. GLE will ignore everything from the exclamation mark to the end of line. It is best not to place comments on the same line as a `text` or similar command, as the exclamation mark is treated as part of the text.

`=@ xxx`

Executes the subroutine `xxx`. Subroutines should be defined at the start of a programming script.

`aline x y [arrow start] [arrow end] [arrow both]`

Draws a line from the current point to the absolute coordinates (x,y). The optional qualifiers will add an arrow head to the line, the size of this head is defined by the current font size, *not* by the line-width.

`amove x y`

Moves to the absolute coordinates (x,y).

`arc radius a1 a2`

Draws an arc of a circle of given radius centred on the current coordinate point. The arc is drawn in an anti-clockwise direction from and angle `a1` to and angle `a2`, with both angles in degrees.

`arcto x1 y1 x2 y2 rad`

Draws a line from the current point to (x1,y1) then to (x2,y2), but with an arc of radius `rad` replacing the vertex at (x1,y1).

`begin box [fill pattern] [add gap] [nobox] [name xyz]`

Draws a box around everything between the commands `begin box` and `end box`. The `add` option places an extra margin, of `gap` cm, around the graphics within the box commands. The `nobox` option removes the outline of the box. The `name` command is used when we wish to refer again to the box in a subsequent `join` command.

`begin clip`

This saves the region between the commands `begin clip` and `end clip` for use with the `begin path clip` command.

`begin origin`

Sets the current point to behave like the origin, (0,0), all commands placed between the two commands `begin origin` and `end origin` will be referred to this point. The command is useful when calling subroutines or when drawing something with `aline` and `amove` commands.

`begin path [stroke] [fill pattern] [clip]`

Initializes the drawing of a filled shape. All the lines and curves generated from `begin path` to the next `end path` command will be stored and then used to draw the shape. `Stroke` draws the outline of the shape, `fill` paints the inside of the shape in the given colour, and `clip` defines the region as a clipping region for use with the `begin clip` command.

`begin rotate angle`

The coordinate system is rotated anti-clockwise about the current point an angle given in degrees.

`begin scale x y`

Scales the commands between `begin scale` and `end scale` by fractions x in the x direction and y in the y direction.

`begin table`

This module is an alternative to the `begin text` module. It reads the spaces and tabs in the source file and aligns the words accordingly.

`begin text [width exp]`

This module allows multiple lines of text to be displayed. The text within the module will be displayed with the line breaks and spaces as it appears in the script, if a `width` option is also present then the text will wrap around at this width. We can set the justification, font, and size with a previous `set` command. Within the block of text we are free to use many of the special characters produced using LaTeX commands.

`begin translate x y`

All the graphics within the translate block are moved by x units to the right and y units up.

`bezier x1 y1 x2 y2 x3 y3`

Draws a cubic Bezier section (3rd degree polynomial) from the current point to the point (x3,y3), with control points at (x1,y1) and (x2,y2).

`bigfile filename.gle`

Reads the file 'filename.gle' and executes it line by line, this saves on memory and allows a file of any size to be run, but some complex multi-line commands are not available.

`box x y [justify jtype] [fill color] [name xxx] [nobox]`

Draws a box of width x and height y. The box is positioned with the `justify` command according to the standard GLE handles, TL for top-left, CB for centre-bottom, etc. The `name` option is for later use with the `join` command, and the `nobox` option removes the outline of the box.

`circle radius [fill pattern]`

Draws a circle of given radius centred on the current coordinate point.

`closepath`

Joins the current point to the beginning of the path.

`curve ix iy [ x1 y1 x2 y2 x3 y3 ... xn yn] ex ey`

Draws a smooth curve through the points [x1 y1 x2 y2 x3 y3 ... xn yn] with intial gradient (ix,iy) and final gradient (ex,ey).

`define marker markername subroutine-name`

This defines a new marker called 'markername' which will call the subroutine 'subroutine-name' whenever it is used. It passes two parameters, the first is the size of the marker and the second is a value from a secondary dataset which can vary the size or rotation of each marker plotted.

`for var = exp1 to exp2 [step exp3] command [...] next var`

The `for...next` structure repeats the commands contained within the block to be repeated while the `for` expression is still valid.

`grestore`

Restores the most recently saved graphic state. It must be paired with a `gsave` command.

`gsave`

Saves the current graphic state including the current graphics transformation matrix, the current point and the current font and colour settings.

`if exp then command [...] else command [...] end if`

If `exp` is evaluated to be true then the commands following `then` are executed, if it is false then the commands following `else` are executed. Either way normal program execution resumes after the `end if` command. Note that `end if` is not spelt `endif`.

`include filename`

The commands in the file 'filename' are read just as they would if they were written out in the program script. This can help simplify complex programs and also allow common routines to be easily recycled.

`join object1.just sep object2.just`

Joins two objects that have been named with either `name` commands or the `save` command. `.just` is the justification that controls which point on the object is used. `sep` can be - for a single line, -> for a single arrow, or <-> for a double arrow.

`marker marker-name [scale-factor]`

Draws the marker 'marker-name' at the current point. The marker size is proportional to the current font size but scaled by the factor `[scale factor]`. There are a number of preset markers in GLE but additional ones can be defined with the `define marker` command.

`postscript filename.eps width-exp height-exp`

Includes an encapsulated postscript file into a GLE graphic. The postscript picture will be scaled according to the width supplied such that the aspect ratio is maintained, the heigh supplied is used to draw a rectangle on the screen.

`rbezier x1 y1 x2 y2 x3 y3`

Identical to the `bezier` command except the points given are relative to the current coordinate point.

`return exp">`

`reverse`

Reverses the direction of the current path, this is useful when we are drawing complex objects and need to indicate which part of the object is to be considered the 'inside'.

`rline x y [arrow end] [arrow start] [arrow both]`

Draws a line form the current point to the relative coordinate (x,y), which then becomes the current point. The optional qualifiers at the end will draw arrow heads at one or both of the line ends; the size of the arrow head is proportional to the current font size.

`rmove x y`

Moves to the relative coordinate (x,y).

`save objectname`

Saves the current point for use with a subsequent `join` command.

`set cap butt | round | square`

Determines the appearance of the end of a wide line.

`set color col`

Sets the colour for use with all future drawing commands.

`set dashlen dashlen-exp`

Sets the length of the smallest dash length that appears in the various line styles.

`set font font-name`

Sets the current font, a list of valid font names is given in the appendix.

`set fontlwidth line-width`

Sets the width of the lines in stroked (Plotter) fonts.

`set hei character-size`

Sets the height of the text in cm, though the actual size will be only 65% of this value.

`set join mitre | round | bevel`

Controls what happens at the join between two thick lines. For the command to work the lines must form part of a continuous path.

`set just left | center | right | tl | etc...`

Controls the positioning of objects and text in relation to the current coordinate point.

`set lstyle line-style`

Sets the line style for all subsequent lines.

`set lwidth line-width`

Sets the line width in cm for all subsequent lines.

`sub sub-name paramter1 paramter2 etc`

Defines a subroutine 'subname', all commands between the `sub` command and the `end sub` command will form part of the subroutine and can be executed at any point using the `@sub-name` command. The parameters are local variables for use within the subroutine and do not affect the variables defined in the program body.

`text unquoted-text-string`

Prints text to the screen.

`write string$`

Prints the text variable 'string$' to the screen.

7.4.2 Functions for use within expresions

`xg(948), yg(0.04)`

Outputs the x and y coordinates of the point at x=948 and y=0.04 of the last graph drawn.

`xend(), yend()`

Moves to the end of the last thing drawn.

`xpos(), ypos()`

The x and y coordinates of the current point.

7.4.3 The graph module

`bar d4 dist spacing`

Draws a bar chart of the dataset d4, the bars are separated by a distance specified by the `dist` option.

`bar dx,... from dy,...`

Creates a stacked bar chart of the datasets dx,... on top of dy,....

`bar dn,... width xunits,... fill col,... color col,...`

Creates a grouped chart of the datasets dn,.... Each dataset has format options separated by commas and in the same order as the dataset list.

`data filename [d1 d2 d3 ...] [d1=c1,c3]`

Reads the data from the file 'filename' into the datasets [d1 d2 d3 ...], by default the datasets will contain the first and second columns , the first and third, the first and forth, etc. We can specify which columns are read into which dataset using the d1=c1,c3 option, which will, for example, read the first and third column into the dataset d1.

In general we can refer to a dataset either by its number, `d2` for example, or we can refer to all datasets using the label `dn`. In the commands that follow we shall use `dn` to control the formatting of all of the datasets. In each case we can replace `dn` with a the label for a single dataset.

`dn bigfile "all.dat,xc,yc" [marker mname] [line]`

This plots the data from the file 'all.dat' point by point. The operation uses less memory than the `data` command but some of the more complex commands are unavailable.

`dn err d5 errwidth width-exp errup nn% errdown d4`

Adds either absoulte or relative vertical error bars to the dataset dn. `errwidth` controls the width of the terminating line.

`dn herr d5 herrwidth width-exp herrleft nn% errright d4`

Similar to the `err` command but the error bars are horizontal.

`dn key "Dataset title"`

Adds a dataset to the key, see the `key` command for the various format options that control the appearence of the key.

`dn line`

Draws a series of straight lines connecting the data points that comprise the dataset.

`dn lstyle line-style lwidth line-width color col`

Sets the line style for any connecting lines of the dataset.

`dn marker marker-name [msize marker-size] [mdata dn]`

Adds markers at each point in the dataset. The `mdata` command allows us to pass parameters to a marker defined with a subroutine.

`dn nomiss`

Connects all points in the dataset dn, even if there are one or more missing values.

`dn smooth | smoothm`

Draws a smooth (3rd degree polynominal) through the points in the dataset, `smoothm` is used for a dataset with multiple y values for each x point.

`dn xmin x-low xmax x-high ymin y-low ymax y-high`

Controls the maximum and minimum plotted values of the dataset.

`fill [x1,d3 | d4,x2 | d3,d4 | d4] color col [xmin | ymin] val [xmax | ymax] val`

Fills with colour `col` the area between two datasets, or between a dataset and one of the axes.

`fullsize`

Command shortcut that sets the vertical and horizontal scales to 1 and removes the bounding box.

`hscale exp`

Scales the horizontal width of the graph compared with the width defined in the `size` command. The default is 0.7 (70%).

`key pos tl nobox hei exp offset xexp yexp`

Controls the various formatting options available for the key.

`let ds = exp [from low to high step exp ]`

Creates a dataset, ds, from an algebraic expression that may, or may not, be a function of other datasets.

`nobox`

Removes the box that, by default, surrounds any graph drawn with the graph module.

`size x y`

Sets the size of the graph within the main GLE page.

`title "title" [hei ch-hei] [color col] [font font] [dist cm]`

Adds a title to the graph, `dist` sets the vertical positioning of the title.

`vscale exp`

Sets the height of the graph compared with the height defined in the `size` command. The default is 0.7 (70%).

`x2labels on`

Prints the labels on the upper x-axis.

Each of the four axes can be formatted individually, in the following we shall look only at the x-axis commands, though a similar set of commands exists for the second x-axis and the two y-axes.

`xaxis color col font font-name hei exp-cm lwidth exp-cm`

Controls the formatting of the x-axis.

`xaxis dsubticks sub-distance`

Sets the spacing of the subticks.

`xaxis grid`

Makes the ticks and subticks cover the entire graph to create a grid.

`xaxis log`

Uses a logarithmic scale for the x-axis.

`xaxis min low max high dpoints n`

Sets the maximum and minimum displayed values on the x-axis. `dpoints` sets the number of decimal places displayed in the labelling.

`xaxis nofirst nolast`

Removes the first or last label from a graph.

`xaxis nticks number dticks distance`

Sets either the number of ticks or spacing between ticks on the x-axis.

`xaxis off`

Turns off the x-axis.

`xaxis shift cm-exp`

Moves the labeling on the x-axis to the left or right.

`xlabels font font-name hei char-hei color col`

Formats the labels on the x-axis.

`"xnames" "name" "name" ...`

Adds the given labels to the x-axis. Use the `xplaces` command to set the spacing of the labels.

`xplaces pos1 pos2 pos3 ...`

Replaces the default labels with labels at `pos1`, `pos2`, `pos3` and so on.

`xside color col lwidth line-width off`

Formats the appearance of the axis line itself.

`ref id="xsubticks" name="xsubticks lstyle num lwidth exp length exp off`

Formats the appearance of the x-axis subticks. `xsubticks off` will turn them off altogether.

`xticks lstyle num lwidth exp length exp off`

Formats the appearance of the x-axis ticks.

`xtitle "title" [hei ch-hei] [color col] [font font] [dist cm]`

adds a title to the x-axis.

`y2title "text-string" [rotate]`

Rotates the right-hand y-axis by 180°.

7.4.4 The key module

`offset x-exp y-exp`

Sets the position of the key as measured from the bottom left hand corner of the graph.

`position justify-exp`

Sets the position of the key according to the standard GLE justify handles.

`text str-exp`

This is the text that will be displayed in the key.

`lstyle style-num`

Controls the line style drawn on one line of the key.

`marker marker-name`

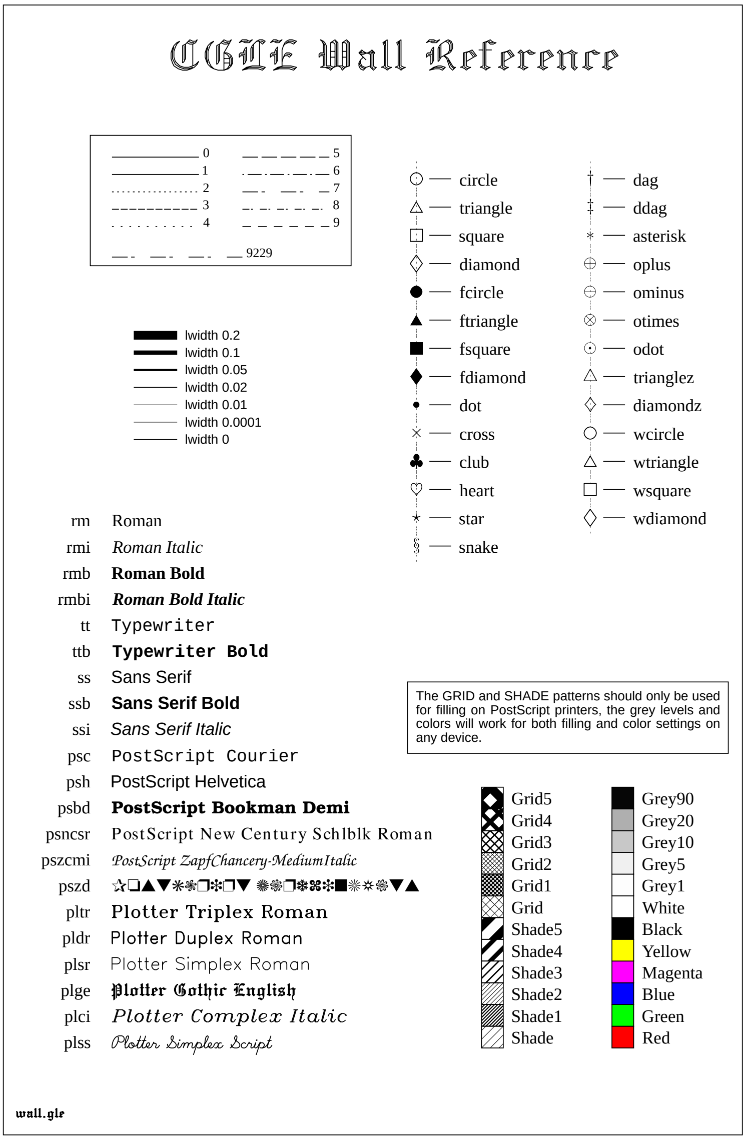
Sets the marker to appear on one line of the key.

`msize exp`

Sets the marker size.

`mscale exp`

Sets the factor that the marker is scaled by.



`color color-name`

Controls the colour of the title, line and marker.

`hei cm-exp`

Sets the text height used in the key.

`fill fill-pattern`

Controls the fill style used on one line of the key.

7.5 GLE wall chart

The GLE wall chart, with marker and line styles and some of the common fonts. For reference here is the program code,

```
!wall.gle - Wall chart for gle
!Original code by Chris Pugmire (I think)


size 17 26 box  ! this is for sticking on your wall
                ! it has fonts, lstyles, shading, colors, markers on it.

set hei 1 just center font plge
amove 9 24.5 !25.5
text CGLE Wall Reference

set just left
set hei .3
amove .3 .4
text wall.gle
amove 2 4
sub sh f$
 set just left
 box -.5 .5 fill f$
 RMOVE .2 .1
 write f$
 rmove -.2 .4
end sub

amove 9.5 10
set hei .3 font ss
begin box add .2
begin text width 7
The GRID and SHADE patterns should only be used for filling
on PostScript printers, the grey levels and colors will work
for both filling and color settings on any device.
end text
end box

set font rm
set hei .4
amove 11.5 2
@sh "Shade"
@sh "Shade1"
@sh "Shade2"
@sh "Shade3"
@sh "Shade4"
@sh "Shade5"
@sh "Grid"
@sh "Grid1"
@sh "Grid2"
@sh "Grid3"
```

```
@sh "Grid4"
@sh "Grid5"

amove 14.5 2
@sh "Red"
@sh "Green"
@sh "Blue"
@sh "Magenta"
@sh "Yellow"
@sh "Black"
@sh "White"
@sh "Grey1"
@sh "Grey5"
@sh "Grey10"
@sh "Grey20"
@sh "Grey90"

sub mm name$
        gsave
        mk = mk+1
        gsave
        set lwidth .0001
        set lstyle 1212
        rmove 0 .4
        rline 0 -.8
        rmove 0 .4
        grestore
        marker name$ 1
        rmove .3 0
        rline .5 0
        rmove .2 -.15
        write name$
        grestore
        rmove 0 -.65
end sub
amove 9.5 22
@mm "circle"
@mm "triangle"
@mm "square"
@mm "diamond"
@mm "fcircle"
@mm "ftriangle"
@mm "fsquare"
@mm "fdiamond"
@mm "dot"
@mm "cross"
@mm "club"
```

```
@mm "heart"
@mm "star"
@mm "snake"

amove 13.5 22
@mm "dag"
@mm "ddag"
@mm "asterisk"
@mm "oplus"
@mm "ominus"
@mm "otimes"
@mm "odot"
@mm "trianglez"
@mm "diamondz"
@mm "wcircle"
@mm "wtriangle"
@mm "wsquare"
@mm "wdiamond"



sub fnt f$ n$
 set font rm
 set just right
 write f$
 rmove .5 0
 set just left font f$
 write n$
 rmove -.5 -.6
end sub

set hei .4
amove 2 14
@fnt "rm" "Roman"
@fnt "rmi" "Roman Italic"
@fnt "rmb" "Roman Bold"
@fnt "rmbi" "Roman Bold Italic"
@fnt "tt" "Typewriter"
@fnt "ttb" "Typewriter Bold"
@fnt "ss"  "Sans Serif "
@fnt "ssb"  "Sans Serif Bold"
@fnt "ssi"  "Sans Serif Italic"
@fnt "psc" "PostScript Courier"
@fnt "psh"  "PostScript Helvetica"
@fnt "psbd"  "PostScript Bookman Demi"
@fnt "psncsr" "PostScript New Century Schlblk Roman"
@fnt "pszcmi" "PostScript ZapfChancery-MediumItalic"
```

```
@fnt "pszd" "PostScript ZapfDingbats"
@fnt "pltr" "Plotter Triplex Roman"
@fnt "pldr" "Plotter Duplex Roman"
@fnt "plsr" "Plotter Simplex Roman"
@fnt "plge" "Plotter Gothic English"
@fnt "plci" "Plotter Complex Italic"
@fnt "plss" "Plotter Simplex Script"

amove 2 20
box 6 3
begin origin
  set hei .3 just left font rm
  amove .5 2.5
  for z = 0 to 4
     set lstyle z
     rline 2 0
     rmove .1 0
     write z
     rmove -2.1 -.4
  next z

  amove 3.5 2.5
  for z = 5 to 9
     set lstyle z
     rline 2 0
     rmove .1 0
     write z
     rmove -2.1 -.4
  next z

  amove .5 .2
  set lstyle 9229
  rline 3 0
  rmove .1 0
  text 9229
end origin

sub lw ww wn$
        set lwidth ww
        rline 1 0
        rmove .2 -.1
        write wn$
        rmove -1.2 .5
end sub
set hei .3 just left font ss
amove 3 16
@lw 0 "lwidth 0"
```

```
@lw .0001 "lwidth 0.0001"
@lw .01 "lwidth 0.01"
@lw .02 "lwidth 0.02"
@lw .05 "lwidth 0.05"
@lw .1 "lwidth 0.1"
@lw .2 "lwidth 0.2"
```