The ISIS

OPEN GENIE

User Manual

F A Akeroyd

R L Ashworth

S D Johnston

J M Martin

C M Moreton-Smith

D S Sivia

version 1.1

Forward

This manual should enable you to become familiar with *Open GENIE* quickly and easily. It therefore complements the *Open GENIE Reference Manual* which should be used to understand the full meaning of *Open GENIE* commands. The reference manual is accessible on the ISIS web server at:

- *http://www.isis.rl.ac.uk/GENIEReferenceManual/*

and the user manual is planned to be accessible by January ’98 at

- *http://www.isis.rl.ac.uk/GENIEUserManual/*

The Open GENIE User Manual is separated into two parts:

- **Part A**. The *User Manual*

  An introduction to the use of Open GENIE

- **Part B**. The *Installation Guide*

  General information on how to install and run Open GENIE.

Assuming that you are new to *Open GENIE* and have just downloaded a copy you will need to consult the *Installation Guide* to get *Open GENIE* installed on your machine. After this we recommend you experiment with some of the example files to get an idea of the capabilities of *Open GENIE*.

For further information, comments, additions of routines that you feel should be included, please contact us at *genie@isise.rl.ac.uk*.

To be kept up to date with new releases send a message to *genie-announce-request@isise.rl.ac.uk* with no subject, and the body of the message containing the two lines, - this will add you to the mailing list

```
SUBSCRIBE
QUIT
```

Table of contents

Part A - User Manual

What is *Open GENIE* ?

GENIE is the name of a programs used for the display and analysis of data from the neutron scattering instruments at the *ISIS facility*. For several years now, a stable VAX/VMS version of the software (GENIE-V2) has been in use at *ISIS* (Rutherford Appleton Laboratory, Oxfordshire, UK), and at several scientific and commercial establishments world-wide. Constant improvements in the capabilities of the neutron scattering instruments at ISIS, and in the hardware and software available for data analysis, has necessitated a re-write of GENIE. The new GENIE is known as *Open GENIE* to reflect the intention that the software be used on a wide variety of different computers and operating systems. It is aimed at supplying scientists with an inexpensive computer package which provides them with access to their experimental data, so that they may analyse the data as required and display the results in a useful format, usually in a graphical form.

Getting Started

If you are new to *Open GENIE*, we recommend that you work through "*The Three Wishes*" section of the *User manual*. This should give you a clear idea of what facilities *Open GENIE* provides and how to use them. After this working through one or two of the examples programs should provide sufficient knowledge to enable you to work with *Open Genie*. For the more experienced GENIE user, a brief read of the *User manual* is recommended to determine the differences between older versions of GENIE, and the new revised *Open GENIE*.

Starting up *Open Genie*

Remote operation of *Open GENIE* at ISIS

a) Via Telnet

Firstly *telnet* into one of the main VMS server machines *ptath*, *horus*, or *thoth* using

```
>telnet ptath.nd.rl.ac.uk
```

If you want to use the graphics you will need to set the X display back to your local machine

```
$set display/create/tran=tcpip/node=your.own.machines.ip.address
```

and start *Open GENIE* by typing

```
$opengenie
```

b) Via a terminal emulator

If you start remote sessions on a UNIX workstation via a terminal emulator such as *eXceed* or *eXcursion*, you can start *Open GENIE* directly by telling "*rexec*" to run the "*rgenie*" command. Note the full pathname may be required, for example

```
>/usr/local/bin/rgenie
```

The "*rgenie*" command can take several parameters, for further details type

```
>man rgenie
```

A typical use may be to start *Open GENIE* with your own "*genieinit.gcl*" file, for example

```
>rgenie “myinit.gcl”
```

An alternative is to use the terminal emulator to set up an *xterm* or a *decterm* on one of the UNIX or VMS workstations from which *Open GENIE* can then be run.

A few useful facts about *Open GENIE*

Commands

Commands in GENIE are entered via the command line. The command line is based on the GNU readline library and you will find that you can recall and edit previous commands by using the *arrow/cursor* keys. Commands are saved between sessions in a “`.genie_history`” file in your home directory, and this can be searched for previous commands. For features of the command line see the appendix A.

Commands in Open GENIE consist of either *assignments* or *keyword* commands. Assignments are made using the “=” sign, for example

```
<destination-location>=<expression>
```

The `destination-location` is a variable or an element of a compound variable such as an array or workspace, and the expression can be:

- a literal value e.g. 5.6 or “*hello*”
- a variable or part of a variable e.g. *Two_theta*, or *w1.npts*
- function expression e.g. *Sin($Pi)*
- unary or binary expression e.g. *-1*, or *1+2*

Any *Open GENIE* procedure can be implemented by a *keyword* command, and if the procedure returns a value it can also be invoked in an *expression*, where the value is *assigned* to a variable. A *keyword* command starts with a *keyword* followed by a list of *qualifiers* and *parameters*. The *keyword* specifies the basic command, while the *qualifiers* modify the action of the command and the *parameters* supply the values which the command operates on. For example, a *keyword* command is

```
>>display/Line w1
```

This command plots the workspace `w1` as a line. By itself the `display` command plots a histogram, the `/line` *qualifier* modifies the command to perform the plot as a line rather than a histogram, and `w1` is the parameter. When the *Open GENIE* procedure is implemented in an *expression*, in the functional style, *qualifiers* are specified by placing a colon separated list after the function name and before the parentheses containing the parameter list. For example, in the case of the `Asciifile()` command to open a file:

```
>>file=Asciifile:open("mydata.dat") # functional style
>>Asciifile/Close "mydata.dat"      # keyword command style
```

Note: comments can be included by using the '`#`' symbol before the comment, everything following this is ignored.

Data types

In *Open GENIE* you can create new data easily, and these can be independent from each other or grouped together into arrays or workspaces. The data can be either an *integer* number , a *real* number, or a character *string*.

Integers are used mainly for counting or specifying quantities, for example array indices.

Real numbers are always stored as floating point double precision values. They can be specified as a number with a decimal point or in scientific notation. Examples are

233.0 +0.5   -.01   33.   1.6e-19

Note: the type of *Open GENIE* data can change with the way it is assigned, for example

```
>R=1.0      # R is a real value of 1.0

>R=2        # R is now an integer
```

Strings may be of variable length and combined and assigned just like the other types of Open GENIE variables. Literal strings are always specified within double quotes e.g. "*my_quote*" to avoid any conflict with variable names. When the character "\" is used in a string it introduces control sequences, for example, "*n*", puts in a newline. For more see the section on *storage* in the *Open GENIE reference manual.*

Data types may be either *constants* or *variables*. *Constants* are prefixed by a '`$`', and can only have a value assigned to them once, usually when they are first created. All available constants can be viewed by typing:

```
>>Show/Const
```

System constants are prefixed by `$_` and can be viewed with

```
>>show/const/sys
```

Variables are created by assigning a value to an unused variable name. A variable name may consist of alphanumeric characters, the "_" character, or numbers, but must start with a letter. Examples are

```
>>A=2.0 ; B=6
```

To see all the defined variables type:

```
>>Show/Var
```

All variables are preceded by an underscore "_" are reserved for internal use by *Open GENIE* and these may change or be removed at any time, and therefore you should avoid using any variable starting with "_". The command

```
>>Show/Var/Sys
```

will show all system variables.

Arrays are structures that contain more than one variable. There are three array types one for each of the basic variable types. An array is created by first generating an empty array of the appropriate size and dimensionality using the `Dimensions()` function and then by assigning values to its elements. For example

```
# Create a RealArray
>>two_d_data = Dimensions(100,200) # created in 2-dimensions
>>two_d_data[1,1] = 5.0            # type is set here to real
>>Printn two_d_data
[5.0 _ _ _ _ ...] Array(100 200 )
```

Arrays may be manipulated as a whole or, alternatively, individual elements may be accessed and modified separately. Note: all elements which have not had a value assigned to them are marked as _. You can specify a value for undefined elements using the `FIX()` command.

Workspaces are compound variables, where individual elements of the variable maybe of different types (contrast with arrays, which are all of one type). Examples of similar structures in other languages are a *record* in Pascal or a *struct* in C. The following example shows the construction of a simple workspace interactively.

```
#  create a brand new workspace
>> mywork = fields()    # creates an empty workspace
>> Printn mywork        # look at it now
Workspace []
(
)
>>
# Now put some fields in it
>> mywork.description = "My test workspace"
>> mywork.some_value = 3.14159
>> mywork.anarray = dimensions(2,3)
>> mywork.anarray[1,1] = 25
>> Printn mywork         # now look again
Workspace []
(
some_value = 3.14159
anarray = [25 _ _ _ ...] Array(2 3 )
description = "My test workspace"
)
>>
```

Note: `structure()` may be used instead of `fields()`.

Operations

The normal operations on *integers* and *real* numbers are

- `x^y`. If x is an integer then the result will be an integer no matter whether y is fractional or not.

- `+`, `-`, `*`, `/`, and the remainder or modulo function `|`. For the quotient and remainder operations on integers the convention is to truncate towards zero.

- <, >, =, <=, >=, `!=`. The comparisons of: less than and greater than, equal to, less than or equal to, greater than or equal to and, not equal to respectively.

- `Sin(x)`, `Cos(x)`, `Tan(x)`, `Arcsin(x)`, `Arccos(x)`, `Arctan(x)`. The trigonometric functions, where `x` is quoted in radians. The constant `$PI` is available for conversion to degrees.

- `Log(x)`, `Ln(x)`, `Exp(x)` the Transcendental functions.

Note: the results of the trig. and transcendental operations on integers are given as real numbers, and when an expression contains integers and real numbers the result is given as real after implicitly converting any integer values to corresponding real values.

The normal operations on *strings* are:

- `+` or `&` concatenates two strings. For example,

  ```
  >>A= "Hello" + " there" & "!"
  ```

- <, >, =, <=, >=, `!=`. For strings of the same length, the comparisons act as a test based on the collating order of the text in the strings. For strings of different lengths, comparisons are between the string lengths.

- `Substring()`, `Locate()`, `Length()` for more information see *String Handling Functions* in the *Reference Manual*.

The basic operations on *arrays* are those of the corresponding data type plus:

- `Array[i,j,k,...]`. This *indexing* allows individual array elements to be accessed using integer indices. Note: all indices start at one. For example

  ```
  >>printn my_array[12,34,5,3,2]
  32
  ```

- `Array[l1:l2,l3:l4,...]`. This *slicing* of an array produces a smaller array with limits of `l1` to `l2`, `l3` to `l4`. If variables are used as limits extra brackets must be included. For example

```
>>a=my_array[2:4,4:100]
>>b=another_array[(i):(k)]
```

- & Appends the RHS array to the end of the LHS array, for this operation both arrays must be of the same variable type.

Note: the unary, trig. and transcendental functions are applied individually to each element of the array. The binary operations are applied to the elements of the two arrays, if the arrays differ in dimensionality, the corresponding elements are selected using storage order. The result of the operation is to create a new array of identical structure to the array on the left hand side of the operator. If the LHS array is longer, the extra elements are set to the undefined value *nil* in the result array, while if the LHS array is shorter the result array is truncated to the length of the LHS array. The comparisons test whether the elements and/or length differ.

The basic operations on *workspaces* are those of the corresponding data type plus:

- `<workspace>.<field-name>`. This *field accessing* allows its value to be read or set. A field is automatically created in a workspace when a value is assigned to it.

The comparisons when operated on workspaces assume that the workspace is an *Open GENIE* spectrum, but these operations can be modified by the user, see the *Open GENIE reference manual.*

File names

Open GENIE takes file names in either the VMS style or UNIX style depending on the system being used. For VMS

```
USER$DISK:[FAA.GENIE]TEST.DAT
```

or on UNIX

```
/usr/users/cmm/genie/test.dat
```

System commands from inside *Open GENIE*

The *cd* command changes the default working directory while in *Open GENIE*. For example:

```
# Change to the examples directory on UNIX
>>cd “/usr/local/genie/examples”
>>dir
```

Note: the directory path is as specified on the native operating system, e.g. VMS or UNIX.

The *dir* command lists the files in the current directory or the directory specified on the host operating system. For example:

```
# List the examples directory on VMS
>>dir “[.examples]”
```

The *pwd* prints the current working directory. For example

```
# Change to the examples directory
>>cd "/usr/local/genie/examples"
>>pwd
/usr/local/genie/examples
```

The *Os* command prints the operating system *Open GENIE* is currently running. For example:

```
# Print the current operating system
>>printn Os()
OSF
```

The *system* command allows you to execute a single command or a command session of the native operating system from within *Open GENIE*. For example:

```
# find some user details on VMS
>>system “SEARCH journal.txt \”flux\” “
IRS14142ZAB/NJR Flux tests  18-FEB-1997 12:00:01 163.2
IRS14143ZAB/NJR Flux tests  18-FEB-1997 13:43:40   6.1
>>
```

If no command is given as a parameter, the `System()` command starts a sub-shell process which is terminated by an *`exit`* on UNIX, or a *`LOGOUT`* on VMS systems. Note that on VMS some normal logical name definitions may be missing in the sub process.

Getting help

Online Help in *Open GENIE* is available in a number of ways:

- to see a list of all *Open GENIE* commands , press <*Ctrl*>*H* on a blank command line; if it pressed after a letter a list of all commands that start with that letter will be printed

- pressing <*tab*> twice will list all commands that match with what has already been typed so far. Therefore pressing <*tab*> twice on an empty command line lists all the commands.

- if the command name is not known use <*Ctrl*>*K* to perform a keyword search of the command descriptions

- if the command name is known, run it with the `/HELP` qualifier to get a description and list of parameters. For example

```
>> display/help
```

The Three Wishes

Most people using *Open GENIE* will wish to do one of three things

> Wish 1 - *Access* their data, either from a file or directly from the data acquisition electronics.

> Wish 2 - *Display* their data on the screen and make a hardcopy on a printer.

> Wish 3 - *Manipulate* their data, for example, to focus or normalise it.

These are described separately in the following sections, but first this….

A Simple Example

Below is a description of a simple *Open GENIE* session to access time of flight data taken on the High Resolution Powder Diffraction (HRPD) instrument which gives you some idea of the usage of *Open GENIE* commands.

```
$opengenie
Open GENIE V1.0 BUILD-23 [Linked] Thu Jan 2 15:26:39 GMT 1997
[library version] 1.0(Running on alpha:VMS:6.2 (built with DEC
C++ V5.3))
>>
```

for UNIX systems use

```
>genie
```

Now that *Open GENIE* is running, most people will wish to set up some defaults to allow easy access to files in the data area of the instrument they are using. If you always use one instrument, these defaults can be set up in a command file to be run automatically when *Open GENIE* starts. Normally *Open GENIE* looks for the file "*genieinit.gcl*" in your home directory, so the defaults can be placed in that file

```
>>set/disk “axplib$disk:”
Default disk: axplib$disk:
>>set/instrument “hrp”
Default instrument: hrp
```

```
>>set/directory "[OPENGENIE.GENIE.EXAMPLES.DATA]"
Default directory: [OPENGENIE.GENIE.EXAMPLES.DATA]
>>set/extension "raw"
Default filename extension: .raw
```

On UNIX, the commands for setting defaults are similar and use the appropriate syntax for UNIX file names, except no `Set/Disk` command is needed. However, on all operating systems it is essential to put string arguments in double quotes.

```
>>set/instrument "hrp"
Default instrument: hrp
>>set/directory "/usr/local/genie/examples/data/"
Default directory: /usr/local/genie/examples/data/
>>set/extension "raw"
Default filename extension: .raw
```

Any raw set of data can then be collected using the *`Assign`* command followed by the run number

```
>>assign 8639
Default input: /usr/local/genie/examples/data/hrp08639.raw
```

An alternative is to use the `Set/File/Input` command which explicitly sets the file to be used. For example,

```
>>Set/File/Input "/usr/local/genie/examples/data/hrp08639.raw"
Default input: /usr/local/genie/examples/data/hrp08639.raw
```

Now that the source for the data has been selected individual spectra can be extracted from the data set using the `spectrum` function, which can be normally abbreviated to "*`s`*".

```
>>w = s(7)
Reading spectrum 7 /usr/local/genie/examples/data/hrp08639.raw
>>noise = s(1)
Reading spectrum 1 /usr/local/genie/examples/data/hrp08639.raw
>>corrected = w - noise
```

Here the seventh spectrum is read into the workspace *`w`*. The first spectrum in the file is subtracted from the spectrum in *`w`* and the result is put into the workspace *`corrected`*. The corrected spectrum can now be displayed on a graph using the `display` command, and the binning number altered to 10

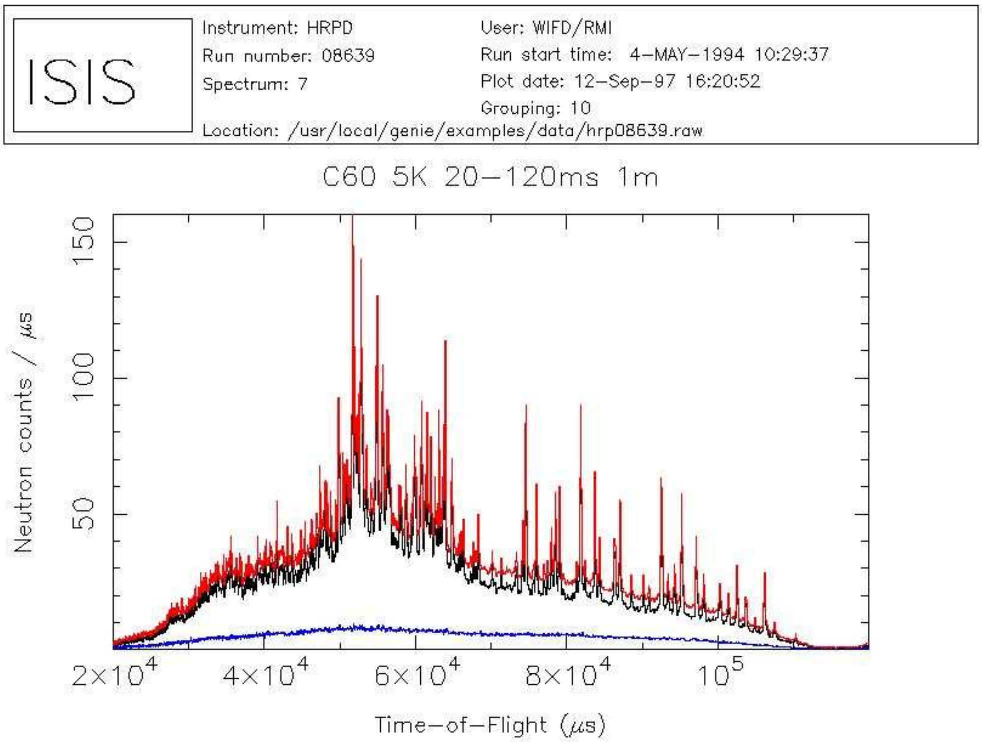
```
>>alter/binning 10
```

```
>>display corrected
Opening graphics device XW
Displayed using bin-grouping of 10
```

To plot the two original spectra on the same plot, but using different colours to show them clearly

```
>>alter/plotcolour $blue
>>plot background
Plotted using bin-grouping of 10
>>alter/plotcolour $red
>>plot w
Plotted using bin-grouping of 10
>>
```

The plot produced is shown below.



This brief example should have given you a good idea what an *Open GENIE* session looks like and a mental framework into which the more detailed information in the next three sections will fit. The data file plotted here is typically available in the "`/usr/local/genie/examples/data`" directory on UNIX or `AXPLIB$DISK:[OPENGENIE.GENIE.EXAMPLES.DATA]`, and by changing the file locations in the example above, you can reproduce this plot. Note: when `help` is typed from within *Open GENIE* you will be told where *Open GENIE* is installed, and from that you can deduce the position of the `examples` directory.

The First Wish - Access to Data

At *ISIS* and all other neutron scattering centres data is collected in '*runs*'. These are typically periods of time in which the sample is kept in constant conditions. The data for a run is stored in a raw data file, and at ISIS the naming convention is <*instrumentxxxx.raw*>, where the run number is *xxxx*. The raw file itself contains one or more time-of-flight spectra and other associated parameters needed for the analysis of the data. This section describes the commands necessary to access data from *Open GENIE*. The description that follows is based on the data formats found at ISIS. However *Open GENIE* can be modified to read other formats used at other facilities. This section details all the commands used to access data from *Open GENIE*.

Accessing raw data spectra

One of the major strengths of *Open GENIE* is the flexibility available in accessing data. The first way of accessing data is based on the old GENIE-V2, where several internal variables control the

a) current input file,

b) current output file,

c) current disk,

d) current directory,

e) current instrument,

f) current file extension

The status of these variables is changed by using the `set()` command (detailed on page 12). The `show/defaults` command gives the input/output defaults, for example

```
>>show/defaults
Current default disk = AXPLIB$DISK:
Current default directory = [OPENGENIE.GENIE.EXAMPLES.DATA]
Current default instrument = HRP
```

```
Current default extension = .RAW
Current default input =
AXPLIB$DISK:[OPENGENIE.GENIE.EXAMPLES.DATA]HRP08639.RAW
Current default output =
>>
```

To read a spectrum or multiple spectra from a raw data file the `assign` and `spectrum ()` commands can be used. An alternative to this is to use the `Get()` command. The `Get ()` command is able to read data from either a raw data file, or another of the supported data formats, see Appendix B. However, this assumes that the data items in a file are either named explicitly, or numbered. For example, the data read from a raw data file could be a spectrum or a user name. The most useful parameters, which can be taken from a raw data file are `TITL`, the run title; `USER`, the user information; `NAME`, the instrument name; `NDET`, the number of detectors; `TTHE`, the two theta table; `NSP1`, the total number of spectra; `NTC1`, the number of time channels; `LEN2`, `L2` for each detector, and each individual spectra file are specified by an associated integer, i.e. specifying 5 would get the fifth spectra from within a file. For example:

```
# print the user name
# then read in all the spectra in the file,
# finally read every third spectrum from a file
>>printn get("USER",
"/usr/local/genie/examples/data/hrp08639.raw")
WIFD/RMI
>>d2 = get(1 : get("NSP1"))
>>spin_up =get(1:50@3)
```

Note: if a file has been set using the `Set/File/Input` command then if no file is stipulated it defaults to this file.

To save *Open GENIE* variables to an intermediate binary file the `Put()` command is used. These variables may optionally be tagged with a label, which can be used for later reading, or comments may also be added. For example,

```
# Copy a whole multidimensional spectrum from a raw file into a
single item in an intermediate file
>> Set/File/Input "/usr/local/genie/examples/data/irs12839.raw"
>> Set/File/Output "example.in3"
>> Put get(1:20) label = "bigspec" comment = "demo"
```

which can then be read with

```
>> get ("bigspec")
```

Accessing ASCII files

To read in data from an ASCII file, for example from the file "*prs_001.obs*", which can be found in the *PRISMA* directory of the *Open GENIE* example area

```
# Read in three data arrays of 30 points (in columns separated
by a whitespace)
# and assign to arrays in fields X, Y, and E in the new
workspace.
>>handle=Asciifile:Open("/usr/local/genie/examples/prisma/prs_0
01.obs")
>>wk1=Asciifile:Readfree(Handle, "X,Y,E",$whitespace, 30)
>>printn wk1
  Workspace []
  (
    x = [0.925 0.975 1.02499999999999 1.07499999999999
1.125...] Array(30 )
    y = [0.22526098 0.11728537 0.126243569999999
0.144312439999999 0.15559034 ...] Array(30 )
    e = [0.009967181499999 0.0043424088 0.004095190999999
0.003976735 0.0038093579 ...] Array(30 )
  )
>>
```

or alternatively

```
>>Asciifile/Readfree Handle "X,Y,E" $whitespace 30
>>wk1=Asciifile:data(handle)
>>
```

The second method enables several different items of data to be accumulated into a workspace before it is returned. For example:

```
>>Asciifile/Readfree Handle "X1,Y1,E1" $WHITESPACE 4
>>Asciifile/Readfree Handle "X2,Y2,E2" $WHITESPACE 4
>>wk1 = Asciifile:data(Handle)
>>printn wk1
  Workspace []
  (
    x1 = [1.0 1.1 1.2 1.3 ] Array(4 )
    y1 = [2.1 2.1 2.2 2.0 ] Array(4 )
```

```
  e1 = [0.0 0.1 0.2 0.0 ] Array(4 )
  x2 = [1.4 1.5 1.6 1.7 ] Array(4 )
  y2 = [2.0 2.0 2.0 2.0 ] Array(4 )
  e2 = [0.1 0.1 0.2 0.0 ] Array(4 )
)
```

Note: you can also use the `Put/Ascii` command to output data to an `Asciifile`.

Other commands concerned with the *input* and *output* of data can be found in the *Open GENIE* reference manual.

The Second Wish - To Display Data

Opening a graphics device.

Examples of the use of many of these commands may be found in the *gcl* procedures in the `/usr/local/genie/examples` directory on ISIS UNIX systems or `AXPLIB$DISK[OPENGENIE.GENIE.EXAMPLES]` on ISIS VMS.

Before you can plot any data, you need to select a device on which to display the data.

```
>>Device/open “device-type” width height
```

The default device is an X-window “*XW*”. Other supported devices include “*TEK4010*” Tek 4010 compatible, “*PS*” postscript, “*CPS*” colour postscript, and “*HPGL*” HPGL driver for 7475A. For full details of the supported devices on each machine see the supported graphics devices in appendix B. The size of the page (in inches) is defined by the *width* and *height* parameters the default value is taken to be 8 inches. Alternatively you can open a device with a name:

```
>>device1=Device:Open (type, width, height)
```

This is useful as *Open GENIE* supports multiple open devices, a given window can then be selected using the `Select` command. For example

```
>>Select Device1
```

Displaying a graphical plot of spectra from a workspace.

After the data has been read into a workspace from a *RAW* file or intermediate *GENIE* file, the data can then be plotted using the *display* command. The plot will be automatically scaled and titled with other relevant information from the workspace. For example:

```
# Display a spectrum but change the title first
>>wm = s(10)
>>wm.title = “This is my Nobel plot!”
>>display wm 0 100 -35 70
```

Alternatively you can plot it directly using

```
>>display s(10)
```

which displays the 10th spectra of the default input file, or alternatively use the `Get ()` command

```
>>Display get(10,“hrp00273.raw”)
```

The parameters after the *workspace wm* are respectively *xmin*, *xmax*, *ymin*, *ymax*. All the parameters, except for the workspace, are optional and where omitted a default value will apply. Specifying a keyword for each parameter allows you to define each individual limit. For example, if the X limits should be defaulted, but the Y limits defined, then you could use:

```
>>Display wm ymin=-35 ymax=70
```

The parameters may also be expressions using functions of variables, for example

```
>>Display w1-noise 0 xmax/100.0 ymin=Min(w1.y) ymax=Max(w1.y)
```

Anonymous placeholders may also be used to indicate a default value - in *Open GENIE* the character ‘_’ is used. The above example could then be written as

```
>>Display w1 _ _ -35 70
>>
```

Other optional parameters that may be specified are:

Linecolour - the colour for drawing the axes and surrounding boxes,

Linewidth - the line width used for drawing line plots,

Textcolour - the colour used for drawing text.

Other qualifiers:

`/HISTOGRAM` - displays as a histogram assuming that the binned data has one more X value than Y value.

`/LINE` - displays a spectrum as if it is a numerically calculated function (i.e. with the same number of X-values as Y-values) and join the points with straight lines. Note: if histogram data is displayed with the `/Line` qualifier, the data will be transformed to take the bin centre positions as the X-values for the data points, this is done with the `Centre_bins()` function.

`/MARKERS` - displays a plot with markers symbols. For a list of *marker types* see appendix D.

`/ERRORS` - display as error bars taken from the workspace error array.

In order to clear the display surface on the currently open device:

```
>>Device/Clear
```

While

```
>>Device/Close
```

deactivates the currently open graphics device.

Performing a multiplot of *2-D* spectra on the same x-axis

This is done using the `Multiplot` command, and it is useful for comparing groups of spectra from different detectors. A pseudo Y-axis is produced where successive plots are placed with an artificial Y offset value (ygap). Currently, multiplot only plots as a histogram. If used to plot a spectra from a raw file then the data source for the multiplot can be specified using the `Set/File` command. For example:

```
# Do multiplots from a file
>>multiplot 1:20 file = “mydata.raw" ygap=10.0
>>multiplot 1:300@3    # plot every third spectrum
```

Optional parameters (note the types) are: Linecolour (Colour), Linetype (LineStyle), Linewidth (Real), Textcolour (Colour1), Textheight (Real).

Overplotting an existing graph

This is done using the `Plot` command, which does not have the ability to create axes or scale the data.. For example:

```
#Plot a vanadium spectrum on top of
#the data as a comparison
>>Display s(5) xmin=20000.0
>>Plot/Line get("vanadium1","normal.in3")
```

Qualifiers: `/Histogram`, `/Line`, `/Markers`, `/Errors`

Altering the graphics parameters in the `Display()`, `Plot()` and `Multiplot()` commands

Modifying characteristics of the plot is achieved using the commands `Alter()`, `Toggle()`, or `Limits()`.

`Alter/Status`	Reports the values of all settings.
`Alter/Binning`	Sets bin grouping for plotting. If the bin grouping value is set to a value greater than one, the display command groups the data by averaging the values of *n-bin* groups. It is possible to apply the same operation to a standard workspace permanently using the `Groupbins()` command.
`Alter/Device`	Changes the interactive device.
`Alter/Font`	Sets the text font for all the text used in the display.
`Alter/Hardcopy`	Changes the default hardcopy device, when a `Hardcopy()` command is issued.
`Alter/Linecolour`	Changes the axes and box colour
`Alter/Linetype`	Changes the line style for line plots
`Alter/Linewidth`	Changes the overall line thickness

`Alter/Markers`	Sets the marker plot symbol
`Alter/Markersize`	Changes the size of markers
`Alter/Plot`	Changes the plot size
`Alter/Plotcolour`	Changes the data plot colour
`Alter/Size`	X window display size by a scale factor
`Alter/Textheight`	Text height as the percentage height of the display
`Alter/Textcolour`	Label and title colour

For example:

```
# Change the axis colour to blue
>>Alter/linecolour $BLUE
```

For a list of *supported graphics devices*, supported colours, supported fonts, supported linestyles, and supported markers see appendix D.

The `Toggle` command can be used to amend plots, acting like an on/off switch for various settings. All the toggle commands switch the setting to its opposite value unless `/Off` or `/On` is specified.

`Toggle/Status`	Reports the values of all toggle settings.
`Toggle/Logx`	X axis log/linear
`Toggle/Logy`	Y axis log/linear
`Toggle/Clear`	Screen clearing between plots
`Toggle/Graticule`	Toggle graticule off/on
`Toggle/Header`	Header plotted on Display() command.
`Toggle/RX`	X axis rounding off/on

`Toggle/RY`	Y axis rounding off/on
`Toggle/Info`	Informational message printing on/off
`[/On]`	Toggles item on
`[/Off]`	Toggles item off

For example:

```
# Switch off informational messages and display
# a data plot with no banner heading.
>> Toggle/info
>> Toggle/header
>> Display s(22)
```

The limits of the graph can be changed using the `Limits()` command. For example:

```
# Set the limits before doing a display
>> Limits/X xmin=10.0E4 xmax=50.0E4
>> Display # Redisplays previous plot with new limits
```

The command `Limits/Default` clears this. Note the actual setting of the limits will still be dependent on whether rounding is switched on for the X-axis.

Magnify a portion of the plot chosen interactively

This is done using the `Zoom` command, which redraws a plot scaling to a box which is selected by the cursor (the lower left-hand corner and the upper r.h corner of the box). The qualifiers `/Errors`, `/Histogram`, `/Line`, `/Markers`, will redraw the plot using error bars, histograms, lines, or markers respectively. While the `/Multiplot` qualifier expands a region of a multiplot. For example:

```
# Zoom in on the current plot
# and re-display with error bars
>> zoom/errors
```

Plotting data from an ASCII file

In this example the data used is *prs_001.obs*, which can be found in the directory of "*../genie/examples/prisma*". To construct a plot if you have read in a non *Open*

GENIE type file, e.g. an ASCII "X,Y,E" file into `w.x`, `w.y`, `w.e`, then after opening a new graphics device using

```
>>Device/Open
```

a window must be created with the `Win_` commands. In this case a Scaled or AutoScaled window is created by

```
>>Win_Scaled 0.1 0.9 0.1 0.9 0.975 7.475 0.02 0.17
```

The first four numbers define the *device* co-ordinates - the position of the plotting box on the page, and the last four numbers are the *global* co-ordinates - the X and Y range of the plot. The points of the graphical device range from 0.0 and 1.0. By using the `Win_Autoscale` command, and quoting the x and y arrays in the parameter list the range of the plot are chosen automatically; for example:

```
>>Win_Autoscaled 0.1 0.9 0.1 0.9 w.x w.y w.e $Red
```

where `w` is the workspace containing the data, and the `$red` parameter will paint the chosen window red instead of the default black.

Now we can create the graph

```
>>Graph/Draw w.x w.y
```

green axes can be drawn by

```
>>axes/draw colour=$green
```

By default the axes are linear, and the x and y axes are labelled horizontally and vertically respectively. To display x/y labels on the graphics device

```
>>labels/draw “X Axis” “Y Axis”
```

To draw text on the graphics device

```
>>text/draw 0.0 1.0 “some text” colour=$yellow size=3.0
angle=45.0 font=$script
```

the position is specified in world co-ordinates.

To display a title on the graphics device

```
>>title/draw “A Title”
```

Comparing this data with the data from the file *prs_001_lmf.cor* (after loading the ASCII data file into the w1.x, w1.y and w1.e) plot the data with markers as blue crosses

```
>>markers/draw w1.x w1.y $Blue $Cross 2
```

To include Y error-bars

```
>>errors/draw/Vertical w.x w.y w.e $Blue 2
```

and to draw a green graticule on the window

```
>>graticule/draw colour=$green
```

Note: for a complete list of qualifiers associated with these commands see the *Open GENIE reference manual.*

To ensure the above picture and its associated objects can be referred to later, after a new picture has been created then initially a picture should be created,

```
>>picture1=Picture()
```

and then the following commands used

```
>>scaled1=Win_Scaled(0.1,0.9,0.1,0.9,0.975,7.475,0.02,0.17)
>>graph1=GRAPH:DRAW(w.x, w.y)
>>axes1=axes:draw(colour=$green)
>>labels1=labels:draw("X Axis", "Y Axis")
>>text1=text:draw(0.0, 1.0, "some text", colour = $yellow
size=3.0, angle=45, font = $script)
>>title1=title:draw("A Title")
>>markers1=markers:draw(w1.x,w1.y,$Blue,$Cross,2)
>>errors1=errors:draw:Vertical(w.x, w.y, w.e $Blue 2)
>>graticule1=graticule:draw(colour=$green)
```

Now to view this plot open a new device

```
>>device1=device:open()
>>Redraw picture1
```

Making alterations

If you wish now to change part of the graphs for example the text in picture1

```
# Want the font to be italic
>>title/alter font=$italic object=title1
```

or you want to change the graticule

```
>>graticule/alter colour=$red object=graticule1
```

In order to remove the changes caused by the `/draw` qualifier the `Undraw` command can be used. For example, if you wanted to *undraw* markers1 then type

```
>>Undraw markers1
```

Note: if the `Undraw` command is followed by a negative integer, *N* then it is the previous N^{th} `Draw` that is undone. In other words, `Undraw -1`, undoes the previous `Draw` command (the default); whereas `Undraw -3` undoes the command two before the last one.

For a whole set of pictures and opened devices made for example *picture1*, *picture2 picture3 picture4* etc. and *device1*, *device2*, *device3*, *device4*, etc., then to redraw *picture4* on *device5*, and *picture 6* on *device2* then type

```
>>Redraw picture4 device5
>>Redraw picture6 device2
```

In order to redraw a previous plot then type

```
>>Redraw -1
```

After a number of `device:open()` commands have been executed then to redraw the current picture on the current device type

```
>>Redraw 0.
```

Display a cursor on the graphics screen to allow annotation

This is done using the `Cursor` command. An obvious use is for tagging peaks with the data value in the X or Y direction at the point selected by the cursor. For example

```
# Display a cursor near the expected position
# of a peak in world co-ordinates.
>>Cursor/Horizontal 10000.0 50.0
```

```
# If /vertical is used instead then the text is drawn
vertically
```

a cursor is displayed and waits for a pressed key, if X - displays "*X co-ordinate*"; Y - displays "*Y co-ordinate*"; P - displays both co-ordinates; T- to add text; and E-exit. See also the `GetCursor` command in the *Open GENIE* reference manual.

To save a hardcopy of the graphics into a file

The `Hardcopy` command allows a copy to be made of either the currently displayed window, or an earlier picture to a postscript file. For example:

```
# Save a copy of the last but one picture as
# colour postscript (found with Redraw/Info)
>>Hardcopy filename = "test.cps" picture = my_pict
```

The default name of the file into which the hardcopy is placed is "*GENIE.PS*". The filename extension is one of the supported hardcopy graphics devices. For full details of supported devices on each machine type see the appendix. The default picture selected is 0 - the last picture created. Normally just typing `Hardcopy` is sufficient to make a hardcopy of the current screen.

Procedures

A whole series of graphics commands can be strung together and stored as a procedure. See for example the various directories of `/usr/local/genie/examples`. The procedures are then loaded using the command

```
>>Load "File.gcl"
```

and executed using the command

```
>>File
```

The Final Wish - To Manipulate Data

This section can be separated into three distinct areas

- mathematical functions
- workspace operations
- data analysis functions

Mathematical functions

As well as providing intrinsic arithmetic operations for different data types *Open GENIE* provides several basic functions for performing mathematical operations. These are coded generically so that the same function can be applied to any data type which is capable of undergoing the operation. For example, the `Sin()` function may be used on a single number, an array or a workspace. Normally, the result of the operation is of the same data type as the value operated upon. Where the operand contains several values (e.g. an array), each value is operated on individually. For example we can create an array of real numbers, and square root them

```
>>my_numbers = Dimensions(10,20) # First create a 10 x 20 array
>>fill my_numbers 1.0 1.0       # Fill the array with some data
>> printn my_numbers
[1.0 2.0 3.0 4.0 5.0 ...] Array(10 20 )
>> printn sqrt(my_numbers)       # print the square roots.
[1.0 1.414213 1.732050 2.0 2.236067 ...] Array(10 20 )
```

The basic mathematical functions are

- trigonometric functions - with the angle in radians

  `Arccos()`, `Arcsin()`, `Arctan()`,`Cos()`, `Sin()`, `Tan()`. For example,

  ```
  # print result in degrees
  >>y = Arccos(0.5) * 180 / $pi
  # take the arccos of all elements in an array
  >>a = Dimensions(10)      # create a 10 element array
  ```

```
>>fill a 0.5             # Set all elements to 0.5
>>y = Arccos(a)
```

- transcendental functions

  `Exp()`, `Ln()`, `Log()`. For example,

  ```
  # antilog a data array
  # which is in logs to base 10
  >>data = Exp(w.y*ln(10.0))
  ```

- other functions

  `Abs()`[1], `Sqrt()`. For example:

  ```
  # Calculate the square root of Pi
  >>printn Sqrt($PI)
  1.77245310234149
  ```

---

[1] *Abs()* calculates the absolute value of a number or of the numbers in an array. All negative numbers will be returned as positive numbers of the same magnitude.

Workspace operations

Workspace operations and transformations form the heart of *Open GENIE*. These allow analysis of the data taking into account its underlying form. For example, the `Units()` command can be used to convert the units of a time-of-flight (TOF) spectrum. This is only possible if some assumptions are made about the fields which are present in the workspaces containing the experimental data. For the `Units()` command to work, a workspace also requires fields giving parameters such as the primary flight path and incident angle (a TOF spectrum).

For example for a 2-D TOF Spectrum needs

Workspace Field	**Description**	**Variable type**
X(1-D)	array of X values	RealArray
Y(2-D)	2-D array of Y values	RealArray
E(2-D)	2-D array of errors for Y field	RealArray
L1	primary flight path (m)	Real
L2(1-D)	secondary flight path (m)	Real array
Twotheta(1-D)	scattering angle (degrees)	Real array
Delta	hold off in microseconds	Real
Emode	energy mode	Integer 0=inelastic, 1=incident, 2=transmitted
Efixed	fixed energy (if applicable)	Real
Xlabel	Units for X values	String
Ylabel	Units for Y values	String

Ut(1-D)	User parameters (this array may be of any length and caters for information not already named in a field)	RealArray

For more on the necessary fields required for operations then see the *Open GENIE* reference manual.

Template routines are called whenever an intrinsic operation involving workspaces is specified. In most cases the errors are propagated through the calculation. The template routines on workspaces are:

- Unary operations

  Trigonometric and transcendental functions. For example

  `w2=ln(w1)` is equivalent to `w2=workspace_ln(w1)`

  and

  `Workspace_arccos(w)` is equivalent to `acos(w)`,

  Other unary functions

  `Workspace_sqrt`, `sqrt(w)`; `Workspace_abs`, `|w|`; and, `Workspace_negated`, `-w`.

- Binary Operations

  `Workspace_add()`, `w1+w2`; `Workspace_append` (joins one end of one histogram to another), `w1&w2`, `Workspace_divide`, `w1/w2`; `Workspace_raised_to`, `w1^w2`; `Workspace_subtract`, `w1-w2`; `Workspace_modulo`, `w1|w2`; `Workspace_multiply`, `w1*w2`.

  The characteristics (i.e. final length, dimensionality of arrays) of the left hand workspace define those of the resultant workspace.

- Comparison Operations

  `Workspace_equal`, `w1=w2`;

  `Workspace_not_equal`, `w1!=w2`.

The definition of the default routines are stored in the file "*workspace_user.gcl*", which is included in the library directory of the *Open GENIE* distribution.

Data Analysis Functions

Arguably the most important purpose of *Open GENIE* is as a tool for scientific data analysis. Not just for preliminary data analysis, but as a framework which provides, or from which can be called, all the tools necessary to perform a complete analysis of the data.

Currently, a large amount of data analysis is performed with a set of disparate FORTRAN programs which have to support their own routines to access data, to drive a user interface and to work with a variety of different graphics packages. What is critical and is very much at the heart of the philosophy of *Open GENIE* is that *the scientist must have control over is how the data is processed.*

A scientist may trust the *Open GENIE* peak fitting routines, alternatively and probably more the case at the moment, the scientists will want to continue to use their own analysis code in a C or FORTRAN subroutine for most things. *Open GENIE* can be used to provide the user interface, graphics and file access. This mechanism is described fully in the next section of this manual on *modules*, and these allow the code to be run as if it was a compiled part of *Open GENIE*. Routines which have been found to be useful in several different areas of analysis are hard coded in C/C++ or FORTRAN for efficiency.

The current routines available are:

- `Focus()`

Focus a range of time of flight spectra from detectors at different scattering angles with given parameters. This command, although written specifically for focusing multiple banks of detectors consecutively can be used for single scanning detectors data. To focus the spectra in *d-spacing* (with the `:D` switch), or in *momentum transfer* (with the `:Q` switch) the time of flight parameters are required, which can be specified from a file (the default are the values from the raw file). Note: the default is the `:D` switch where spectra are summed in *d*-space. For example:

```
# Focus a time of flight spectrum in D-spacing
# get the TOF parameters from the raw data file
```

```
>>w = focus( 1:17, “irs12838.raw” )
# Focus default file, every third period
>>assign 3045
>>refl = focus(3:90@3, _, mydet)
```

- `Integrate()`

Integrate a *1-D* or *2-D* spectra. For example:

```
# print the integrals of two spectra
>> printn integrate(1:2, 38000, 78000, "tfx00345.raw")
INTEGRATE: Integrating between 38000 and 7800
  Workspace []
  (
    error = [2341.4700 1027.1312 ] Array(2 )
    sum = [5482585.0 1054968.0 ] Array(2 )
  )
```

The result of the integrate command is returned as a workspace with two fields: the ‘`SUM`’ field contains either a single value or an array of integrals, and the ‘`ERROR`’ field contains the propagated error of the result.

- `Peak()`

Provides interactive peak fitting of a workspace allowing selection of the bounds of the peak and the form of fit to use. The available fits for use are:

1. Gaussian - a Gaussian-peak sitting with a straight-line background,

2. Gaussian convoluted with an exponential - a Gaussian convoluted with a sharp-edged exponential, sitting on a straight-line background,

3. Lorentzian - a Lorentzian-peak, sitting on a straight-line background,

4. Lorentzian convoluted with exponential - a Lorentzian convoluted with a sharp-edged exponential, sitting on a straight-line background,

5. Voigt - a Voigt convoluted with a sharp-edged exponential, sitting on a straight-line background,

6. Polynomial - fits an n^{th} degree polynomial,

7. Voigt convoluted with an exponential - a Voigt convoluted with an exponential.

Once fitted the peak parameters of the fit are displayed and made available as a result, so that they can be used later. For example:

```
# Fit peaks from a genie intermediate file
>>fit = peak( get(3, "mydata.in3"))
>>printn fit
```

Note: See the `Peakfit()` and `Peakgen()` in the *Open GENIE reference manual* for more control and detail.

- `Rebin()`

The rebin command performs an interpolation from one histogram into another histogram with the same integrated count, but with modified X-boundaries. Some numerical precision is lost during a re-binning operation and there is usually a degree of peak broadening, so multiple re-binning is to be avoided. The `/Lin` and `/Log` qualifiers allow a *Linear* or *Logarithmic* rebinning between the specified boundaries to be performed. For linear rebinning the boundaries are specified in steps, for logarithmic bin widths these are calculated such that for any given binning range

$$X_{n+1} - X_n = Step * X_n$$

For example, given the command

```
>>w = rebin:log(w, 10.0, 0.1, 15.0)
```

Here the step is 0.1, and the range of the rebinning is between 10 and 15. The bin boundaries generated will be `[10, 11, 12.1, 13.31, 14.641, 15].`

A workspace can also be rebinned according to another workspace. For example:

```
# Read a spectrum and re-bin the same as an
# already loaded vanadium workspace
>>w=s(1)
>>w=rebin(w, vanadium.x)
```

- `Units()`

This command converts the units of time-of-flight spectra. Using the switches they can be converted respectively to:

`/C or /Channel` to Channel numbers (one way)

`/D` To D-Spacing (A)

`/E` To Energy (meV)

`/LAM` To Wavelength (Å)

`/Q` To Momentum Transfer ($Å^{-1}$)

`/SQ` To (Momentum Transfer)^2 (A^{-2})

`/T` To Time of Flight (μs)

`/LA1` To primary flight path wavelength

`/W` To energy transfer E1-E2 (meV)

`/WN` To energy transfer E1-E2 (cm^{-1})

`/TAU` To reciprocal time-of-flight (μs/m)

For example:

```
# Read in a spectrum whilst converting to Wavelength
>>w = units:Lam(s(1))
```

The *t.o.f* parameters must be set correctly in the input workspace (or set using the `Set/Par` command) in order to perform the conversion, and if the keyword syntax is used, i.e. `Units/C w` the workspace will be converted destructively.

The *Open GENIE* Module Interface

This section will answer the following questions:

- What is a module?
- When would I want to use modules?
- How do I write a module
- How do I load and run a module
- Where can I find out about other (pre-compiled) modules

What is a module?

A module is a compiled FORTRAN program that can be loaded into, and become part of, a running *Open GENIE* process. Subroutines in the module, provided they follow a given set of rules, can be called to manipulate arbitrary *Open GENIE* variables. A module is usually implemented as a shared library or dynamic linked library.

When would I want to use modules?

Modules are used when:

- A numerically intensive task must be performed for which GCL would be too slow and for which there is no in-built *Open GENIE* command
- You already have a working FORTRAN program for the task and do not wish to rewrite the application in *GCL*
- You wish to use *Open GENIE* as an ‘add on’ to another program to provide, for example, the input to the program, or to display the output.

How do I write a module

A module consists of subroutines only - there is NO main program. These subroutines can be divided into two types: those that perform the actual calculations, and those that pass data to/from *Open GENIE* and the calculation routines; the latter called “*wrappers*” as they provide a jacket for the original calculation routines.

1. Write FORTRAN Subroutines to do your tasks

The subroutines can take any parameters you desire. Our example will be based on the subroutine “*myfunc.for*” in the examples directory:

```
C
C *** The function - just square the X array
C
      SUBROUTINE MYFUNC(X, Y, NPT)
      IMPLICIT NONE
      INTEGER NPT, I
      REAL X(NPT), Y(NPT)
      DO I=1,NPT
      Y(I) = X(I) * X(I)
      ENDDO
      RETURN
      END
```

2. Write a FORTRAN “*wrapper*” subroutine to interface between *Open GENIE* and your calculation subroutine

Open GENIE variables are passed to and from the module using a temporary workspace. Data is placed into the input workspace with chosen field names, and then accessed in FORTRAN by using this field name and an access routine appropriate for the data type being sent. For example, the *frame count* could be stored in the variable “*NRAW*” and accessed using the `module_get_int()` function in FORTRAN. Returning data is similar - the various `module_put()` FORTRAN subroutines allow the construction of a returned workspace containing various data types with named variables. The wrapper subroutine must obey the following rules:

- It must be a SUBROUTINE taking just two parameters, called `PARS_GET` and `PARS_PUT`. These parameters should be declared `EXTERNAL` and not be accessed in anyway, except via “`module_...`” commands.
- It must allocate the variables needed by the user’s SUBROUTINE and transfer data to/from them using “`module_get_...`” and “`module_put_..`” calls.
- As input/output devices are re-assigned by *Open Genie*, `READ(5,*)` commands should be avoided and screen output should be handled via the “`module_print`”

and “`module_information`” commands. Inside the module *Open GENIE* variables are accessed by workspace field names. For example, in the program below the array X is loaded from the field labelled ‘*XVALS*’, where it has been placed by a GCL procedure. There is no restriction on how you label a variable, but often you will use the variable name itself. All “`module_get_..`” routines must specify `Pars_Get` as the first parameter, and similarly all “`module_put...`” routines specify `Pars_Put` (these variables are, in fact, just references to the input/output parameter workspaces of the `Module/Execute` command.

For a full list of `module_.()` functions see the FORTRAN module interface in the *Open GENIE reference manual.*

For our example “*myfunc*” the wrapper would be:

```
C *** the wrapper subroutine for MYFUNC
C *** input into array X, output into array Y
C
      SUBROUTINE DO_MYFUNC(PARS_GET, PARS_PUT)
      IMPLICIT NONE
      INTEGER NPT_MAX, NPT
C *** We can handle NPT_MAX data points
      PARAMETER(NPT_MAX=1000)
      REAL X(NPT_MAX), Y(NPT_MAX)
      EXTERNAL PARS_GET, PARS_PUT
C *** print a "blue" message to the user
      CALL MODULE_INFORMATION(“Inside module --- MYFUNC called!!!”)
C *** Set NPT to the max. size of the array
C *** On return, NPT will be the number of data points passed
      NPT = NPT_MAX
C *** The input was placed in a workspace field called 'XVALS' in GCL
      CALL MODULE_GET_REAL_ARRAY(PARS_GET, 'XVALS', X, NPT)
      IF (NPT .GT. NPT_MAX) THEN
      CALL MODULE_ERROR(“myfunc module”, 'Too many points”, “ ”)
      RETURN
      ENDIF
      CALL MYFUNC(X, Y, NPT)
C *** now return the result (Y) in a workspace field called 'YVALS'
      CALL MODULE_PUT_REAL_ARRAY(PARS_PUT, 'YVALS', Y, NPT)
      RETURN
      END
```

3. Make the Module Library

You need to compile the file containing the *function* and its associated *wrapper* using the `Module/Compile` command - this will produce a "`.so`" file to load into *Open GENIE*.

```
>>Module/Compile "myfunc.for" Symbols = "do_myfunc"
```

This command first runs a FORTRAN compiler on the code, then it produces a shared library from the object code. If the compilation fails for any reason you will see various error messages on the screen and the creation of the shared library will be aborted. On a successful run, you will see a message similar to the following:

```
** Module now compiled - load with: Module/Load "myfunc.so"
```

The `Symbols` parameter must be set to a comma separated list of *wrapper* subroutines that you wish to call from *Open GENIE* (via `Module/Execute`).

4. Load the module into *Open GENIE*

To load the module, use `Module/Load` command with the name of the "`.so`" file created by the `Module/Compile`. You can also supply a comment that will be displayed when you later type `Module/List`.

```
>>Module/List "myfunc.so" "My function module!"
```

You only need to reload a module if you change the source code and have executed another `Module/Compile` command.

5. Package function arguments into a workspace

You now need to create a workspace to hold the parameters for the `Subroutine` you wish to call in the module. The fields in the workspace must have the same names as those you specified in the second argument of your "`module_get...`" calls above. In our case, we only have one variable and we called it "*XVALS*", so assuming X is the array we wish to send:

```
>>X=Dimensions(10) # create a new array
>>Fill X 1.0 1.0 # Fill array with numbers 1.0 to 10.0
```

```
>>Pars=Fields() #create an empty workspace for module arguments
>>Pars.Xvals = X # add X to workspace as field “Xvals”
```

6. Execute the module and get the result

This is achieved by:

```
>>MY_RESULT = Module:Execute(“my_func”, Pars)
```

where “*my_func*” is the name of the FORTRAN wrapper routine you wrote, and *Pars* is the packaged arguments workspace created above. The values specified in “`module_put..`” calls will appear in the returned workspace *MY_RESULT*; in the above case *MY_RESULT* will contain one entry called *YVALS*, so we can type:

```
>>Printn MY_RESULT.YVALS
```

and see squared X values!

A Real-Life Sample Module

The following *module* was created by Spencer Howells at ISIS - the code is available in the “*ex2*” subdirectory of the examples area as *g2s.gcl* and *g2s.for*.

First the GCL procedure that calls the module:

```
# Procedures for G2S
#
PROCEDURE g2s
PARAMETERS
wg = workspace
RESULT ws
LOCAL g2s res
ws = wg
g2s=fields() ; g2s.lptin=wg.ntc
g2s.Xin=wg.x ; g2s.Yin=wg.y ; g2s.Ein=wg.e
g2s.qmax=inquire("g2s> qmax ")
g2s.npt =inquire("g2s> number of points ")
g2s.ic =inquire("g2s> window function code ")
g2s.rho =inquire("g2s> number density ")
module/load "g2s.so" ;
res=module:execute("g2s", g2s)
ws.ntc=res.lptout
ws.x=res.Xout ; ws.y=res.Yout ; ws.e=res.Eout
```

```
ws.xlabel=res.xcaptout ; ws.ylabel=res.ycaptout
ENDPROCEDURE
```

Now the FORTRAN program that performs the actual calculation:

```
SUBROUTINE G2S(g2s_get, g2s_put)
EXTERNAL g2s_get, g2s_put
INTEGER mn, lptin, lptout
PARAMETER (mn=33000)
REAL*4 Xin(mn),Yin(mn),Ein(mn),Xout(mn),Yout(mn),Eout(mn)
REAL*4 xnew(mn),yw(mn)
CHARACTER*40 xcaptout, ycaptout
CHARACTER*10 char
LOGICAL LMOD
DANGLE=38.1*1.112/150.0/150.0
PI=4.0*ATAN(1.0D0)

call module_get_int(g2s_get, 'lptin', lptin)
call module_get_real_array(g2s_get, 'Xin', Xin, lptin)
call module_get_real_array(g2s_get, 'Yin', Yin, lptin)
call module_get_real_array(g2s_get, 'Ein', Ein, lptin)
call module_get_real(g2s_get, 'qmax', QMAX)
call module_get_int(g2s_get, 'npt', npt)
call module_get_int(g2s_get, 'ic', ichar)
call module_get_real(g2s_get, 'rho', RHO)
if(lptin.eq.0)then
  call module_error(" G2S", 1 "ERROR ** No input data", " ")
endif

if(npt.eq.0)then
call module_error("G2S>",1"ERROR No of output points zero"," ")
RETURN
endif

lptout=npt
if(lptout.GT.mn)then
call module_error(" G2S>", 1 "#points reduced from 33000", " ")
lptout=mn
endif
delq=QMAX/lptout

if(rho.lt.1e-10)then
call module_error(" G2S>", 1 "ERROR ** rho is zero", " ")
RETURN
endif
```

```
      if(ichar.eq.0)then
      LMOD=.false.
      else
      if(ichar.eq.1)then
      LMOD=.true.
      else
      LMOD=.false.
      endif
      endif
      if(LMOD)then
      A=PI/xin(lptin)
      do nn=1,lptin
      yw(nn)=SIN(xin(nn)*A)/xin(nn)/A
      end do
      else
      do nn=1,lptin
      yw(nn)=1.0
      end do
      endif
C
C FORM VECTOR OF EQUALLY-SPACED R'S AT WHICH THE FOURIER TRANSFORM C
IS TO BE COMPUTED (RMAX IN ANGSTROMS), AND THE NUMBER OF R-POINTS.
C
      DO NR=1,lptout
      xout(NR)=delq*NR
      end do
C
C THE NUMBER OF POINTS IN THE RANGE OF DATA TO BE TRANSFORMED.
C
      xd=(xin(2)-xin(1))/2. !half x-channel
      do n=1,lptin
C xnew(n)=xin(n) +xd !offset - mid channel
      xnew(n)=xin(n)
      yw(n)=yw(n)*(yin(n)-1.)*xnew(n)
      end do
C
C COMPUTE FOURIER TRANSFORM OF THE DATA
C
      delr=(xin(lptin)-xin(1))/(lptin-1)
      AFACT=delr*2.0/PI
      DO 35 NR=1,lptout
      FS=0.0
      RP=xout(NR)
      DO N=2,lptin
      SINUS1=SIN(xnew(N-1)*RP)
      SINUS=SIN(xnew(N)*RP)
```

```
      FS=FS+ (SINUS*yw(N)+ 1 SINUS1*yw(N-1))/2.0
      end do
      yout(NR)=FS*AFACT
35    CONTINUE
C

      pir=PI*PI*2.*RHO
      do n=1,lptout
      yout(n)=yout(n)*pir/xout(n) +1.
      end do

      xcaptout=" Q (Angstrom-1)" ycaptout= " Structure Factor S of Q"

      call module_put_int(g2s_put, "lptout", LPTOUT)
      call module_put_real_array(g2s_put, "Xout", Xout, lptout)
      call module_put_real_array(g2s_put, "Yout", Yout, npt)
      call module_put_real_array(g2s_put, "Eout", Eout, npt)
      call module_put_string(g2s_put, "ycaptout", ycaptout)
      call module_put_string(g2s_put, "xcaptout", xcaptout)
      call module_information(" G2S> output is in point mode")

      RETURN
      END
```

The C Module Interface

The C module interface is used in almost exactly the same way as the FORTRAN interface, and provides similar routines - in fact most have the same name but with a prefix of *cmodule_* rather than *module_*. Invoking from the GCL command line is similar, except that an extra qualifier of `/C` must be specified for *COMPILE* and *EXECUTE* operations (it could also be specified on a load for consistency, but is not strictly necessary). For example,

```
>>MODULE/COMPILE/C "myprog.c"
>>MODULE/LOAD "myprog.so"
>>VALS=MODULE:EXECUTE:C("myfunc", PARS)
```

The major difference between the two is in the arguments - the C module routines can allocate memory for returned array structures using `malloc()`, unlike in FORTRAN77 where a pre-allocated array must always be passed. To indicate that you wish a routine to allocate the memory for you, set the array pointer to NULL before passing the address of it to the routine. For example:

```
GenieWorkspace *pars_get, *pars_put; /* passed from GENIE */
const char* name = "twotheta"; /* parameter to get */
fort_real* val_array =NULL; /*tell routine to malloc() memory*/
fort_int len;
/* then get array the array*/
cmodule_get_real_array(pars_get, name, &val_array, &len);

/* do some manipulations and return new values */
cmodule_put_real_array(pars_put, name, val_array, len);
free(val_array); /* release storage */
```

If a pointer is not NULL, the routine will assume it points to valid storage of size "*len*" - on return, "*len*" will be set to the number of elements of the array actually set, or 0 on an error. A C module must contain the following header files

```
#include <genie_cmodule.h>
#include <genie_cmodule_ver.h>
```

to set up the correct definitions and typedefs.

The `MODULE/COMPILE/C` *Open GENIE* command will insert the correct path for picking up these header files, but they are contained in the `$GENIE_DIR/library` directory if you wish to peruse them

The C module function must be declared like:

```
void my_cmodule(GenieWorkspace* pars_get, GenieWorkspace*
pars_put);
{
if (!cmodule_version_ok(CMODULE_MAJOR_VERSION,
CMODULE_MINOR_VERSION))
{
cmodule_print("Library version mismatch in MY_C_MODULE",
CMODULE_PRINT_ERROR);
return;
}
/* all OK, proceed */
}
```

Variables are obtained and set by calling the following functions, with either *pars_get* or *pars_put* and the variable name: (taken from `Genie_Dir/library/genie_cmodule.h`)

```
/* Get REAL parameter NAME from GENIE as VAL */
void cmodule_get_real(GenieWorkspace* pars_get, const char*
name, fort_real* val);

/* Return REAL variable VAL to GENIE as NAME */
void cmodule_put_real(GenieWorkspace* pars_put, const char*
name, fort_real val);

/* Return DOUBLE variable VAL to GENIE as NAME */
void cmodule_put_double(GenieWorkspace* pars_put, const char*
name, fort_double val);

/* Get STRING parameter NAME from GENIE as VAL
This routine ALWAYS allocates space for val with malloc(),
and you will need to free() at a later stage*/
void cmodule_get_string(GenieWorkspace* pars_get, const char*
name, char** val);

/* Return STRING variable VAL to GENIE as NAME */
void cmodule_put_string(GenieWorkspace* pars_put, const char*
name, const char* val);

/* Return STRING array VAL of LEN elements to GENIE as NAME */
void cmodule_put_string_array(GenieWorkspace* pars_put, const
char* name,const char* val[], fort_int len);

/* Get INTEGER parameter NAME from GENIE as VAL */
void cmodule_get_int(GenieWorkspace* pars_get, const char*
name, fort_int* val);

/* Return INTEGER variable VAL to GENIE as NAME */
void cmodule_put_int(GenieWorkspace* pars_put, const char*
name, fort_int val);

/* Get REAL array parameter NAME from GENIE as VAL
On Input:
The routine will malloc() space for the array if (*val ==
NULL),otherwise it will assume *val points to an array of size
len
On Output:
On error, *len will be set to 0
Otherwise, *len will be set to the number of elements read into
*val

*/
void cmodule_get_real_array(GenieWorkspace* pars_get, const
char* name, fort_real** val, fort_int* len);
```

```
/* Comments as for cmodule_get_real_array() */
void cmodule_get_double_array(GenieWorkspace* pars_get, const
char* name, fort_double** val, fort_int* len);

/* Return a REAL array VAL of length LEN to GENIE as NAME */
void cmodule_put_real_array(GenieWorkspace* pars_put, const
char* name, const fort_real* val, fort_int len);

/* Return a DOUBLE array VAL of length LEN to GENIE as NAME */
void cmodule_put_double_array(GenieWorkspace* pars_put, const
char* name, const fort_double* val, fort_int len);

/*Return an INTEGER array VAL of length LEN to GENIE as NAME */
void cmodule_put_int_array(GenieWorkspace* pars_put, const
char* name, const fort_int* val, fort_int len);

/* Print a message s to an output stream governed by option */
void cmodule_print(const char* s, fort_int option);

/* possible values for "option" in cmodule_print() are */
#define CMODULE_PRINT_NORMAL 0 /* Like GCL PRINTN */
#define CMODULE_PRINT_INFORMATION 1 /* Like GCL PRINTIN */
#define CMODULE_PRINT_ERROR 2 /* Like GCL PRINTEN */

/* Return a multi-dimensionsal array to GENIE; the array val
has ndims dimensions and these are stored sequentially in the
array dims_array[]*/
fort_int cmodule_put_nd_real_array(GenieWorkspace* pars_put,
const char* name,const fort_real* val, const fort_int*
dims_array, fort_int ndims);

/* return MAJOR and MINOR version of genie module interface */
fort_int cmodule_get_version(fort_int* major, fort_int* minor);

/* Check versions MAJOR and MINOR are compatable with the
current genie module interface; return 1 if all OK, 0 if an
error */
fort_int cmodule_version_ok(fort_int major, fort_int minor);
```

An example of a program to illustrate calling a module from C can be found in the directory `$GENIEDIR/examples/modules/c_example.c`. While an example of a GCL procedure to call the C module can be found at `GENIEDIR/examples/modules/c_example.gcl`

Example *Open GENIE* programs

The basic commands of *Open GENIE* plus multiline commands, for example `IF - ELSE - ENDIF` statements can be combined into *gcl* procedures. Below is a list of examples, these are arranged, so that the programs can be easily run by *Open GENIE* test routines, or by someone trying to get an idea of what *Open GENIE* can do. Most programs have their own directory, with the data required taken from files stored in this area[1].

If program fails to link, change into the example directory and type

```
>make links
```

this will usually find the appropriate data files and set up the appropriate "data" directory link.

It is best that these programs are load and run from the appropriate directory. For example, using the the file *dopamine.gcl* in the directory `/usr/local/genie/examples/simple_graphics`. Firstly change to this directory, and then load the program using

```
>>Load "dopamine.gcl"
```

and run these programs using the name of the file, in this case

```
>>dopamine
```

Before running most of the programs you will have to execute a `device/open` command.

- **Analysis**

focus_example.gcl - this example puts the necessary defaults into a workspace, that is required for focusing data from an IRIS raw data file.

---

[1] For large data files a link is made in the program directory to an external data area to avoid keeping large files in the source tree, however the access remains the same.

integrate_example.gcl - integrates a single spectra and a set of single spectra

rebin_example.gcl - rebins the data to the same range, but different number of time channels, then integrates the spectra. The procedure then rebins to a given X range, to a set of given X values, and to linear steps in a given range and then integrated. The data is then rebinned using logarithmic steps and integrated.

- **Contour**

contour_test.gcl - draws a filled colour contour plot, and then overplots two sets of contour lines,

- **IRIS**

add.gcl - procedure to analyse IRIS diffraction data. Developed by P. Marshall

- **Data**

This directory holds the data used for some of the procedures in the examples section.

- **ex1**

dssdemo.gcl - this demonstrates the use of different plot styles, with error bars and line plots of linear and logarithmic scales.

- **io**

contains various examples of programs concerned with the input and output of data from genie.

ascii_file.gcl - demonstrates the use of *Open GENIE* routines to read/write ASCII data using the ASCIIFILE command.

ascii_io.gcl - demonstrates the use of *Open GENIE* routines to output and input data in a fixed easy reading ASCII file format using the `Get` and `Put` command

intermediate_file.gcl - gives an example of the use of the *Open GENIE* routines to manipulate intermediate files.

raw_file.gcl - example of the use of *Open GENIE* routines to read and manipulate spectra from ISIS raw files

- **modules**

This directory gives examples of wrapping routines - *c_example.gcl* and *fortran_example.gcl* - for the calling of a module from within *Open GENIE* written in *gcl*, and examples of a such modules for C and FORTRAN- *c_example.c* and *fortran_example.f*

- **newget**

In this section examples are given of C, C++, and FORTRAN files, which can be used like the `get()` function of *Open GENIE.*

- **peak**

pktest.gcl - a procedure which generates data from a quadratic equation and then fits a cubic equation to the data, with the linear part of the function inputted by the user and then held constant. The data, the fit and the residuals are then plotted.

- **prisma**

diffuse.gcl, *phon.gcl* - examples of publication graphics by Mark Harris. Loads ASCII data from a file creating a graph and plotting observed and theoretical data as markers and lines respectively.

- **report**

report.gcl - this procedure demonstrates the ability to read data using an external FORTRAN program, and the use of *CONTOUR* and *CELL_ARRAY* commands in making a colour contour plot on data taken on MARI.

- **simple_graphics**

boxes.gcl - this procedure draws boxes on the plot window with corners chosen by the operator.

dopamine.gcl - this procedure reads in a *.PRO* file. It then plots two ranges of observed data as points with the calculated data plotted as a line. The difference between the calculated and data plot below. The data file OPT17.PRO was obtained from a Rietveld fit of a structural model of deuterated dopamine hydrobromide to HRPD data.

funcplot.gcl - this plots a function of your own specification by writing a module "*on the fly*" and then using this module to generate the data points

test_mutitplot.gcl - a procedure which produces a multi-plot from a spectra, and then Examples of multiplots constructs a multiplot from a 2-D array

textcirc.gcl - draws the string "A simple text string" in a circle.

- **slides**

slides.gcl - this procedure is a slide demonstration of *Open GENIE's* main features.

- **two_d_graphics**

contour_test.gcl - displays a contour plot of a set of neutron inelastic spectra taken on IRIS.

glad.gcl - draws a colour intensity map of S(Q, ω) data taken on MARI, which also allows you to see slices of the intensity variation with constant energy, and momentum.

Further Developments and Advanced programming

A *Tk Tcl* Interface is currently under development at ISIS, and the idea is to provide a rudimentary interface and also a set of useful utilities e.g. a tcl "display" command that will, as well as drawing a plot, allow you to interactively change e.g. line colour by clicking on a line. However, the above should allow anybody who wishes the ability to knock up a GUI for a GCL command procedure they write.

Open GENIE is a much more detailed and powerful system than is obvious from the rest of this manual. There are many things which can be done using *Open GENIE*, for example

write procedures in an object oriented command syntax

extend the data class system to advanced data types

code procedures directly in *smalltalk*

or extend *Open GENIE* by writing in C++

The End of Part I

At the end of a session all variables created during it are lost when

```
>>Exit
GENIE exiting
$
```

is typed.

However, all the variables from one *Open GENIE* session can be saved into a binary image file. This is useful when a number of *gcl* procedures have been loaded, and will mean you don't have to type `Load "procedure.gcl"` next time. This is achieved using the `save()`

```
>>Save "my_saved_session"
```

The session can be re-started by the command

```
genie "" "my_saved_session"
```

The `""` are needed because the first argument can be used to load up a *gcl* file on start-up, instead of `$HOME/genieinit.gcl`. Thus you are able to define commands for different start-ups, for example

VMS: `marigenie :== opengenie sys$login:mariinit.gcl`

UNIX C shell: `alias marigenie "genie $HOME/mariinit.gcl"`

Part B - Installation Guide

Installing *Open GENIE*

Open GENIE is available as a binary distribution for the common machine types and in a source code distribution for anything else. If the binary distribution is available for your machine and operating system, this is the easiest way to install *Open GENIE*. The binary distribution kits are marked with the name of the machine, the operating system and the manufacturer of the machine range if known. For example

```
alpha-dec-osf3.2
```

means that the version of *Open GENIE* in the binary distribution was made for a machine with an alpha/AXP processor, in a machine manufactured by Digital Equipment Corporation and running Version 3.2 of the *OSF/1* operating system, which is now called *Digital UNIX*. In general all these should match with your system for a binary kit to work. If you are running a version of UNIX, but are unsure of these details you can download the file "*config.guess*" directly from the ISIS website at *http://www.isis.rl.ac.uk/GenieUsermanual/config.guess* website, and run it on your system with the command

```
>sh config.guess
```

This script will attempt to guess which version of UNIX you are running and return a string similar to the one above. If it matches one of the GENIE binary distributions this will be the appropriate version to use for your machine.

Installation instructions for VMS and UNIX machines are different so there are different sections describing how to install each one. If your machine does not have a binary version of the kit available, you may be able to build one from the source code, to do this you will need to read the section on *Source code installation*. Note for example instructions to *mount* the CD-ROM see appendix E.

Binary Installation

Choose the binary installation instructions below which match the type of hardware on which you are installing Open GENIE.

a) *UNIX Installation*

All UNIX kits are supplied as compressed tar files - made by the tar program on the appropriate machine. To decompress these files you will need to have *gunzip* installed. NB *gunzip* is available from the web site as a tar file if you do not have this, for building on your system.

To install Open GENIE follow the instructions given here for a Digital Alpha machine, but substituting the appropriate names and version numbers for your distribution kit of Open GENIE

```
gunzip genie-1.1-alpha-dec-osf3.2.tar.gz

tar xvf genie-1.1-alpha-dec-osf3.2.tar

cd genie-1.1-alpha-dec-osf3.2
```

To install GENIE in the default system location with data files in `/usr/local/genie`, executable files in `/usr/local/bin`, and library links in `/usr/local/lib` type:

```
./configure

make install
```

You will need to do the `make install` from the *superuser* or *root* account. If you wish to see what `make install` will do without actually performing the install, type

```
make -n install
```

Note: the above install can only be executed by the superuser and does a system wide installation. If you do not have access to the superuser account, you may change the `./configure` command to give different default installation directories for yourself, for example

```
./configure --prefix=$HOME
```

will change the default installation directories, putting any binary files into your personal account's *$HOME/bin* directory, the rest of genie into a *$HOME/genie* directory, and library symbolic links in *$HOME/lib*. Note: to run *Open GENIE* automatically your *bin* directory will need to be in your *PATH*.

Once *Open GENIE* is installed it can be invoked by running the *genie* script in the *bin* directory. If the *bin* directory is specified in your shell *PATH* environment variable, you should be able to just type:

```
genie
```

If you are using a C shell then before the command will work properly you need to type:

```
rehash
```

b) *VMS Installation*

Below are the steps needed to get *Open GENIE* running on a Dec alpha/AXP machine running VMS6.2 or later.

All VMS/alpha kits are supplied as compressed zip files, made by the public domain *zip* program. To decompress you will need to have *unzip* installed. If unzip is not installed by default on your system then to set up the command type the following line:

```
$unzip :== $ disk:[directory]unzip_axp.exe
```

where *`disk`*`:[`*`directory`*`]` is the location of the `unzip_axp.exe` file (this is available from the *Open GENIE* web pages).

Uncompress the *Open GENIE* kit using the command below (you will need approximately 40000 blocks of disk space to successfully uncompress and install *Open GENIE*):

```
$unzip genie-1_0-alpha-dec-vms6_2.zip
```

This creates a directory called `disk:[directory.OPENGENIE_1_1]`. To run *Open GENIE* you will need to set up the definition

```
$opengenie :== @disk:[directory.opengenie_1_1]genie_setup
```

either your own “*LOGIN.COM*” or the system wide “*SYLOGIN.COM*”. This enables *Open GENIE* to be run using the command `opengenie`. For installing *Open GENIE* on a system which is shared between several users you may want to install the shared library and executable files as known shareable images. This is done by adding the lines

```
$INSTALL ADD disk:[directory:OPENGENIE_1_1.GENIE]genie.so
/OPEN/HEADER/SHARED/RESIDENT

$INSTALL ADD disk:[directory:OPENGENIE_1_1.GENIE]GMAIN
/OPEN/HEADER/SHARED/RESIDENT
```

to the *SYSTARTUP_VMS.COM* file in *SYS$STARTUP*.

c) *WindowsNT & WINDOWS 95 installation*

Windows/NT and Windows-95 are currently not supported, but will be made available in subsequent releases.

Source Code Installation

Unless you are fully prepared to compile a fairly complex software system written in C/C++ we recommend that you check for a binary installation of *Open GENIE* and install that instead.

Source code installation is normally more complicated than the Binary installation, but it is useful if:

a) you need to install *Open GENIE* on a machine/operating system combination for which we have not yet provided a binary installation.

b) you wish to develop or modify the *Open GENIE* code in some way.

It is also very likely that you will need to have some familiarity with the C, C++ and FORTRAN compilers on your system. Generally we would recommend that you install GNU g++ and use either *g77* or the native FORTRAN compiler for building *Open GENIE*.

Before embarking on a source code build, it may well be worth mailing us at *genie@isise.rl.ac.uk* to check that we are not planning to do a similar build or that someone else has not already done one.

The source code is distributed in a compressed tar file which you will need to unwind in a suitable development area. The *PGPLOT* and *Splash* libraries are distributed as separate distributions which will need to be unpacked in *PGLOT* and *Splash* subdirectories respectively. We have to do this because *Open GENIE* is being distributed under the GNU licence, and the other packages though freely available are not. Following this the trees must be patched to update them with modifications and files added for *Open GENIE*. The commands needed will be similar to the ones shown below.

```
cat genie-1.1.tar.gz | gunzip | tar xf -
cd genie-1.1
cat ../pgplot5.1.1.tar.gz | gunzip | tar xf -
cd splash
cat ../../splash1.7.tar.Z | gunzip | tar xf -
```

The *pgplot* tree should now be in the "*pgplot*" subdirectory and *splash* in the "*splash*" subdirectory. The patches can be applied with the commands

```
cd genie-1.1/pgplot
patch -p2 -s < pgplot-5.1.1.patch
cd ../splash
patch -p2 -s < splash-1.7.patch
```

These patches should apply fully without asking questions, if you get a question such as "`File to patch:`", check that *pgplot* and *splash* were extracted correctly, so that the paths `genie-1.1/pgplot/makemake` and `genie-1.1/splash/Makefile.in` both point to valid files.

Ideally, the code will now build automatically if you type.

```
cd genie-1.1
./configure
make all
```

this builds the development version of *Open GENIE* which can be run by typing `./parser` from within the `src` directory. To go the whole way and build an install kit for your system you will need to do a

```
make kit
```

to build a compressed tar file for a binary installation. This can then be installed to a users directory or system wide as described in the section on *Binary Installation*. Note that to build a kit you will also need a copy of GNU `autoconf` installed (see `ftp://ftp.nd.rl.ac.uk/pub/packages/gnu`).

If you are struggling email *genie@isise.rl.ac.uk* and we will endeavour to help. When you do contact us for advice, please provide us with examples of the errors that occur in the mail message. Remember that we probably can only guess at what might be wrong as we are unlikely to have the same system that you have to test things on.

Please check out the operating system specific notes below, they could save you a lot of time!

Operating system specific notes

At the moment our primary development platform is Digital UNIX (OSF/1) on alpha AXP and this is generally the one which works best for building without a hitch. Linux works fine, but may need a bit of fiddling if you run a different distribution to us. IRIX (for version 5.3) works, but is likely not to work immediately for later versions. For VMS, only attempt this if you are strong willed and familiar with VMS/POSIX and have it installed on VMS 6.2 or later. VMS building on VAX is not recommended although we have done it!

We provide notes to help you with building on specific operating systems, these notes can also be found in the README.<os> files of the source code distribution.

Linux

This is not an exhaustive list but should help:

1. One version of g++ on the Linux we build on (1.2.13) does not find its own include files by default. Before building define the include path with

   ```
   export CPLUS_INCLUDE_PATH=/usr/include/g++-include
   ```

2. It may also be necessary to edit `src/Makefile` and remove the flag -extend_source from the `$(F77)` rule

3. After the first build attempt,

   ```
   $cd library
   $make -i
   ```

   Note g77 return status of 4 stops a correct build

4. cd src and touch lexcommands.c. To ensure that flex doesn’t overwrite this with unworking code.

5. go into pgplot and edit the makefile to remove -f2c from the begining of the "LIBS=" line. Re-make to build the pgxwin_server properly

6. Now try “`make all`” again, it should run through this time.

IRIX

First run `./configure`, currently this picks up f2c if it is installed which is probably not what is wanted. Now edit “*./config.cache*” and change the line

```
f2c_c_cos=${ac_cv_lib_f2c_c_cos= “yes”}
```

to read

```
f2c_c_cos=${ac_cv_lib_f2c_c_cos= “no”}
```

and re-run `configure`.

VMS

These are the instructions for compiling under VMS POSIX, tested on Open VMS (AXP) 6.2 with POSIX 2.0.

Before entering VMS POSIX:

```
DEFINE/JOB POSIX$INCLUDE SYS$SCRATCH
```

After entering POSIX:

```
sh vms_posix.setup
```

will building the kit using a fairly large number of rather VMS/POSIX specific routines and produce a similar file to the UNIX versions.

In general it is possible that if the `configure` doesn’t work then it may be the result of picking the wrong FORTRAN compiler. To change this you can edit the file “*config.cache*” and then re-run the `configure` command.

Appendices

Appendix A

Features of the command line

- Pressing <*TAB*> on a partially completed command will complete the command if it is unambiguous, otherwise pressing <*TAB*> a second time will list possible options.
- Pressing <*TAB*> also works for partially completed file/directory paths
- Pressing *<Ctrl>H* on a partially complete command will give a one line summary of all commands matching that pattern
- Pressing *<Ctrl>K* will start a "command keyword" help search - type in a one word query, and all commands whose description matches this will be printed
- Pressing *<Ctrl>R* will start an incremental reverse search of previously typed commands
- Pressing *<Ctrl>L* will clear the screen and redraw the current line at the top

Additional features

- Pressing *<Ctrl>*_ will undo a last edit/change you made to the command line
- Pressing *<Ctrl>X<Ctrl>U* will also undo the last edit/change
- Pressing *<Ctrl>U* will discard the rest of the line starting from the current cursor position
- Pressing *<Ctrl>A* will move the cursor to the beginning of line
- Pressing *<Ctrl>E* will move to end of the current line

Appendix B

Supported Data File Formats

Open GENIE supports access to the file formats listed below:

- ISIS Raw File (Read only)
- GENIE-II Intermediate file (Read only)
- *Open GENIE* Intermediate File (Read and Write, Machine independent)
- *Open GENIE* ASCII File (Read and Write)
- HDF Files (Write only)

These files can all be read directly with the `Get()` and `Put()` commands by specifying numbered or named data elements. *Open GENIE* supports access to any format of ASCII file via the `Asciifile()` command.

Appendix C

Supported Graphics Devices

Below is a list of graphics devices currently supported by *Open GENIE*, these rely upon the PGPLOT package and it may well be possible to use other pgplot drivers with *Open GENIE*. In the distributed version of *Open GENIE* we have only checked out the functioning of the devices below on all platforms. To see other devices which may be available on a particular version of *Open GENIE* type

```
>> Device/Open “Help”
```

Device Name	**Description String**
Tcl/Tk driver	“TK” or “tk”
X-Windows devices	“XWINDOW” or “xw”
Postscript (portrait)	“PS” or “ps”
Colour Postscript (portrait)	“CPS” or “cps”
Postscript (landscape)	“PS” or “ps”
Colour Postscript (landscape)	“VCPS” or “vcps”

Appendix D

Supported Graphics Attributes

Fonts

Open GENIE uses the PGPLOT fonts when drawing fonts onto the open graphics device, currently, the supported fonts available are:

```
$NORMAL
$ROMAN
$ITALIC
$SCRIPT
```

Note: For accessing special scientific/Greek characters within these fonts, the documented PGPLOT escape sequences may be used in the *Open GENIE* text string being plotted. For example "*Time-of-Flight (\gms)*" where "*\gm*" produces a Greek μ character. For further details see the PGPLOT documentation, available at `ftp://astro.caltech.edu/pub/pgplot`.

Markers

Markers types available for plotting.

```
$POINT
$PLUS
$STAR
$CIRCLE
$CROSS
$BOX
```

Linestyles

Linestyles available for plotting.

```
$FULL
$DASH
$DOT_DASH
$DOT
```

Colours

Pre-defined colours available for the graphics by default. For using further colours in the graphics, see the *Open Genie reference manual*.

```
$BLACK
$WHITE
$RED
$GREEN
$BLUE
$CYAN
$MAGENTA
$YELLOW
$ORANGE
$LIGHT_GREEN
$SEA_GREEN
$LIGHT_BLUE
$PURPLE
$CRIMSON
$DARK_GRAY
$LIGHT_GRAY
```

Appendix E

Commands required for installing *Open GENIE* from a CD ROM

Please note that these commands vary depending on the version of the operating system you are using.

Installing on UNIX with Digital UNIX

Initially find the CD

```
>file /dev/rrz*c
```

and look for an RRD device e.g. `RRD42` on `/dev/rrz4c` then mount the CD

```
>mount -t cdfs -o ro,rrip /dev/rz4c /mnt
```

then get the kit

```
>cp /mnt/osf/genie-1.1.31F
```

Note: you will need to have the CDFS kernel option enabled.

Installing on VMS

Type in, for example

```
$mount DKA400:GENIE11
$copy sys$scratch: /log
```