GDA User Guide

Release 9.11

Diamond Light Source

Dec 21, 2018

Contents:

CHAPTER

ONE

INTRODUCTION TO THE GDA

This guide is intended to give an overview of the GDA framework and explain the functionality of the Jython scripting environment embedded within it. This includes how to write scripts, perform data collection by scans and plot data from scans. It does not cover the beamline- or technique- specific parts of the GDA.

This guide explains: How to use the GDA. Where and how your data will be stored. How to perform experiments using basic commands such as scan. Guidelines on writing Jython scripts and Jython modules. Coding and documentation standards. Advanced scripting tools. How to write your own Scannables - the objects which are controlled by scans.

Note: Some of this chapter is specific to gda use at Diamond.

1.1 The GDA Project

The Generic Data Acquisition (GDA) framework is a Java-based distributed system used on Diamond's beamlines to control experiments and collect data. The project to develop the software was initiated at SRS, Daresbury until release 4.2 in October 2003. From then it was co- developed by teams from both facilities until release 7.0.0 in late 2007. Since then Diamond's GDA has been purely developed by the Data Acquisition and Scientific Computing Group.

Each beamline uses a copy of the GDA software configured for its own particular hardware and experimental requirements. Experiments can be controlled using beamline- or technique-specific GUI panels, a command-line terminal or by writing and running scripts (macros).

1.2 Logging onto a beamline workstation

The GDA software is run from workstations on the beamline. Beamlines use one of two methods to log into these workstations:

1. Shared user account login. On some beamlines you will login to the same shared user account. The name of this account is the name of the beamline followed by 'user' all in lower case. For example, if your beamline were I01 the account would be 'i01user'. The password to login using this type of account will be given to you by the beamline staff. Note however that the workstations on beamlines that use this method will be automatically logged in to this account when started or rebooted.
2. FedID account login. On other beamlines you will login using your own Facilities user FedID name and password issued to you by the User Office.

As of March 1st, 2007 all beamlines use the first, shared account method, but this will change. It is sometimes possible to control the beamline from the outside by logging into one of these workstations remotely.

Regardless of how you login on the beamline the GDA system will store your data (and some other files) in a shared location on your beamline's storage server. Read access control to this data will be determined by beamline staff. Let them know if you are concerned about the security of your data. This storage server is described in the following section.

1.3 Starting the GDA

Before starting the GDA software you may have to set up the shell environment. If you login with your own FedID account type:

```
source /dls/ixx/etc/ixx-profile.sh
```

Starting the GDA is a two step process. The bulk of the GDA software is started from a terminal by typing:

```
gdaservers
```

from a Linux terminal or:

```
gdaservers.bat
```

from a Windows terminal. This starts up the GDA server software on one of the beamline's central control server machines. This software may be running before you start using the system, but asking it to start again causes no harm. A GUI client is used to interact with this server. About 30s after starting the server you should see the message "Server Initialisation Complete". Then type:

```
gdaclient
```

from a Linux terminal, or:

```
gdaclient.bat
```

from a Windows terminal to start the GUI client. The GUI client will ask you for a name and password when it starts up. This name is used by the GDA to determine where to put your data. If you have logged in to your own user account using a FedID, select the option to automatically use the FedID of that account. Otherwise if you are logged into a shared user account, give it your FedID and password allocated by the User Office.

If you try to start the GDA client and get a string of connection errors followed by a final "Can't find a Command Server" error, either the server is not running and needs to be started or you didn't wait long enough before starting the client. You may close the GDA client software running on your workstation without effecting the operation of the command server. If you need to leave your computer, rather than locking the screen to prevent access to the GDA command server, you may close the client software and log off. You may then login and start up the client server again and begin where you left off.

1.4 The GDA Client

The GDA client acts as an interface to the command server. The client has a GUI with a number of panels accessible via tabs. The available panels vary by beamline. Depending on the nature of your experiment and beamline you may be able to use experiment-specific panels to control the beamline. Other useful panels include:

- A configuration panel that has a number of pictures showing each piece of controllable hardware (or optical elements) on the beamline. These pictures also provide one mechanism for moving these elements.
- A scripting terminal panel that provides a versatile way to control these elements and perform experiments.
- A Jython editor panel that provides a way to write scripts that can be saved and then run using a button in this panel.

Methods to control the beamline using these panels are described in the rest of this manual.

1.5 Basic Control

The hardware that can be controlled by the GDA on your beamline is displayed on the configuration panel. Whether you intend to use this panel or not, this section provides an introduction to controlling the hardware on a beamline. In the configuration panel, clicking on each of the available Optical Elements (OEs) will show an interactive picture of that element.

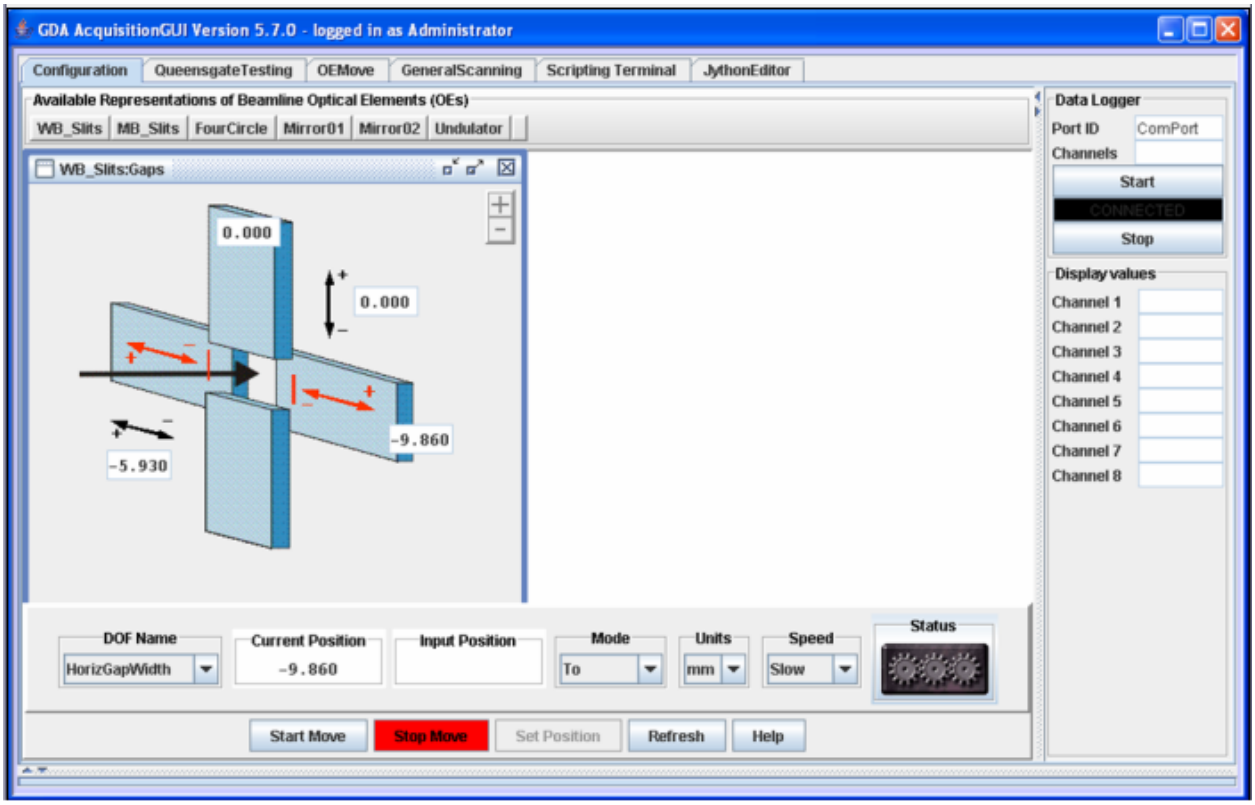


Fig. 1.1: OE configuration panel

The hardware that can be controlled by the GDA on your beamline is displayed on the configuration panel. Whether you intend to use this panel or not, this section provides an introduction to controlling the hardware on a beamline. In the configuration panel, clicking on each of the available Optical Elements (OEs) will show an interactive picture of that element.

1.6 Data Storage

All experimental results will be saved initially on a beamline's central storage area. You will not normally have much choice in where your experimental data is stored. The GDA software saves the results from each experiment or scan in a standard place. This rigidity allows the data to be migrated away from the beamline and to be backed up, archived, and made accessible via projects such eScience and iGrid.

1.6.1 Storage directory path

Each beamline has a storage server with upwards of 5Tbytes of storage. When you log into your user account a directory from this storage server is automatically mounted. The name of this directory and path to access it depends upon your beamline's name. Every beamline has a three digit, lower-case identifier such as i01 or b01. Using i01 as an example, the directory on a Linux workstation would be:

```
i01-storage.diamond.ac.uk:/dls/i01, mounted as /dls/i01
```

On Windows workstation the directory would be:

```
//i01-storage.diamond.ac.uk/i01, mapped to X:
```

Note: If you need to access the identifier for your beamline, it is stored within your account space as an environment variable called BEAMLINE.

1.6.2 Storage directory structure

There is one common directory on a beamline's storage server. Data from all users and experiments is stored here. It contains the following sub-directories:

```
storagedir
|_ data          The GDA stores data here
|_ scripts       GDA Scripts used to control experiments
|_ logs          The GDA stores log files here
|_ spool         Beamline systems store temporary files here
```

The data directory contains the data for all the experiments performed recently on the beamline. Each experiment will be identified by the year it was performed in, the proposal number assigned by the User Office, and a visit number. The visit number will normally be 1 unless an experiment takes more than one visit to Diamond to complete. For each experiment, the GDA software will create a sub-directory within the data directory according to this pattern:

```
/dls/<beamline>/data/<year>/<sciencearea><proposal>-<visit>
```

For example, data on beamline i18 (spectroscopy), for proposal identifier 1243 on the first visit would be stored in: /dls/i18/data/2007/sp1234-1 under Linux X:data2007sp1243-1 under Windows.

1.6.3 Data Files

Most experiments are composed of a number of scans. A typical scan involves rotating a sample through a number of angles taking a measurement or an image at each. Each successive scan you perform is assigned a sequential number called the scan number. The results from performing a scan will be stored in the current experiment directory with the name:

```
<beamline>-<scan number>.<extension>
```

where the extension depends on the type of scan. Such a file is typically a set of tabular data describing the points scanned and the value of certain measured parameters.

Some scans, such as those where images are taken, will result in more than one data file. In this case each file resulting from that scan is also given a sequence number resulting in a set of files named:

```
<beamline>-<scan number>-<sequence number>.<extension>
```

This system is due to be replaced by a NeXus system where each scan results in only one file (see below). Also, this naming scheme may not be adhered to on some of the MX beamlines where analysis software that requires different naming schemes is sometimes used.

1.6.4 Example storage directory

The following shows an example storage directory structure on the imaginary beamline i01.

```
storage
|_ scripts
|_ logs
|_ spool
|_ data
|
|_ 2007
|  |
|  |_ expa-1           Proposal called exp1, first visit
|  |  |_i01-1.nxs      data from 1st scan
|  |  |_i01-2.nxs      data from 2nd scan
|  |  |_i01-3.nxs      data from 3rd scan
|  |
|  |_ expb-1           Proposal called expb, first visit
|     |_i01-1.nxs      tabular data from 1st scan
|     |_i01-1-1.jpg    1st CCD image
|     |_i01-1-2.jpg    2nd CCD image
|     |_i01-1-3.jpg    3rd CCD image
|     |_i01-2.nxs      tabular data from 2nd scan
|     |_i01-2-1.jpg    1st CCD image
|     |_i01-2-2.jpg    2nd CCD image
|     |_i01-2-3.jpg    3rd CCD image
|
|_ 2008
   |
   |_ expa-2           Expa's second visit
      |_i01-1.nxs      data from 1st scan
      |_i01-2.nxs      data from 2nd scan
      |_i01-3.nxs      data from 3rd scan
```

1.6.5 Transition to NeXus data files

The system for storing data files described above is being replaced by one based on the NeXus format which is being developed as an international standard for representing results from neutron, x-ray and muon sources. With this method each scan will always produce only one file containing all the data.

This new method will be phased in during 2008, and is currently available for testing. In order to turn on NeXus file writing then just set the java property:

```
gda.data.scan.datawriter.dataFormat=NexusDataWriter
```

During the testing phase this will also produce a legacy SRS type format file.

CHAPTER

TWO

THE GDA COMMAND LINE

2.1 Background

The scripting engine of the GDA is an extended version of Jython (Python implemented in Java). It is now common across all current beamlines at Diamond. It is used extensively on measurement beamlines such as I02-I04 and I22 to provide the underlying command infrastructure to support the Graphics User Interfaces (GUIs). It has proven very popular particularly on the experimental beamlines such as I06, I15, I16 and I18 where it provides command-line control of experiments to perform scans, creates bespoke objects to operate within those scans and enables general automation of experimental procedures.

The great advantage of Jython is that its syntax and functionality very closely resembles that of the widely used and popular Python language with the additional advantage that most Java libraries are automatically available. The language and concepts have proven easy to learn by non-specialised programmers and there has been a very significant uptake from Diamond beamline staff to do customisation of beamline operations.

2.2 Basic Concepts

The scripting language in the GDA is Jython, which is Python implemented in Java. Python was developed to have a simple and quick to learn syntax while still remaining a fully-featured programming language. Some basic concepts used in the GDA Jython environment which are listed in more detail in the later sections of this guide:

1. In the GDA Client, Jython commands may be typed in the JythonTerminal panel which provides a terminal-like prompt, or scripts (macros) may be typed in, saved and run from the JythonEditor panel. It is useful to note that the same commands can be used in both panels, and that they both use the same namespace in Jython i.e. a variable defined in the Jython command-line will be accessible from within scripts and vice-versa. User scripts must be stored in /dls/iXX/scripts (where XX is the beamline number).

2. There is a core set of commands for operating the beamlines. These are known as the GDA 'extended syntax' as to use these commands you do not have to use 'proper' Jython syntax. To save typing when using these commands you may omit brackets and commas. For example, instead of typing:

```
>>> pos(myMotor,10)
```

   you only have to type:

```
>>> pos myMotor 10
```

   (Although the first version is still a valid command to type). This is convenient for the most common commands. It is possible to dynamically extend the syntax in this way with your own commands using the 'alias' command.

3. Data is normally collected in beamlines using scans. Most scans are stepped scans i.e. move motors, collect data from detectors. move motors, collect data from detectors...etc. So to do this, scans contain two types of objects: detectors and 'Scannables'. Scannable objects conceptually act like motors in the sense that they have a position

and you can move them, but they can represent anything from low-level hardware to a complex calculation. Scans may be nested with no limit to the number of dimensions.

4. The data collected by scans is plotted live to the GDA Client. Scan data (from old data files as well) is held inside data objects which can be plotted or be used in mathematical operations.

2.3 Code Completion

Code completion is available on the command line. This can be triggered using either *Ctrl-Space* (similar to eclipse) or *tab* (similar to bash/zsh etc).

If there is only one option, it will automatically inserted

```
>>> conti<TAB>
>>> continue
```

If there are multiple choices, they will appear in a popup above the cursor. Typing will continue to filter the options and *tab* or *enter* will insert the selected one.

Note: a *tab* will still indent code if the cursor is at the beginning of the line

```
>>> |print 'helloWorld'
>>>     |print 'helloWorld'
```

2.4 Example usage of the main Jython GDA commands

List all scannable objects:

```
>>> pos
```

Help:

```
>>> help
```

List all devices:

```
>>> ls
```

List all scannable devices (devices that implement the Scannable interface):

```
>>> ls Scannable
```

Import demo scannable definitions:

```
>>> import scannableClasses
>>> from scannableClasses import *
```

Make a new instance of SimpleScannable:

```
>>> simple = SimpleScannable('simple', 0.0)
```

Scan *simple* from 0 to 1 in steps of 0.01:

```
>>> scan simple 0.0 1.0 0.01
```

Get current position of *simple*:

>>>pos simple

Move *simple* to 0.5:

>>>pos simple 0.5

Delete an existing object:

```
>>> del simple
```

See the Jython training manual for more detailed descriptions and further examples.

2.4.1 Example devices

A Jython module containing several demonstration scannable objects is contained in the user scripts folder ('documentation/users/scripts/scannableClasses.py'. This file can be opened, viewed and edited in the Jython Editor view in the GDA client. (If this view is not visible at startup, select the 'JythonEditor' view from the View menu in GDA.)

New users can gain familiarity with the Jython terminal by following the examples below. Users should type in the Jython commands below from the GDA Jython Scripting Terminal. Each command follows the Jython terminal prompt '>>>'. A short description precedes each command or set of commands.

To superimpose successive scans on previous scans, the 'Create new graph' and 'Clear old graphs' should be left unchecked.

Import all the classes from the demonstration 'scannableClasses' module (if not already done so above):

```
>>> import scannableClasses
>>> from scannableClasses import *
```

Help is available for most of these classes:

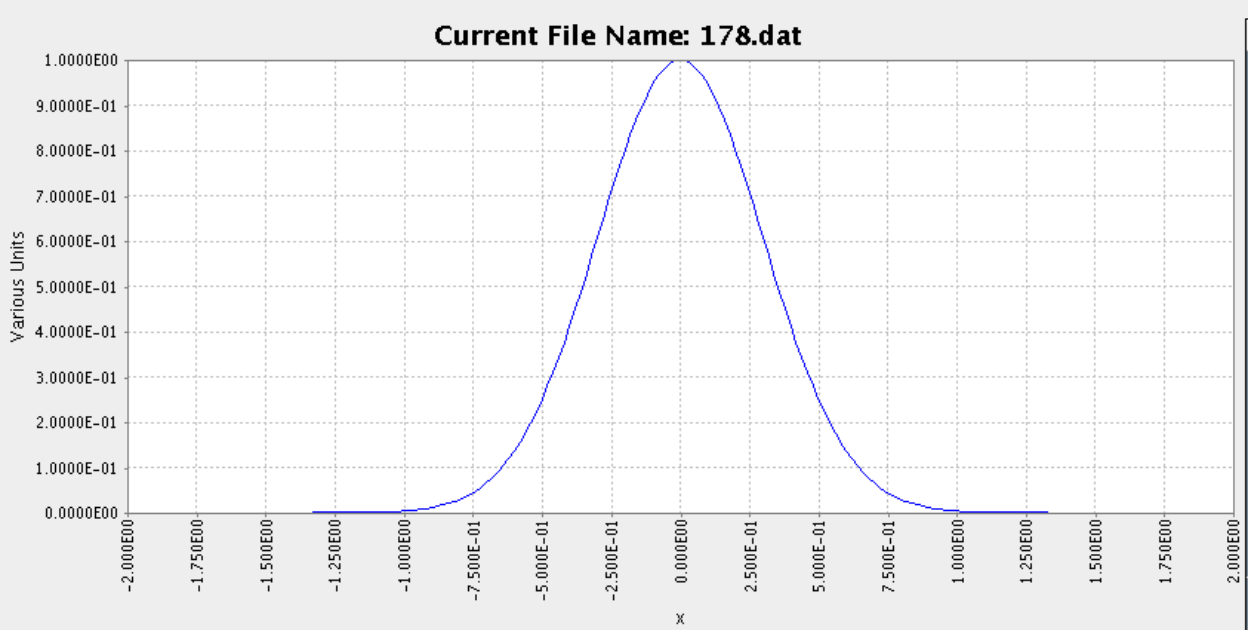
```
>>> help ScannableGaussian
>>> help ScannableSine
```

Make an instance of ScannableGaussian:

```
>>> sg = ScannableGaussian('sg', 0.0)
```

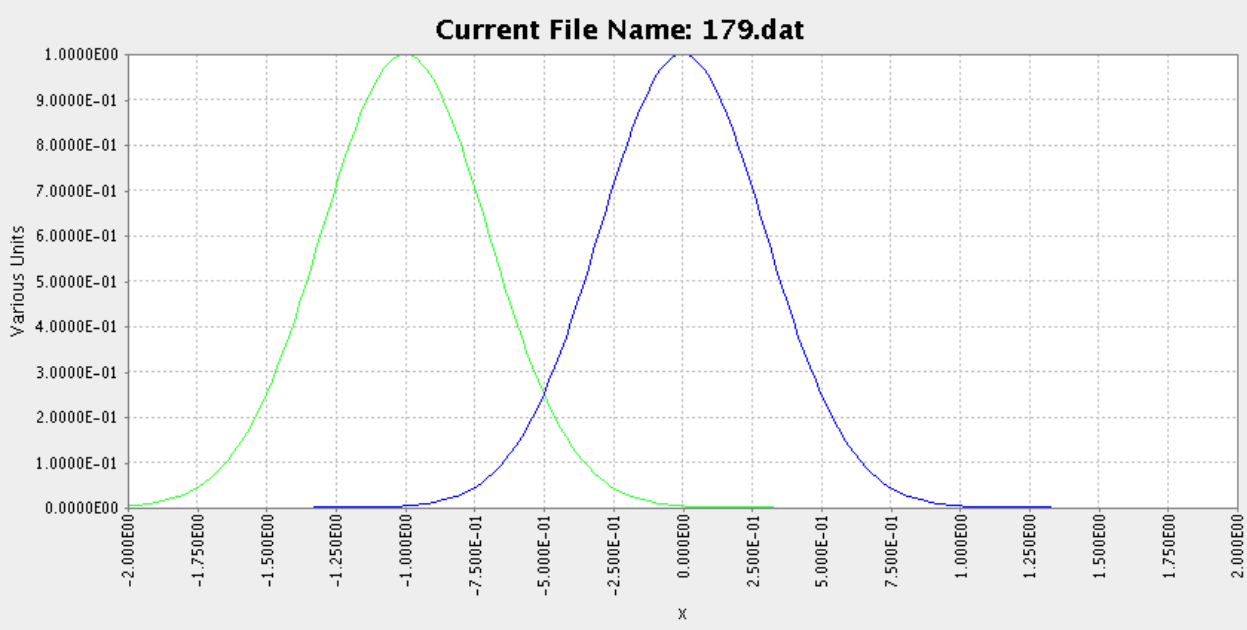
Scan it from -2 to 2 in steps of 0.02:

```
>>> scan sg -2.0 2.0 0.02
```



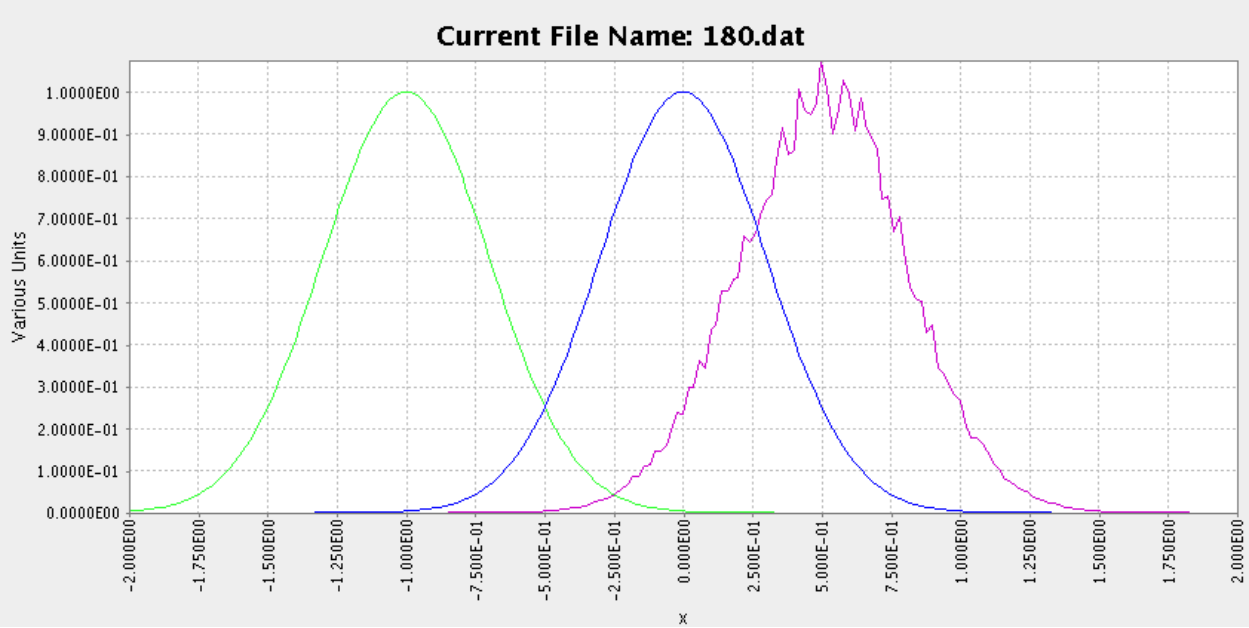
Change its centre to -1 and rescan:

```
>>> sg.centre = -1
>>> scan sg -2.0 2.0 0.02
```



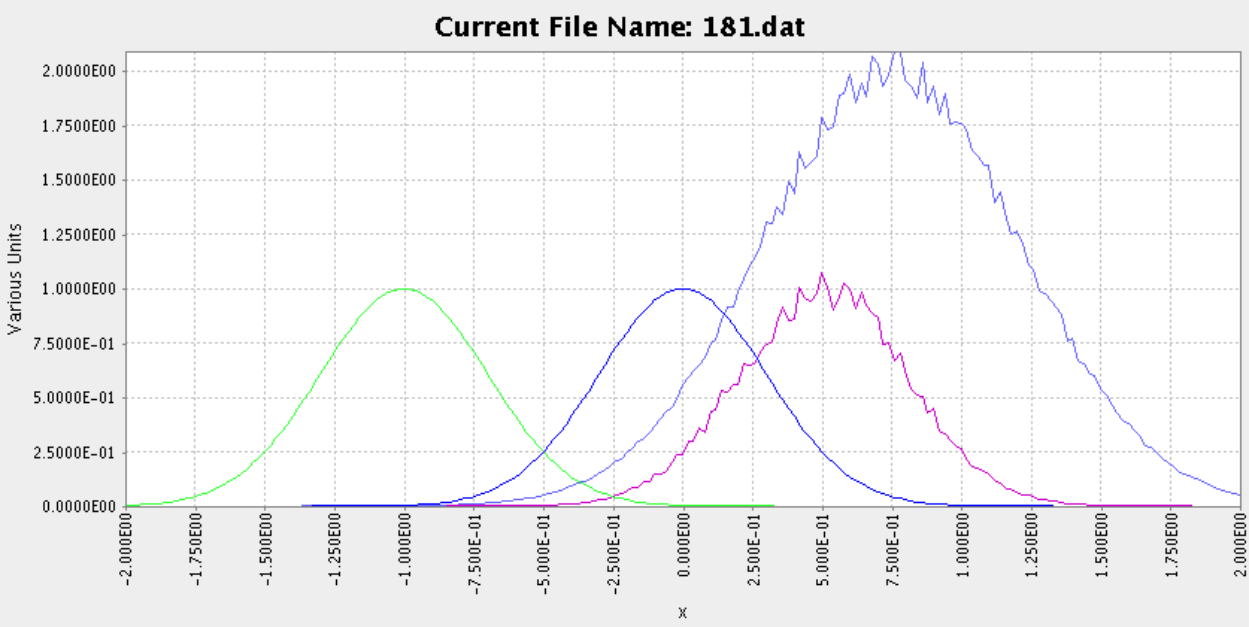
Move again, add some noise, and rescan:

```
>>> sg.centre = 0.5
>>> sg.noise = 0.2
>>> scan sg -2.0 2.0 0.02
```



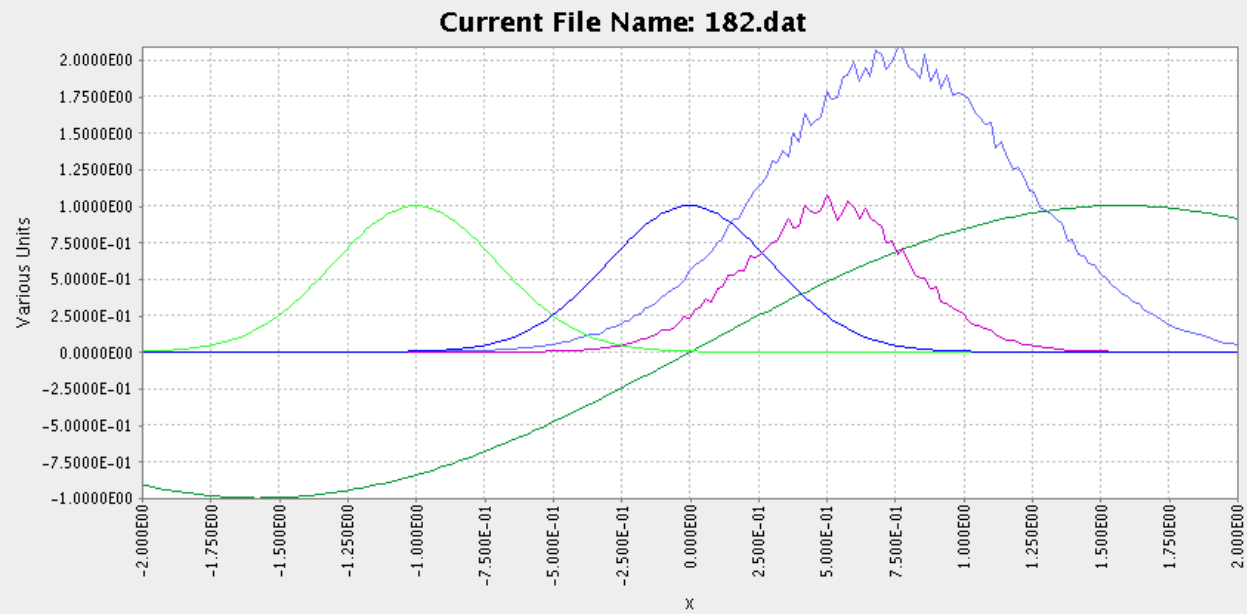
Make a new instance of ScannableGaussian, setting values for its additional optional properties, and scan it:

```
>>> sg2 = ScannableGaussian('sg2', 0.0, centre=0.75, width=1.54, height=2.0, noise=0.1)
>>> scan sg2 -2.0 2.0 0.02
```



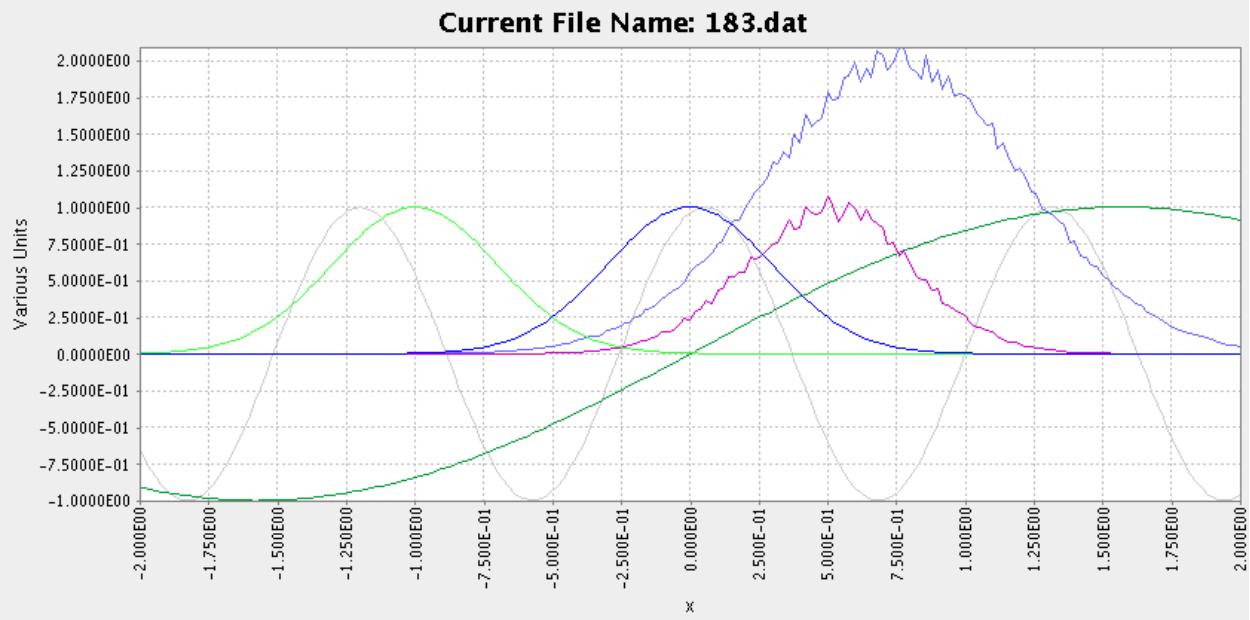
Make an instance of a ScannableSine class and scan it:

```
>>> ss = ScannableSine('ss', 0.0)
>>> scan ss -2.0 2.0 0.02
```



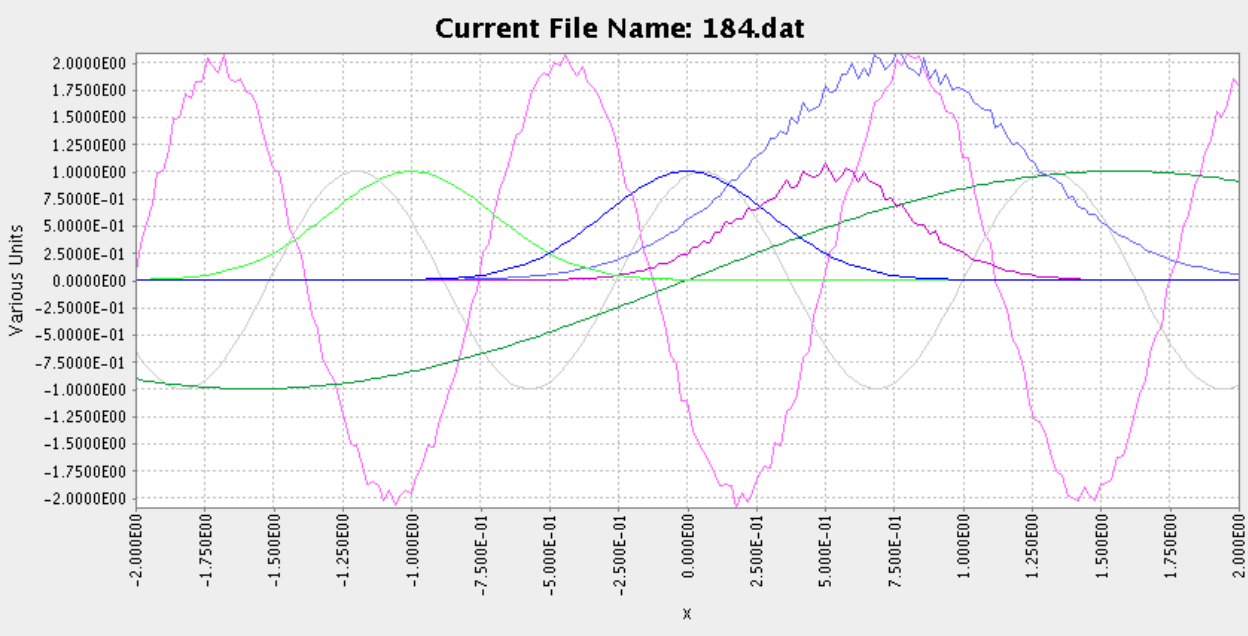
Change the period and phase of ss and rescan:

```
>>> ss.period = 0.2
>>> ss.phase = 1.0
>>> scan ss -2.0 2.0 0.02
```



Change the magnitude, phase, and noise, of the sine, and rescan:

```
>>> ss.magnitude = 2.0
>>> ss.phase = 0.5
>>> ss.noise = 0.2
>>> scan ss -2.0 2.0 0.02
```



Multiple scans can also be nested to an arbitrary level. To illustrate a nested scan with two levels, i.e. an inner scan nested within an outer scan, we can define the outer scan to set the value of the inner scan. The example class ScannableGaussianWidth in the scannableClasses module (in directory documention/users/scripts) takes an existing ScannableGaussian instance, and sets the width of the ScannableGaussian to its own current value. The enclosed ScannableGaussian can be scanned at each width across a user-defined range.

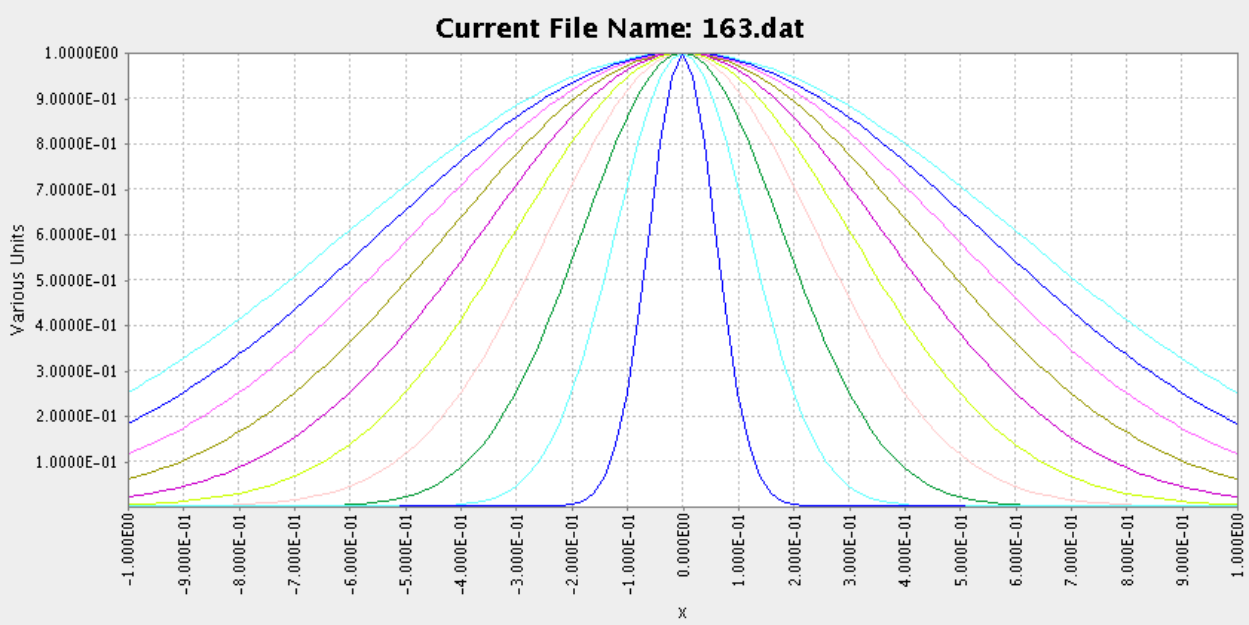
Instantiate a new inner ScannableGaussian and then a new ScannableGaussianWidth:

```
>>> sgi = ScannableGaussian('sgi', 0.0)
>>> sgw = ScannableGaussianWidth('sgw', sgi)
```

Perform the nested scan:

```
>>> scan sgw 0.2 2.0 0.2 sgi -1.0 1.0 0.02
```

The results of this scan in the Terminal plot window are shown below:



2.4.2 Using the plotting functions in GDA

Apart from the basic plotting window in the Terminal view which displays the current scan, GDA also has some advanced plotting capabilities for previously-recorded scans. These are designed for post-scan analysis and visualisation.

All the information about this kind of analysis is available in the User guide for the 'Diamond Scisoft Data Analysis Plugin'

2.5 Sources for help

- Beamline manuals which list how to use the specific panels and Jython objects used on that beamline.
- Useful Jython websites:
- The help command in the GDA Jython interpreter.

CHAPTER

THREE

EXTENDED SYNTAX COMMANDS

A list of the GDA 'Extended Syntax' commands which are available on all beamlines:

foo (*x*)
: Return a line of text input from the user.

list ()
: lists all the types of objects representing hardware on this beamline

list <interfacename>
: lists all the objects of the given type (interface) on this beamline

pos ()
: lists all Scannables and Detectors i.e. the objects which can be used in scans

pos <object>
: returns the position of the Scannable object

pos <object> <position>
: moves the Scannable to the given position

pos <object> <position> <object> <position>
: concurrent move of multiple Scannables

inc <object> <amount>
: relative move version of the pos command

inc <object> <amount> <object> <amount>
: concurrent relative move of multiple Scannables

help ()
: lists the extended syntax commands

help <object>
: gives a description of the object if available

run <script_name>
: runs the given script as if it were opened and run from the JythonEditor. Note that the script must be located in the /dls/iXX/scripts folder.

pause ()
: checks to see if one of the pause/resume or halt buttons in the JythonTerminal panel have been pressed. Use this in long scripts to have a convenient place to pause/resume/abort the script.

reset_namepsace ()
: restarts the GDA Jython environment without the need to restart the entire GDA software. This is useful if you have a problem with the namespace. Note: this will not rebuild connections to hardware.

alias <method_name>
: add the given function to the extended syntax so that you do not have to add ()'s to call the function. Useful for very commonly used methods on the beamline.

watch <object>
: opens a pop-up box in the JythonTerminal panel and shows a constantly refreshed value of the Scannable

history()
: list the history of commands typed into that terminal

!<command_string>
: repeat the latest command which starts the same as the given string.

record [on|off]
: starts/stops recording all terminal output to a file placed in the scripts directory

level <object>
: returns the level attribute for this Scannable. Levels are used to provide ordering when moving Scannables during scans

level <object> <value>
: changes the level value for this Scannable. 5 is the default for Scannable objects and 10 is default for Detectors.

list_defaults()
: lists the Scannables and detectors whose positions and outputs will be included in scans by default without including them when typing a scan command

add_default <object>
: adds a Scannable or detector to the list of defaults

remove_default <object>
: removes a Scannable or Detector from the list of defaults

The format of commands related to scanning are listed later. There are also more commands available from Scannable objects which are also listed later.

CHAPTER

FOUR

WRITING SCRIPTS

4.1 Basic scripting

Scripts can be entered and run directly from the script editor panel of the GDA. You may work on more than one script, and of course open and save them.

To run a script from the command line type:

```
>>> run "ScriptName"
```

4.1.1 locations

`/dls/iXX/scripts` scripts written by users for their experiment.

`/dls/iXX/software/gda/config/scripts` beamline-specific scripts written by Diamond staff (read-only).

`/dls/iXX/software/gda/scripts` core GDA scripts available to all beamline (read-only).

(where XX is the beamline number)

4.1.2 namespaces

The Jython command-line in the JythonTerminal panel of the GDA GUI uses the same namespace as any script, so variables created in one are accessible from the other.

4.2 coding Standards

The Python language has an industry accepted set of coding and documentation standards. In this section we present the standards that are of particular benefit to GDA script writers.

We discuss how to and how not to layout code and how to document your scripts (including documentation of classes and modules). The section concludes with recommended practise on naming variables, objects, functions, classes and modules.

4.2.1 Indentation

We recommend that you always use 4 spaces per indentation level. This will make cutting and pasting code far easier and this indentation is easy to read. Never mix tabs and spaces.

4.2.2 Long lines

Try to limit all lines to a maximum of 72 characters. You can wrap long lines using Python's implied line continuation inside parentheses or use a backslash. For example:

```
# wrapping lines with backslash
if width == 0 and height == 0 and \
colour == 'red' and emphasis == 'strong':
    f()

# wrapping lines with parentheses
if (width == 0 and height == 0 and
colour == 'red' and emphasis == 'strong'):
    f()
```

4.2.3 Documentation Standards and help

You should write comments in your code as much as possible to explain to any other user of the code (or yourself if you look at the code at some point in the future) what the code does and why. Don't overdo it either.

You should write docstrings for all public modules, functions, classes, and methods. DocStrings are not necessary on non-public methods.

Jython has a built-in mechanism for documentation. This is the __doc__ attribute. You write your comments in triple quotes (‘‘‘) in the very first line of every class or method definition. The contents of these quote are then accessible via the __doc__ attribute. In the GDA you can quickly access these comments via the help command:

```
def myMethod():
    '''This is my comment explaining what this method does.

    The Jython convention is to have a single line which is a
    concise summary of the object or method starting with a
     capital letter and ending with a period. It should have
    not mention the functions name. If you need more detail,
    leave a blank line and then one or more paragraphs separated
    by blank lines.
    '''
```

Access the comments in two ways:

```
>>> print myMethod.__doc__
>>> help myMethod
```

Help is available in this way for many built-in GDA objects and classes.

4.2.4 Naming Conventions

Recommended naming conventions are discussed for all the common entries in Jython

Package and module names

Modules should have short, all-lowercase names with underscores as separators: e.g::

```
my_module
```

Class names

Class names should start with a uppercase letter and subsequent words should be capitalised. This is known as CamelCase e.g.:

```
MyClass
```

Exception names

Exceptions should be classes, so use the class naming convention with the suffix "Exception": e.g.:

```
MyClassException
```

Global variable names

Global variables are meant for use inside one module only. Start with a lowercase letter and capitalise words: e.g.:

```
myGlobalVariable
```

Function names

Function names should start with a lowercase letter and subsequent words should be capitalised: e.g.:

```
myFunction
```

Always use 'self' for the first argument to instance methods of classes. Always use 'cls' for the first argument to class (static) methods.

4.3 Importing Modules

Jython import statements require discussion since there are several ways we can import modules. If we use the simplest form of import:

```
>>> import myModule
```

all code in the module is executed and then all the symbols defined in the module are added to the namespace myModule. To access any of the symbols in the namespace you must qualify the symbol with the namespace. For example, if the module defines a class called MyClass, then to instantiate an object of this class you must write:

```
>>> import myModule
>>> myObject = myModule.MyClass()
```

However if you use an alternate form of import:

```
>>> from myModule import *
```

all code in the module is executed and then all the symbols defined in the module are added to the global namespace, rather than the namespace myModule. This means we can now use unqualified names to access symbols defined in the module. Thus the above example can now be written:

```
>>> from myModule import *
>>> myObject = MyClass()
```

4.4 Interrupting Scripts

Although the GDA provides buttons to pause and stop your scripts, you will find that your script may not get paused or stopped immediately. Indeed it may take quite a long time for these actions to take effect. This is a consequence of how the underlying Java system works. If you understand the mechanism the GDA uses to pause and stop your scripts you will be able to write additional code that will make your script more responsive.

When you press either the pause or stop button in the GDA workbench, the GDA system sends a signal to your Jython script. Upon receipt of the signal the Java system will perform one of two possible actions. If your script is busy executing code, then Java sets an interrupt flag to indicate you should pause or stop. However the Jython system will not attempt to check this flag until you complete the current function call or loop statement and hence your script becomes unresponsive. But if your script is already paused (in a sleep state), then Java will interrupt your code immediately and stop the script.

Because you can never be sure when you will need to pause or stop the script you can call the function:

```
>>> pause()
```

or as it is aliased just:

```
>>> pause
```

to ask the GDA to check the interrupt flag at any time. If you make this call inside a function or loop you will ensure a prompt response. For example modify your loops to call pause periodically:

```
>>> for i in range(1,10000):
>>>      if (i % 100 = 0) pause() # pause every 100 iterations
>>>          # your_code ...
```

4.5 Matrices

The GDA is deployed with the Jama matrix package to perform matrix calculations. Jama is intended to be the standard matrix package for Java. It was co-authored by MathWorks who produce Matlab. As any Java class can be used within Jython without any special programming, Jama classes may be called directly. However a wrapper class has been written to enable Jama objects to interact with the Jython environment more easily.

To use this library, import the GDA Jama wrapper by typing:

```
>>> from Jama import Matrix as M
```

Then matrices can be created easily by typing command such as:

```
>>> a = M([[1,2,3], [4,5,6]])
```

These matrices can then easily be manipulated in Jython. They can interact with Jython lists, tuples and integers in ways you would expect of matrices. To illustrate:

```
>>> print a * 3
     [[3, 6, 9], [12, 15, 18]]
```

And:

```
>>> b = M([1 2 3])
>>> print b + [4, 5, 6]
    [5,7,9].
```

4.6 Advanced Scripting

This section will be more of interest to beamline staff writing scripts which are repeatedly used on the beamline by users.

4.6.1 Interacting with the user

You might write a script which requires user input every time it is run, such as asking for a variable. To do this within the GDA you can use Jython's built in raw_input function.

To use this command:

```
>>> raw_input('Prompt for input: ')
```

When this is called the script will be paused and the prompt in the JythonTerminal GUI panel changes to ask the user to input something there. After the user has pressed return whatever has been typed in is returned as a string by the requestInput command. For example:

```
>>> target = raw_input("Where would you like to move s1_upper to?")
>>> pos s1_upper float(target)
```

Note: When using `raw_input` from the RCP client, input can be entered on any connected client, not just the one that started the script or ran the command.

When using `raw_input` from a remote connection, input will be restricted to that connection.

4.6.2 Error handling in scripts

If you have a script which will be repeatedly used for a task, rather than a simple one-off script to perform an experiment, you may wish to add error handling to your script so that if anything goes wrong (such as a hardware failure) while running the script an appropriate message is displayed and action taken to cleanly stop equipment or resolve the problem.

The Jython language has an error handling mechanism similar to many other languages. Protected code is contained within try/except blocks which 'wrap' the enclosed code and pickup exceptions if they occur within them and run code to resolve or report the problem.

One thing to note is that all Jython errors inherit the Jython class `org.python.core.exceptions`. This does not inherit from the Java exception base class `java.lang.Exception`. This is important to remember when writing error handling for scripts. When scripts are halted or stopped by user intervention, it is a Java exception which is raised; however many other errors which may occur in scripts will be Jython errors. Scripts error handling should be able to catch both forms of exception.

For example:

```
>>> try:
...     # your logic goes here
...     # have lots of calls to gda.jython.commands.ScannableControl.pause()
...     # to allow users to pause/halt the script cleanly
... except InterruptedException, e:
...     print "a user-requested halt!"
...     # code here to stop the equipment and return the beamline to a safe state
... except java.lang.Exception, e:
...     print "a Java exception must have occurred!"
```

```
...     # code here to stop the equipment and return the beamline to a safe state
... except:
...     print "a Jython error must have occurred!"
...     # code here to stop the equipment and return the beamline to a safe state
...
```

Alternatively, Jython has a finally clause which is always operated after a try block whether an exception has been thrown or not. This may be more suitable in some circumstances.

```
>>> try:
...     # your logic goes here
...     # have lots of calls to gda.jython.commands.ScannableControl.pause()
...     #to allow users to pause/halt the script cleanly
... except InterruptedException, e:
...     print "a user-requested halt!"
... except java.lang.Exception, e:
...     print "a Java exception must have occurred!"
... except:
...     print "a Jython error must have occurred!"
... finally:
...     # code here to stop the equipment and return the beamline to a safe state
...     # this is always called whether an exception was raised or not
...
```

4.6.3 Persistence

Persistence is when you wish to store variables so that they can be saved a retrieved after software restarts. The GDA provides an easy to use mechanism for this. Values are stored in xml files stored in the /dls/iXX/software/gda/var directory. These xml files can be created, read and saved from the Jython environment.

To use this system you must create a XMLConfiguration object (which represents the XML file). You can then easily read and store values to this file using a name to identify each piece of information. You must ensure that you call the save method to make sure the XML file is saved after every change. Here is example code on how to use this:

```
>>> from uk.ac.diamond.daq.persistence.jythonshelf import LocalParameters
>>> config = LocalParameters.getXMLConfiguration("my_parameters_file")
>>> config.setProperty("mythings.myint", 42)
>>> config.setProperty("mythings.mystring", "blarghh")
>>> aJavaListOfStrings = ['one', 'two', 'three']
>>> config.setProperty("otherthings.mylist", aJavaListOfStrings)
>>>
>>> config.save() # Make sure to save them!
>>>
>>> Integer myint = config.getInt("mythings.myint")
>>> String mystring = config.getString("mythings.myint")
>>> String[] stringArray = config.getStringArray("otherthings.mylist")
>>> List stringList = config.getList("otherthings.mylist")
```

4.6.4 Logging

When running long scripts it can often be useful to log what is happening as it goes along. Jython has a built in logging library and this can be used to send messages to the central GDA logs as well as to the console. This can be preferable to simply printing to the terminal as the source can be tracked and it can be disabled without editing scripts.

To use logging in your scripts import logging and create a logger.:

```
>>> import logging
>>> logger = logging.getLogger('script_name')
>>>
>>> # To use the logger
>>> logger.info('This message will be printed to the console and the logs')
script_name: This message will be printed to the console and the logs
```

If your script uses *error handling*, the logger from above can extract stacktraces to help debug the causes. This is done using the `exc_info` keyword to include the current exception:

```
>>> try:
...     # with stage_x being a faulty motor
...     stage_x.getPosition()
... except DeviceException as de:
...     logger.error('Failed to get motor position', exc_info=True)
...
script_name: Failed to get motor position - Error getting position
  gda.device.MotorException: <message of underlying motor error>
```

The full stack trace is written to the logs. Only the underlying causes are displayed to the user.

4.6.5 Writing to external files

Writing to text files can be a useful way of recording the state of GDA and the beamline during a running script. Jython has built in methods of opening, writing to and closing files that prevent you having to manually handle any errors that may occur.

To open a file for writing:

```
>>> with open('/path/to/file/to/open', 'a') as output:
...     output.write('text to write to file\n') # \n adds a new line
```

This replaces any `try:...except:...finally:...` blocks needed when writing to files. The file can be used at any time until the end of the `with` block where it will be closed, even if an error occurred somewhere in the block.

Note: Using this will add text to the end of a file. To overwrite the current contents of the file, pass 'w' instead of 'a' in the open command.

CHAPTER

FIVE

SCANNING

This section discusses scanning which is the main method of collecting data in the GDA.

5.1 The definition of a scan

In the GDA there are two types of scan available: step-based scanning and continuous scanning. Continuous, or fast, scanning is when detectors are collecting data throughout the time that other hardware such as motors are operated. Such scanning can be very convenient because the scan can take less time. However continuous scans offer less flexibility to users as wiring is required between specific motors involved in the scan and the detector electronics.

Steps scans require no such specific wiring. They have a distinct separation between movement of hardware and then detector collection and readout. Such a scan has two lists of objects inside it: Detectors and Scannables. At each node in a step scan, the Scannables are operated (these operations are grouped by the level attribute which each Scannable has. So from the lowest level upwards all Scannables at a particular level are operated concurrently and once they have finished their movement the next level is operated and so on.). Once the Scannables have been operated then data is collected from all Detectors concurrently and once they have all completed then the data from that round of collection plus the new locations of all Scannables are recorded to file.

Scannables are a generic type of object representing anything which is not a detector which needs to be operated between each round of collecting data. They could be as simple as a motor, but could be a calculation based on several motor positions, or a Scannable could even represent a script which is run at every point in the scan.

Scannables can be defined as representing a number, or array of numbers. Their operations analogous to motors i.e. they have methods to 'move' them, a method to requested their status to see if they are moving and a method to get their current position. They are the main type of object that users will interact with in the GDA

5.1.1 The structure of a step scan

To explain in detail how a scan works, below is listed the structure of a step scan. An explanation of the parts of the scan is explained in the rest of this section. This structure should be helpful for those writing their own Scannable classes and need to understand scans in detail.

Prepare for the scan:

1. call the atScanStart method on all Scannables which are part of the scan
2. if the flag has been set to return all Scannables to their original positions after the scan, record all Scannables current positions
3. create a new data file

Iterate through all dimensions of the scan, at each level:

1. call the atScanLineStart method on all Scannables

2. move the Scannable of that dimension to its next step
3. if at the inner-most dimension, loop through all points in that scan. At each point:
   (a) call the atPointStart on every Scannable in the scan
   (b) move the Scannable of that dimension and every other Scannable that is not a separate dimension in the scan (if the parameters from the command-line said that the Scannable should be operated). All the Scannables to be operated at this point are operated in groups defined by their level attribute. Once all Scannables in a level have finished moving (this is known by polling their isBusy method), then the Scannables from the next level are operated
   (c) collect data from all Detectors which are part of the scan
   (d) record the data and Scannable positions in the data file. The data is also broadcast to the GDA client for visualisation when required. This is the only point in the loop that the getPosition method of Scannables is called
   (e) call the atPointEnd on every Scannable in the scan
4. call the atScanLineEnd method on every Scannable in the scan

Finish the scan:

1. call the atScanEnd method on every Scannable in the scan
2. close the data file
3. if the flag has been set to return all Scannables to their original positions after the scan, then move all Scannables back to position noted at the start

5.2 Scan commands

The scan commands have the format:

```
scan <scannable> <start> <stop> <step> [<scannable> [<start>] [<stop>] [<step>]] ...
     ... [<detector1>] [<detector2>]
```

Only the first Scannable with its start, stop and step values is required. Other Scannables are optional and they be operated during the scan or merely have their position recorded at each step in the scan. If subsequent Scannable objects have three arguments after their name then this counts as an extra dimension to the scan. There is no limit to the number of dimensions which can be included in a scan.

The detectors are also optional in the command: in the GDA Jython environment there is a list of default Scannables and Detectors which will be included in every scan without having to include the object in the scan command.

To explain the various possible options this format gives, here are some more explicit explanations followed by examples:

5.2.1 Other types of step scans

There are other step scans available which work in a similar manner but take alternative input parameters. These scans and their commands are:

TO BE RE-ANOUNCED!

5.3 Scan options

A scan command is comprised of a list of scannable to be moved or observed and a list of detectors to be triggered. At each point in the scan, after the scannable and detectors have completed their tasks, the positions of the scannable and the data from the detectors is read and recorded. By default unless scannables are given different priorities they will move concurrently (see section below on priorities). There are a number of different types of scan each called with a different syntax:

- **One parameter scan**:

```
>>> scan scannable start stop step [detector [exposure time]]
```

  The scannable is swept from the start value to the stop value in step size steps. At each point, if specified, the detector is triggered. An optional exposure time can be given for a detector.

- **Concurrently moved parameters scan**:

```
>>> scan scannable start stop step scannable2 start step (detector)
```

  A second Scannable is varied concurrently with the first. This second device does not need a stop value, as the number of points visited is determined by the values for the first device.

- **Multidimensional scan**:

```
>>> scan scannable start stop step scannable2 start stop step (detector)
```

  This example is a two dimensional scan in which an entire scan over scannable2 is performed after each move of the first Scannable.

- **Move-to-keep-still scan**:

```
>>> scan scannable start stop step scannable2 start (detector)
```

  In this example scannable2 is moved to the start position and then this position is maintained as scannable is scanned. An example below shows why this may be useful.

  At each point in the scan along with the data from the detector, a line is stored in the data file recording the value of every Scannable listed in the command. This means Scannables can be used like detectors to record values. For example.

- **Monitoring a Scannable**:

```
>>> scan scannable start stop step scannable2 scannable3 (detector)
```

  For each point in the scan the value of the 2nd and 3rd Scannables (along with the first of course) are recorded.

There is no limit to the number of Scannables or detectors which can be included in any scan. The syntax of these commands can be mixed to build up complex scans.

5.4 Example scan commands

Following are some useful example scans used on beamlines at Diamond.

Energy scan. The scan command might be used to sweep the energy, or wavelength, of photons illuminating a sample. For example:

```
>>> scan pgmenergy 500 2000 0.1 uv 2
```

This scans the x-ray energy from 500eV to 2000eV in 0.1eV step. At each scan point, an image is taken from a camera called uv. The camera exposure time is set to 2 second. This command is the same:

```
>>> scan pgmenergy 500 2000 0.1 ca43s 0.5
```

except at each point a drain current is measured by reading from a scalar card channel ca43s. The scaler card counting time is set to 0.5 second. (These commands are from I06 and require pgmenergy, uv and ca43s to be defined on your beamline.)

Time scan. The scan command may be used to perform a scan with respect to time:

```
>>> scan x 1 100 1 ct4 1 detector
```

This scans over a dummy Scannable x that acts simply as a counter. For each step of x, a timer scannable ct4 waits for one second and then the detector is triggered. This is a useful scan for tracking the stability of a measurement or to see what happens if another device is moved. (This is from I16. To use this command the Scannable for x and ct4 must be defined in your beamline.)

Move-to-keep-still scan. The scan command may sweep some degrees of freedom, while keeping a second constant, even while the second is influenced by the first. For example:

```
>>> scan en 7.0 7.1 0.001 hkl 100 detector
```

This sweeps a scannable en that controls the energy or wavelength selected by a monochromator. As the wavelength is varied, the angle that a scattered ray of interest leaving the sample varies. To detect this ray the sample must be rotated as the wavelength varies to keep the ray directed at a fixed detector. The scannable hkl 100 orients the sample so that the particular ray described by the Miller indices (1,0,0) is detected. As this orientation is a function of wavelength the hkl scannable will cause the sample to rotate appropriately as the wavelength is swept.

5.5 Default devices and detectors

There may be some Scannables or detectors which you might want to operate in every scan. To avoid having to type the object's name in every scan, there is a list of "defaults" which are included in every scan:

list_defaults returns the list of default objects operated in every scan

add_default <object name> add the given object to the list of defaults

remove_default <object name> remove the given object to the list of defaults.

You may find that some detectors are added to the list of defaults automatically when the GDA is started.

5.6 Configuring Scannable movement priority (levels)

Normally Scannables are moved between points concurrently; that is they are all started moving at the same time and once (one by one) they have all finished moving the data is read and recorded. However it is also possible to explicitly define numerical priorities to Scannables. Devices are moved to the next scan point in sequence with the highest priority (lowest number) devices moving first. Starting with highest priority, all devices with that priority are moved concurrently. Once these have completed their move all the devices at the next highest priority are moved and so on. Once this sequence is complete and all the devices have reached final positions the values of all the detectors and positions of all the devices are sampled and recorded.

The default level for a Scannable is 5. This can be read or modified using the level command.

To return the level of the Scannable (higher priority = lower number):

```
>>> level <Scannable>
```

and to set the level of the Scannable to a given value:

```
>>> level <Scannable> <integer>
```

For example to move x and y where y's actual position depends on the position of x, the priority of x should be set higher than that of y. y will now not be moved until x has completed its move; if y reads the value of x to use in a calculation it will use the new value of x.

As another example a wait device might simply wait n seconds if asked to move to n. To use this to add a settling delay after all devices have moved assign it a low priority. Once the rest of the devices have moved the wait device will then 'move' for n seconds before the positions of all devices are read and the detectors triggered.

CHAPTER

SIX

WRITING NEW SCANNABLES

The definition of what a Scannable is has been designed to be as flexible as possible. This is so that Scannables can represent as wide a variety of concepts as possible - from pieces of hardware through to complex calculations.

Simply put, a Scannable represents a number or an array of numbers and acts in a manner analogous to a motor. In others words you can retrieve the numbers the Scannable holds using the getPosition() method, and you can have the Scannable perform some operation to changes those numbers by using one of the move methods. You can also ask the Scannables status to see if it ready to accept a call to one of the move methods by using the isBusy() method.

Ways to interact with a scannable include:

myScannable()
: returns a pretty print version of the Scannable

pos myScannable
: returns a pretty print version of the Scannable

myScannable()
: returns the current position of the scannable

pos myScannable 10
: moves the Scannable

myScannable(*10*)
: alternative syntax to move the Scannable

pos myScannable 10 myOtherScannable 20
: moves two scannables at the same time

inc myScannable 1
: relative move of the Scannable

myScannable.**isBusy**()
: returns True (1) if the Scannable is moving

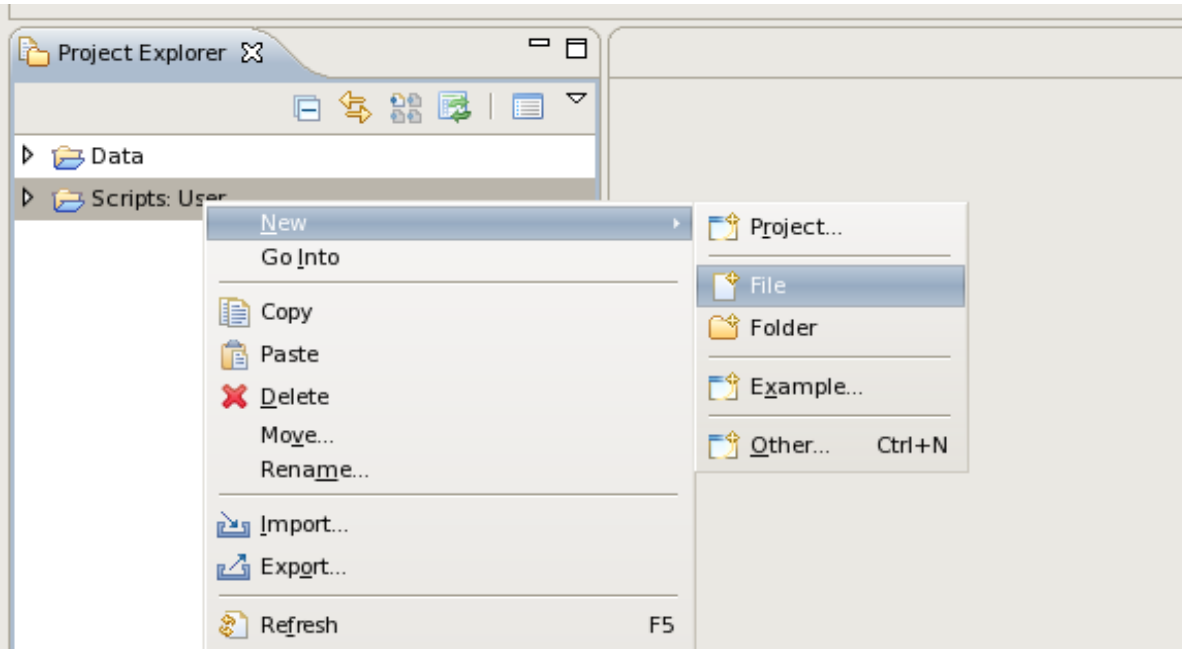
myScannable.**waitWhileBusy**()
: returns when the Scannable has finished moving

myScannable.**a**(*10*)
: asynchronous move: instructs the Scannable to move and then returns immediately

myScannable.**ar**(*1*)
: asynchronous relative move

6.1 How to create a Scannable

Create a script file. Open the Scripts perspective. Within the Project Explorer view expand the 'Scripts: User' folder to expose the 'src' child of this folder. Select the 'src' item, press the right mouse button and select New File.



In the "New File" dialog enter the filename (with extension .py) and press Finish. The new file will be created and will be displayed in the editor view ready for you to start adding content.

To run the script enter the use the run command, e.g. to run the code in the script file MyScript.py:

```
>run "MyScript.py"
```

If the script file contains the definition of a Scannable, i.e. class , then you do not run the script but rather 'import' it and then issue a command to create an instance of the object:

```
>import MyScannable.py
```

If you change the content of MyScannable you need to reload it:

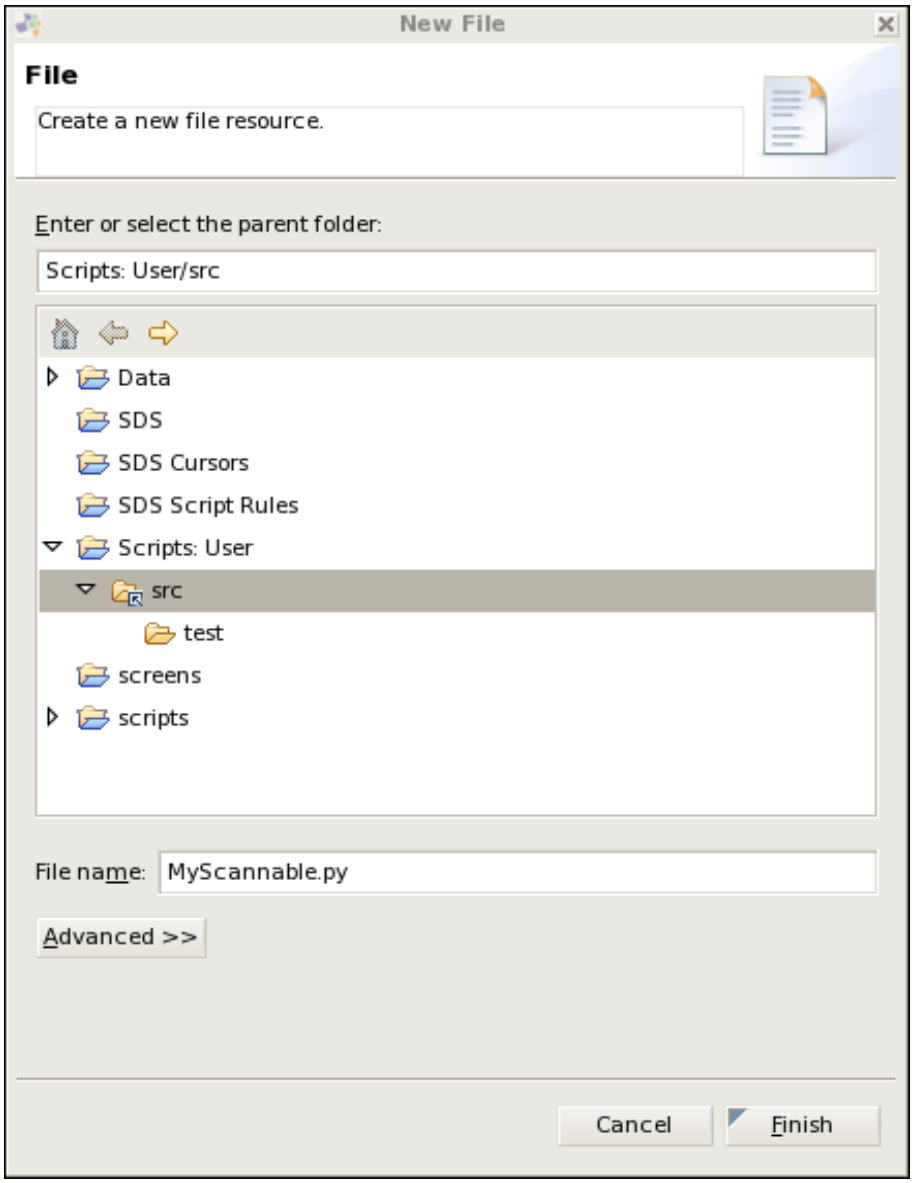
```
>reload(MyScannable)
```

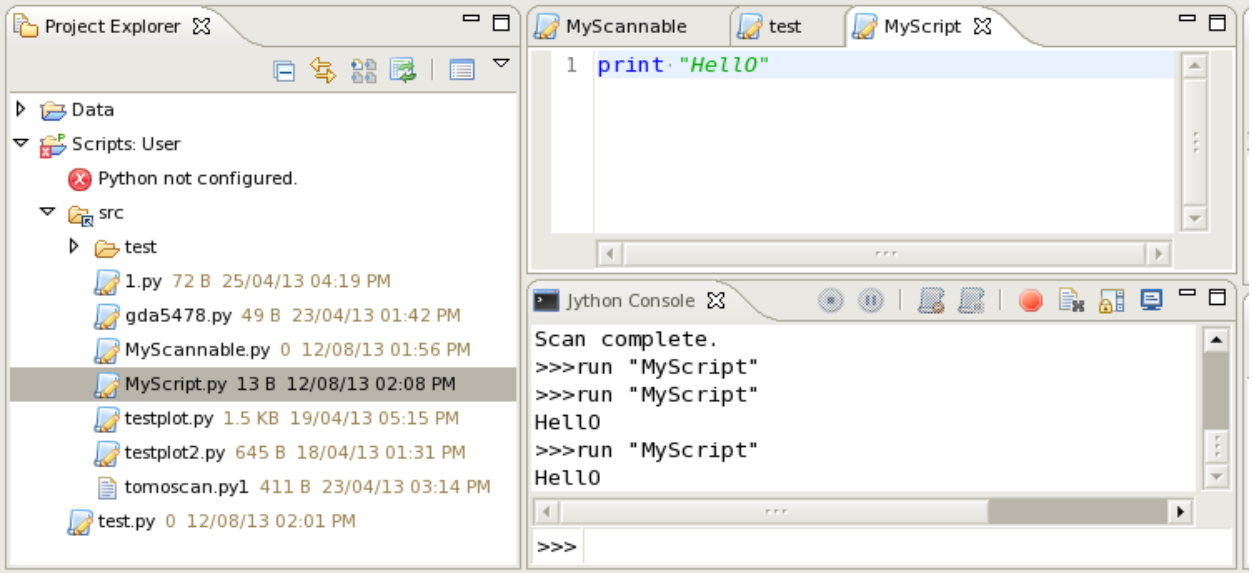
After importing the file you can instantiate an instance of the class defined in the file by executing its constructor, using the code shown below

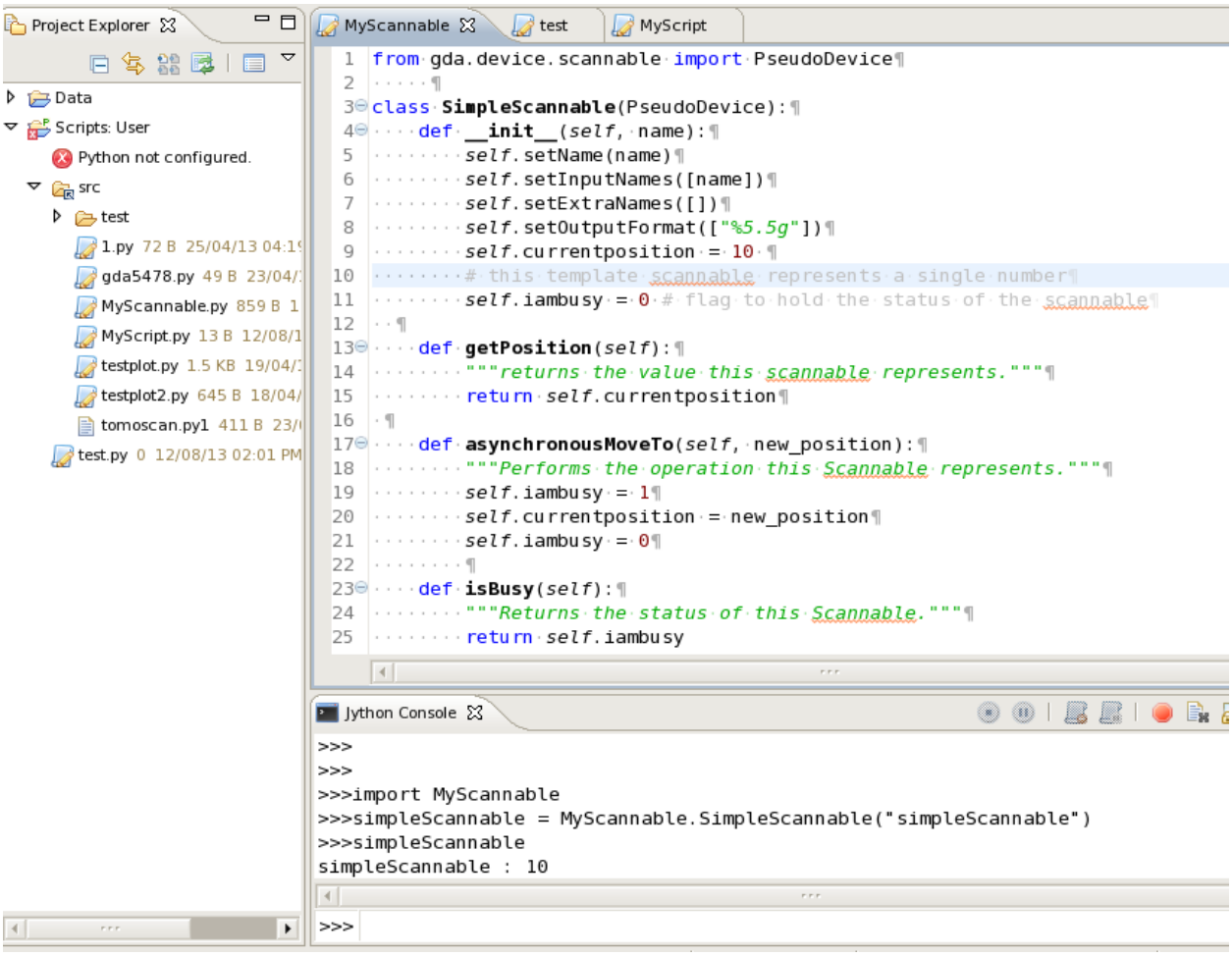
```
> simpleScannable = MyScannable.SimpleScannable("simpleScannable")
```

6.2 Basic Template

To write your own Scannable, you need to extend a base class which provides most of the functionality required. You simple then need to implement three methods relating to returning the position, performing the operation and returning the status. A template for this most simple case is shown below:


New File
File
Create a new file resource.
Enter or select the parent folder:
Scripts: User/src
Data
SDS
SDS Cursors
SDS Script Rules
Scripts: User
src
test
screens
scripts
File name:
MyScannable.py
Advanced >>
Cancel
Finish


Project Explorer
Data
Scripts: User
Python not configured.
src
test
1.py 72 B 25/04/13 04:19 PM
gda5478.py 49 B 23/04/13 01:42 PM
MyScannable.py 0 12/08/13 01:56 PM
MyScript.py 13 B 12/08/13 02:08 PM
testplot.py 1.5 KB 19/04/13 05:15 PM
testplot2.py 645 B 18/04/13 01:31 PM
tomoscan.py1 411 B 23/04/13 03:14 PM
test.py 0 12/08/13 02:01 PM
MyScannable
test
MyScript
1 print "Hello"
Jython Console
Scan complete.
>>>run "MyScript"
>>>run "MyScript"
Hello
>>>run "MyScript"
Hello
>>>


Project Explorer
Data
Scripts: User
Python not configured.
src
test
1.py 72 B 25/04/13 04:1
gda5478.py 49 B 23/04/
MyScannable.py 859 B 1
MyScript.py 13 B 12/08/1
testplot.py 1.5 KB 19/04/
testplot2.py 645 B 18/04/
tomoscan.py1 411 B 23/
test.py 0 12/08/13 02:01 PM
MyScannable
test
MyScript
1 from gda.device.scannable import PseudoDevice
2
3 class SimpleScannable(PseudoDevice):
4 def __init__(self, name):
5 self.setName(name)
6 self.setInputNames([name])
7 self.setExtraNames([])
8 self.setOutputFormat(["%5.5g"])
9 self.currentposition = 10
10 # this template scannable represents a single number
11 self.iambusy = 0 # flag to hold the status of the scannable
12
13 def getPosition(self):
14 """returns the value this scannable represents."""
15 return self.currentposition
16
17 def asynchronousMoveTo(self, new_position):
18 """Performs the operation this Scannable represents."""
19 self.iambusy = 1
20 self.currentposition = new_position
21 self.iambusy = 0
22
23 def isBusy(self):
24 """Returns the status of this Scannable."""
25 return self.iambusy
Jython Console
>>>
>>>
>>>import MyScannable
>>>simpleScannable = MyScannable.SimpleScannable("simpleScannable")
>>>simpleScannable
simpleScannable : 10
>>>

```
from gda.device.scannable import PseudoDevice
from gda.device.scannable import ScannableUtils

class MyScannable(PseudoDevice):
    def __init__(self, name):
        self.setName(name)
        self.setInputNames([name])
        self.setExtraNames([])
        self.setOutputFormat(["%5.5g"])
        self.currentposition = 10 # this template scannable represents a single number
        self.iambusy = 0 # flag to hold the status of the scannable

    def getPosition(self):
        """returns the value this scannable represents."""
        return self.currentposition

    def asynchronousMoveTo(self, new_position):
        """Performs the operation this Scannable represents."""
        self.iambusy = 1
        self.currentposition = new_position
        self.iambusy = 0

    def isBusy(self):
        """Returns the status of this Scannable."""
        return self.iambusy
```

Once these three methods have been implemented, the class can act like any other Scannable and be used in scans and within the other commands listed in this guide. The constructor is a bit more complex as it needs to define what numbers the Scannable holds. It is explained in more detail below.

6.3 The Scannable Constructor

The Scannable constructor shown in the basic template sets up the Scannable for use:

```
# constructor
def __init__(self, name):
    self.setName(name)
    self.setInputNames([name])
    self.setExtraNames([])
    self.setOutputFormat(["%5.5g"])
    self.currentposition = 10 # this template scannable represents a single number
    self.iambusy = 0 # flag to hold the status of the scannable
```

In this example The two last lines of this constructor sets the Scannable's starting value and creates a flag which will determine its busy status. These two attributes are specific to this class and are used in the three methods which the class implements.

The other lines in the constructor are obligatory for the Scannable to work properly and are designed to enable this class to interact properly with the base class which this class extends (namely PseudoDevice).The first of these lines sets the Scannable's name. This name is used when pretty-printing the class to the Jython terminal. The other three lines set up three arrays which define what numbers this Scannable represents, what they are called and what format that should be printed out in when pretty-printing the Scannable to the terminal:

```
self.setInputNames([name])
self.setExtraNames([])
self.setOutputFormat(["%5.5g"])
```

The InputNames array gives the size of the array that this Scannable's rawAsynchronousMoveTo expects. Each element of the InputNames array is a label for that element which is used in file headers etc. Note that this array can be zero-sized if required.

The ExtraNames array is used in a similar manner to the InputNames array, but lists additional elements in the array returned by the Scannable's rawGetPosition method i.e. the array returned by rawGetPosition may be larger than the array required by rawAsynchronousMoveTo. This allows for the possibility that a Scannable may hold and return more information than it needs in order to move or perform whatever operation it does inside its rawAsynchronousMoveTo method. This array would normally be zero-sized if required.

The OutputFormat array lists the formatting strings for both the elements of the InputNames and ExtraNames arrays. This is used when pretty-printing the output from rawGetPosition. It is vital that the size of the OutputFormat array is the same size as the sum of the sizes of the InputNames and ExtraNames arrays for the Scannable to work properly.

6.4 Methods available to implement in the Scannable Interface

There are a number of methods which may optionally be implemented in Scannable classes to extend their functionality. Most of them are for use during step scans in case you need extra operations outside of the normal operation of repeatedly moving Scannables and collecting data.

`scannable.`**`atPointStart`**`()`
: called immediately before moving any Scannables at every point in a step scan

`scannable.`**`atPointEnd`**`()`
: called immediately after collecting data at every point in a step scan

`scannable.`**`atScanLineStart`**`()`
: called at the start of every sub-scan in a multi-dimensional scan

`scannable.`**`atScanLineEnd`**`()`
: called at the end of every sub-scan in a multi-dimensional scan

`scannable.`**`atScanStart`**`()`
: called at the start of every scan

`scannable.`**`atScanEnd`**`()`
: called at the end of every scan

`scannable.`**`stop`**`()`
: called when aborting a scan or pressing the 'Panic Stop' button on the JythonTerminal. This should be used to enable aborting the operation being performed within the Scannable's rawAsynchronousMoveTo method.

`scannable.`**`toString`**`()`
: string implement this to override the pretty-print output for this Scannable

`scannable.`**`isPositionValid`**`()`
: given an object representing a position, this should return 1 (true) if the position is acceptable to this Scannable

6.5 More templates

Here are a number of more detailed templates, with annotation, which may be helpful:

6.5.1 Full Scannable template

A fuller version of the basic template shown before:

```
from gda.device.scannable import PseudoDevice

#
# A template for all Scannable classes.
#
# The rawIsBusy, rawGetPosition, and rawAsynchronousMoveTo methods must be
# implemented.
#
# The others (commented out here) are optional depending on how your scannable
# works.
#
#*****************************************************************************
# Note: the inputNames, extraNames and outputFormat arrays defined in __init__:
# Your Scannable could represent no numbers, a single number or an array of
# numbers. These arrays define what the Scannable represents.
#
# The inputNames array is a list of labels of the elements accepted by the
# new_position argument of the rawAsynchronousMoveTo method.
#
# The extraNames array is a list of labels of extra elements in case the array
# returned by rawGetPosition is larger than the array accepted by
# rawAsynchronousMoveTo.
#
# The outputFormat array is used when pretty-printing the scannable and lists
# the format to use for each element in the array returned by rawGetPosition.
# It is very important that the size of this array matches the sum of the sizes
# of inputNames and extraNames.
#
#
#*****************************************************************************
#
class scannableTemplate(PseudoDevice):

    #
    # The constructor.
    #
    def __init__(self, name):
        self.name = name
        self.currentposition = 10 # this scannable represents a single number
        self.setInputNames([name])
        self.setExtraNames([])
        self.setOutputFormat(["%5.5g"])
        self.iambusy = 0 # flag to hold the status of the scannable

    #
    # Returns the value represented by this Scannable. This should be a number
    # or an array of numbers
    #
    def rawGetPosition(self):
        return self.currentposition

    #
    # Does the work represented by this Scannable. If this takes a long time,
    # then you should run a separate thread from within this method. See the
    # threaded_scannable_template.py script for details on how to do this.
    #
    def rawAsynchronousMoveTo(self, new_position):
        self.iambusy = 1
```

```
        self.currentposition = new_position
        self.iambusy = 0

    #
    # Returns false (0) if the action started by rawAsynchronousMoveTo has been
    # completed
    #
    def rawIsBusy(self):
        return self.iambusy

    #
    # Called when panic stop called on the system.
    #
    # def stop(self):
    #     print str(self.name), "stop called!"

    #
    # Implement this to override the pretty-print version of this Scannable
    #
    # def toString(self):
    #     return self.name

    #
    # Given an object, this returns true (1) if that object is a valid position
    # for this scannable to use in its rawAsynchronousMoveTo method
    #
    # def isPositionValid(self):
    #     return 1

    #
    # Called just before every node in a scan
    #
    # def atPointStart(self):
    #     print str(self.name),"doing atPointStart()!"

    #
    # Called after every node in a scan
    #
    # def atPointEnd(self):
    #     print str(self.name), "doing atPointEnd()!"

    #
    # In multi-dimensional scans, called before each line in the scan
    # This is still called once in single dimensional scans.
    #
    # def atScanLineStart(self):
    #     print str(self.name), "doing atScanStart()!"

    #
    # In multi-dimensional scans, called after each line in the scan
    # This is still called once in single dimensional scans.
    #
    # def atScanLineEnd(self):
    #     print str(self.name), "doing atScanEnd()!"

    #
    # Called at the start of the scan (called once in multi-dimensional scans)
    #
```

```
    # def atScanGroupStart(self):
    #     print str(self.name), "doing atGroupStart()!"

    #
    # Called at the end of the scan (called once in multi-dimensional scans)
    #
    # def atScanGroupEnd(self):
    #     print str(self.name), "doing atGroupEnd()!"
```

6.5.2 Threaded Scannable Template

A Scannable whose operation inside rawAsynchronousMoveTo is run in a separate thread:

```
from gda.device.scannable import PseudoDevice
from java.lang import Thread, Runnable

#
# Threaded version of the class in scannable_template in case the work performed
# within the rawAsynchronousMoveTo takes a long time.
#
class threadedScannableTemplate(PseudoDevice):

    #
    # The constructor.
    #
    def __init__(self, name):
        self.name = name
        self.currentposition = 10 # this scannable represents a single number
        self.setInputNames([name])
        self.setExtraNames([])
        self.setOutputFormat(["%5.5g"])
        self.iambusy = 0 # flag to hold the status of the scannable

    #
    # Returns the value represented by this Scannable. This should be a number
    # or an array of numbers
    #
    def rawGetPosition(self):
        return self.currentposition

    #
    # Creates a new moveScannableThread object to do the work and then starts it
    # in a new thread.
    #
    def rawAsynchronousMoveTo(self, new_position):
        self.iambusy = 1
        newThread = moveScannableThread(self, new_position)
        t = Thread(newThread)
        t.start()

    #
    # Returns false (0) if the action started by rawAsynchronousMoveTo has been
    # completed
    #
    def rawIsBusy(self):
        return self.iambusy
```

```
#
# An object called internally by the threadedScannableTemplate.
#
# It is very important that the busy flag is set to 0 at the end of
# the run method.
#
class moveScannableThread(Runnable):

    #
    # Constructor for this class
    #
    def __init__(self, theScannable, new_position):
        self.myScannable = theScannable
        self.target = new_position

    #
    # Does the work to move what the Scannable represents. This is run in a new
    # thread started by the line in rawAsynchronousMoveTo: t.start()
    #
    def run(self):
        print "you have asked me to move to", str(self.target)
        self.myScannable.currentposition = self.target
        self.myScannable.iambusy = 0
```

6.5.3 Detector Template

For an object which you wish to act like a Detector:

```
from gda.device.detector import PseudoDevice

#
# A template class to use as a basis to create your own Detector objects.
#
# Detectors must work in the following manner:
#     - a call to collectData to collect some new data. Ideally this should be
#       asynchronous (i.e. the function returns immediately and the work is done
#       in a new Thread). See threaded_detector_template.py for this.
#     - repeated calls to getStatus may be made by external classes to see if
#       data is still being collected.
#     - once getStatus returns false (0) then a call to readout maybe made by
#       external classes to collect the data.
#
#
class templateDetectorClass(PseudoDevice):

    #
    # The constructor.
    #
    def __init__(self, name):
        self.setName(name)
        self.isCollecting = 0
        self.myData = 0

    #
    # Performs the work to collect some data. This method should not return the
    # data, but instead keep the status field up to date.
```

```
    #
    def collectData(self):
        self.isCollecting = 1
        print "you have asked me to collect data!"
        self.myData += 1
        self.isCollecting = 0

    #
    # Returns true (1) if this object is busy collecting data
    #
    def getStatus(self):
        return self.isCollecting

    #
    # Returns the last data which was collected. This should only be called when
    # getStatus returns false
    #
    def readout(self):
        return self.myData
```

6.5.4 Threaded Detector Template

A threaded version of the Detector template:

```
from java.lang import Thread, Runnable
from gda.device.detector import PseudoDevice

#
# A more complex template for detectors in which the work to perform the data
# collection is performed in its own thread.
#
#
#
class threadedTemplateDetectorClass(PseudoDevice):

    #
    # The constructor.
    #
    def __init__(self, name):
        self.setName(name)
        self.isCollecting = 0
        self.myData = 0

    #
    # Performs the work to collect some data. This method should not return the
    # data, but instead keep the status field up to date.
    #
    def collectData(self):
        self.isCollecting = 1
        newThread = collectDataThread(self)
        t = Thread(newThread)
        t.start()

    #
    # Returns true (1) if this object is busy collecting data
    #
    def getStatus(self):
```

```
        return self.isCollecting

    #
    # Returns the last data which was collected. This should only be called when
    # getStatus returns false
    #
    def readout(self):
        return self.myData;

#
# A method called internally by the threadedTemplateDetectorClass to collect
# the data in a separate thread.
#
# It is very important that the isCollecting flag is set to 0 at the end of
# this method.
#
class collectDataThread(Runnable):

    def __init__(self, theDetector):
        self.myDetector = theDetector

    def run(self):
        print "you have asked me to collect data!"
        self.myDetector.myData +=1
        self.myDetector.isCollecting = 0
```

6.5.5 Multi-Dimensional Scannable Template

An example of how to write a Scannable which represents an array of numbers, including more output from the position command than what is received by the move method:

```
from gda.device.scannable import PseudoDevice

class MultiElementTestClass(PseudoDevice):

    def __init__(self):
        self.setName("y")
        self.setInputNames(["first","second"])
        self.setExtraNames(["third"])
        self._position = [20,30]
        self.setOutputFormat(["%4.10f","%4.10f","%4.10f"])

    def rawGetPosition(self):
        return self._position + [10]

    def rawAsynchronousMoveTo(self,new_position):
        if len(new_position) == 2:
            self._position=new_position

    def rawIsBusy(self):
        return 0
```

CHAPTER

SEVEN

DATA ANALYSIS

7.1 Plotting

The Plotter object provides a different set of methods to plot one or two-dimensional data either in one, two or three dimensions. This chapter will give an example for each of these different plotting types.

7.1.1 1D plots

7.1.2 2D image plots

7.1.3 Viewing Images

The ability to view images from many of the detectors at diamond has been incorporated into the GDA. Currently the images from the following detectors can be loaded into the GDA:

- ADSC Detectors on the MX beamlines *ADSCLoader*
- MAR Detectors *MARLoader*
- Pilatus Tiffs *PilatusTiffLoader*
- CBF files *CBFLoader*
- JPEG, TIFF and PNG images *JPEGLoader,TIFFLoader and PNGLoader*

The above can be used in the following way within the Jython terminal:

```
>>> from gda.analysis.io import *
>>> sfh = ScanFileHolder()
>>> sfh.load(FileReader("FileName"))
>>> Plotter.plotImage("Data Vector", sfh[0])
```

Images can be saved from a ScanFileHolder in either PNG or JPEG format (PNGSaver and JPEGSaver) with the possibility of scaling the image such that it will be able to be saved in the requested format. The upper limit of JPEG is 255 and PNG is 65535 for any intensity of pixel value. If the pixel intensity is greater than this value the image can be scaled to fit the maximum depth of the image format. The images can be scaled using PNGScaledSaver and JPEGScaledSaver:

```
>>> from gda.analysis.io.import
>>> # Assuming that a scan file holder has been created
>>> # containing the data and is called 'sfh'
>>> sfh.save(FileSave("FileName"))
```

If the ending is not specified the proper ending will be added and if there are many images they will be called 'Image00001.xxx, Image00002.xxx' etc.

7.1.4 3D surface plots

This chapter will provide a step by step example on how to generate a 3D surface plot of a two dimensional data or image file

- Generate a ScanFileHolder object:

```
>>> data = ScanFileHolder()
```

- Load the data file:

```
>>> data.load(OtokoLoader("/s/Science/DASC/LinuxFiles/B09000.806"))
```

- Display the data as a 3D surface plot:

```
>>> Plotter.plot3D("Data Vector",data[0])
```

For the third line to work you have to make sure that you have the Data Vector panel in your actual configuration. This is the simplest way to surface plot a 2D dataset. When no third parameter is provided and the data contained in the ScanFileHolder is too large to display at once either because there is not enough memory available or the display refresh rate would be too low and therefore not interactive it will automatically subsample. If you prefer not to subsample but rather would like to display a subset of the data as a window on the whole data, change the last command to this:

```
>>> Plotter.plot3D("Data Vector",data[0],True)
```

If the dataset has more than two dimensions. The plotter will automatically choose the first two dimensions for generating the surface plot. If the dataset has only one dimension it still will work but the plotter will automatically switch to a different usage mode that allows ploting a series of one dimensional datasets (see the section called 1D plots).

CHAPTER

EIGHT

PLOTTING

8.1 ScanFileHolder Class

The first class to be detailed here is the ScanFileHolder. This class acts as storage for a single scan, the idea behind this is that multiple scans can be loaded in, and then either checked or compared as is required. The basic functionality shown here can all be incorporated into scripts or used as is in the jython terminal. All the examples shown here are for use in the Jython terminal, and are based on some scans being performed on simulated or real components.

8.1.1 Basic introduction to Loading and Visualising files

The Data Analysis and Visualisation toolkit for the GDA is designed to make the visualisation and manipulation of all data collected on the beam-line quick and effective. This documentation will concentrate on demonstrations and examples, so having a working copy of the GDA attached to the DLS network would be useful. Lets Start with a brief example of how to load in a scan file and visualise the data. To Start with the following script is an example of loading in a previous scan and visualising some of the data.

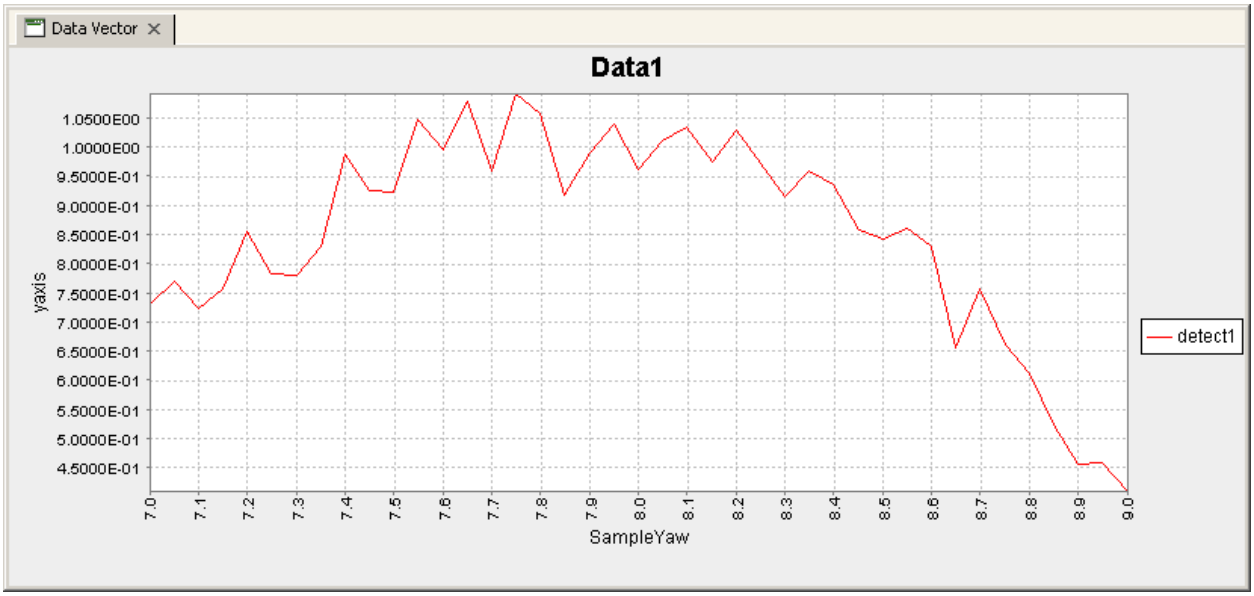
```
>>> data = ScanFileHolder()
>>> data.loadSRS()
>>> data.info()
Trying to open /home/ssg37927/gda/gda-7.4/users/data/70.dat
ScanFileContainer object containing 41 lines of data per DataSet.
0        SampleYaw
1        detect1
2        detect1
3        time
4        I0
5        It
6        If
7        TEY
8        PEY
9        FY
10       Drain
>>> data.plot(0,1)
```

This should display the following in the “Data Vector” tab in the GDA:

Remember that you can undock the “Data Vector” tab and put it wherever is easiest to see it, such as on another screen or beside the Jython terminal.

Going through the script line by line:

01 Creates the object which will hold all the data we want to manipulate and visualise.



02 This command tells the data object to load in the last SRS datafile created by the GDA. This can be expanded to various different things depending on what is needed. If a positive integer is given(###), then the SRS datafile of that name is loaded (###.dat) from the current working directory. If a negative integer is given, then the scan performed that many times previously is loaded, for example loadSRS(-2) will load the scan performed 2 before the current one. If the full path and filename to a particular SRS file is given, then that is the one which will be loaded.

03 This command displays a an outline of the data inside the object. This allows easy identification of files as well as direction for the plotting commands and others

04-16 This is the output from the previous command, showing information about what has been read in from the file.

17 This command plots data from the object out to the Data Vector panel

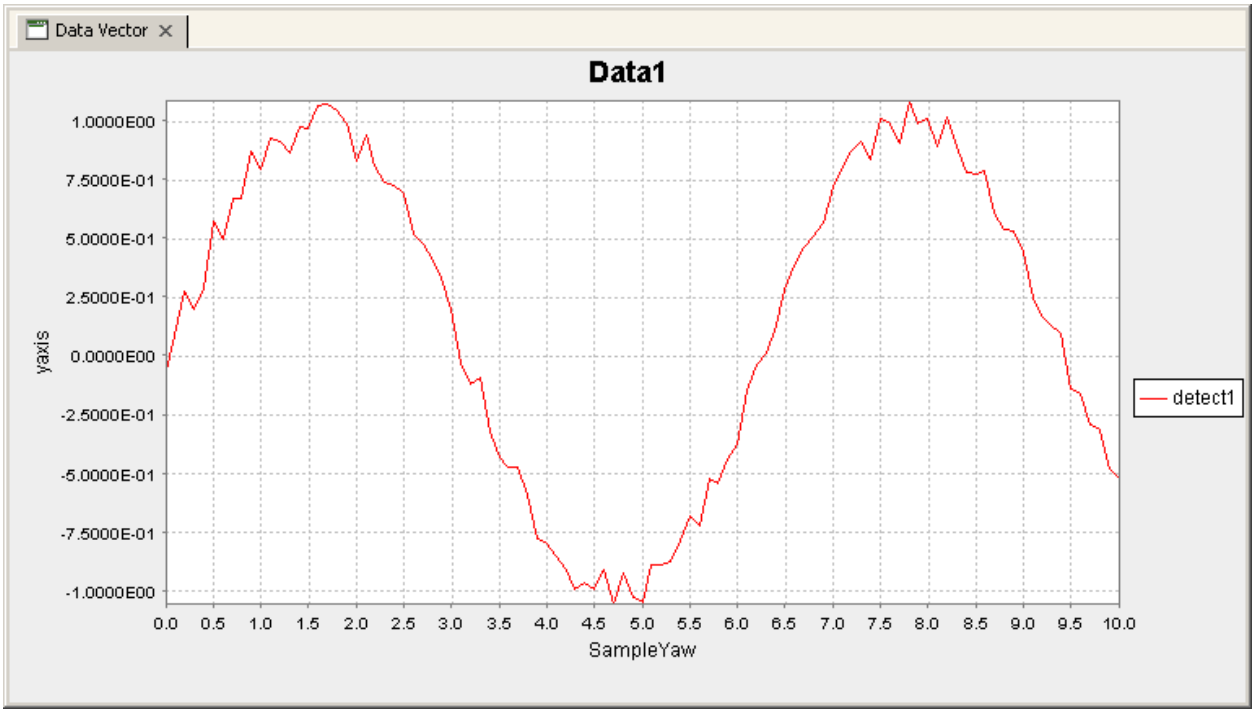
8.1.2 Plotting examples

One of the functions available in the Analysis and Visualisation toolkit is the ability to quickly look at collected data from within the GDA. In the previous example we saw a simple plot of 1 set of collected data against another, but there is other functionality available. The following script uses Various different forms of the print command. It Also makes use of another version of the "loadSRS" command. This only works however on scans which have been recorded in the current directory and won't work if the data directory has been changed since the scan was taken.

```
>>> data = ScanFileHolder()
>>> data.loadSRS(-13)
>>> data.info()
ScanFileContainer object containing 101 lines of data per DataSet.
0    SampleYaw
1   detect1
2    detect1
3    time
4    I0
5    It
6   If
7    TEY
8    PEY
9   FY
```

```
10   Drain
11   Channel_0
12   Channel_1
13   Channel_2
14   Channel_3
15   Channel_4
16   Channel_5
17   Channel_6
18   Channel_7
>>> data.plot(0,1)
>>> data.plot("SampleYaw",11)
>>> data.plot("SampleYaw",[11,"Channel_3",1])
```

Should provide the following outputs:



01 creates the object which will hold all the data we want to manipulate and visualise.

02 Loads in the scan that was taken 13 scans ago.

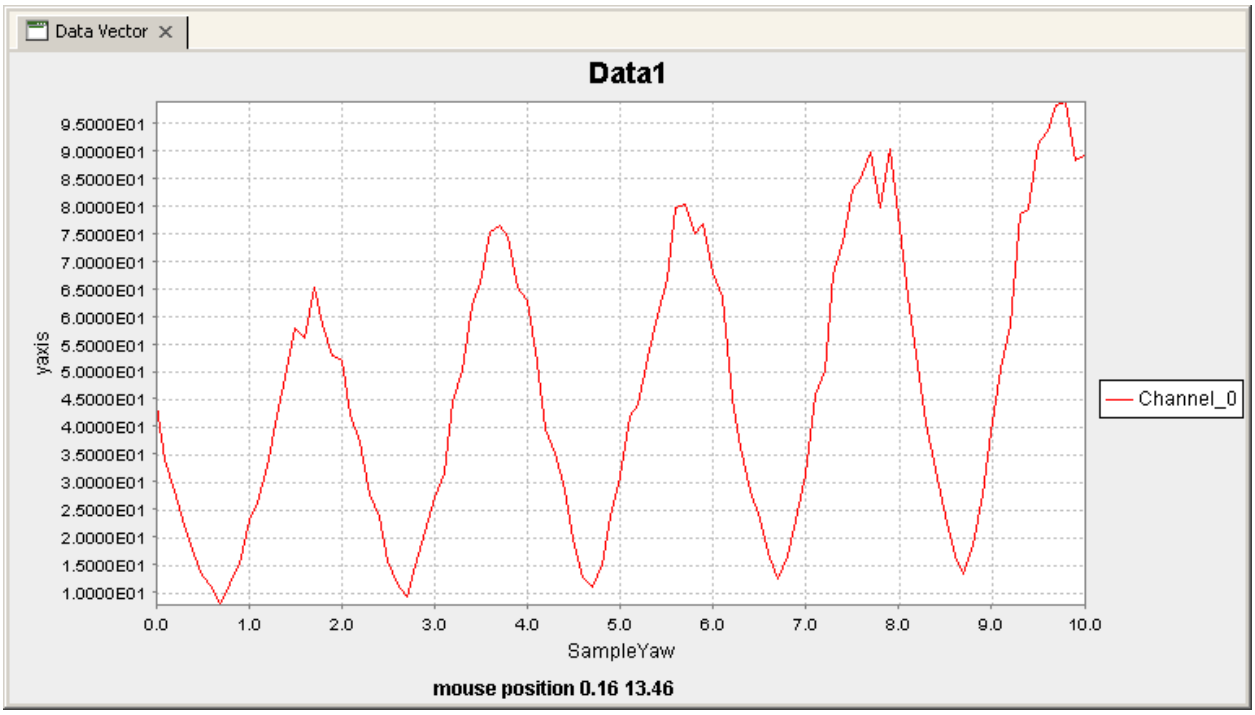
03 This command displays a an outline of the data inside the object. This allows easy identification of files as well as direction for the plotting commands and others

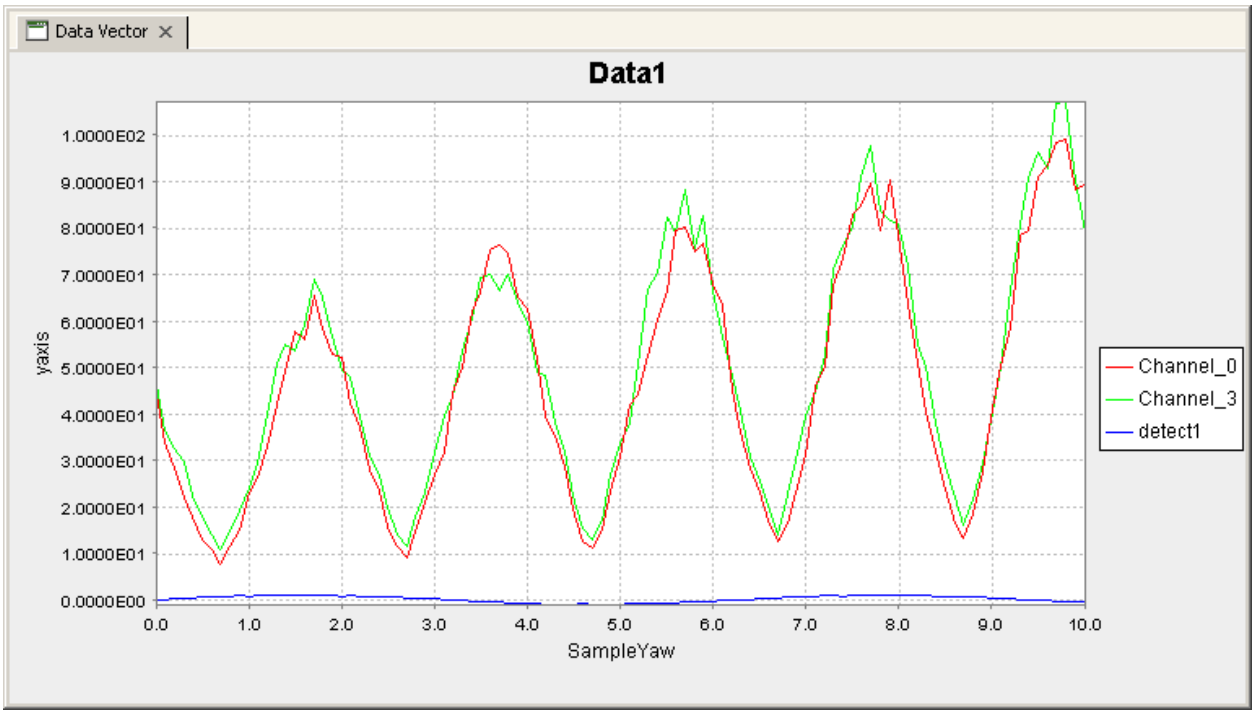
04-23 This is the output from the previous command, showing information about what has been read in from the file.

24 This simple plot command plots the data highlighted in line 05 of the code, against that in line 06 of the code. These are marked by the numbers 0 and 1, and that corresponds to there places in the file. The first value is the x axis, and the second is the Y axis which is to be plotted.

25 This plot command makes use of the ability to use either the number representing the data, or the name of the data to plot the information.

26 In this plot, a list is used as the y axis, and as can be seen from the diagram, all of the y axis are plotted here on the diagram. This also shows that numbers and names can be mixed in the list freely, making access to specific data relatively easy.


Data Vector
Data1
9.5000E01
9.0000E01
8.5000E01
8.0000E01
7.5000E01
7.0000E01
6.5000E01
6.0000E01
5.5000E01
5.0000E01
4.5000E01
4.0000E01
3.5000E01
3.0000E01
2.5000E01
2.0000E01
1.5000E01
1.0000E01
yaxis
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
SampleYaw
Channel_0
mouse position 0.16 13.46


Data Vector
Data1
1.0000E02
9.0000E01
8.0000E01
7.0000E01
6.0000E01
5.0000E01
4.0000E01
3.0000E01
2.0000E01
1.0000E01
0.0000E00
yaxis
0.0
1.0
2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
SampleYaw
Channel_0
Channel_3
detect1

8.1.3 Retrieving Data

The next section shows a lot of information about manipulating data once it is in the form of a dataset. The ScanFileHolder is essentially a container for lots of different datasets, and getting them out to look at more closely can be done in several ways.

```
>>> data = ScanFileHolder()
>>> data.loadSRS(-31)
>>> data.info()
ScanFileContainer object containing 21 lines of data per DataSet.
0    SampleYaw
1    detect1
2    time
3    I0
4    It
5    If
6    TEY
7    PEY
8    FY
9    Drain
10   Channel_0
11   Channel_1
12   Channel_2
13   Channel_3
14   Channel_4
15   Channel_5
16   Channel_6
17   Channel_7
>>> dataset1 = data.getAxis("detect1")
>>> dataset1.disp()
DataVector Dimentions are [21]
[-5.5897e-02, 2.4862e-02, 2.9554e-01, 2.7648e-01, ...
>>> dataset1 = data.getAxis(10)
>>> dataset1.disp()
DataVector Dimentions are [21]
[1.9380e+01, 1.4850e+01, 2.1220e+01, 2.9110e+01, ...
>>> dataset1 = data[11]
>>> dataset1.disp()
DataVector Dimentions are [21]
[2.2580e+01, 1.4330e+01, 2.4240e+01, 3.1930e+01, ...
```

01 creates the object which will hold all the data we want to manipulate and visualise.

02 Loads in the scan that was taken 31 scans ago.

03 This command displays a an outline of the data inside the object. This information will be used here to assist in getting particular data out of the object

04-22 This is the output from the previous command, showing information about what has been read in from the file.

23 This command creates a dataset from the information in the ScanFileHolder associated with the title "detect1"

24 This command displays the information inside the new dataset, including its size and the data contained within.

25-26 The output from the disp() command

27 This command creates a dataset from the information in the ScanFileHolder associated with the position 10. This represents "Channel_0" as seen in line 15

28 The disp() command again, showing the now different data held in the dataset.

29-30 The output from the disp() command

31 This command uses the Pycontainer aspect of the dataset to allow very quick access to the ScanFileHolder.

32 The disp() command again, showing the now different data held in the dataset.

33-34 The output from the disp() command

8.1.4 Full Method Listing

The table provided here provides in full detail the main functions of the ScanFileHolder Class

class `ScanFileHolder`

`getAxis` (*int n*)
Returns a Dataset corresponding to the input number.

`getAxis` (*String name*)
Returns a Dataset corresponding to the input String in the header.

`getImage` ()
Returns a copy of the Dataset that contains the image data, if an image has been loaded into the ScanFileHolder. *Deprecated*

`getPilatusConversionLocation` ()
Returns the location of the Pilatus conversion software

`setPilatusConversionLocation` (*String path*)
Sets the location of the Pilatus conversion software

`getPixel` (*int x*, *int y*)
Returns the Double value of the pixel at (x,y)

`info` ()
Prints a description of the ScanFileHolder to the Jython terminal

`loadPilatusData` (*String path*)
Loads in the Pilatus tif specified in the input, this needs to be a full filename specification.

`loadSRS` (*int scan_no*)
If the number is positive, then this is the number of the SRS datafile which will be loaded from the current directory. If the number is negative, then the current file number is found, and the input number removed from it, and that SRS file is loaded in. For example if the current scan number is 12 and -4 is input, then "8.dat" will be loaded.

`loadSRS` ()
Loads in the data from the last scan to be performed

`loadSRS` (*String path*)
Loads in the SRS datafile specified by the filename in the input

`ls` ()
Displays to the Jython Terminal all the data names in the ScanFileHolder

`plot` ()
This plots the Image to the "Data Vector" tab in the GUI

`plot` (*int axis*)
Plots the axis specified from a number of sources. If the input is a string, then that is the name of the axis to be plotted. If the input is a number, then that is the number of the axis to be plotted. If the input is a dataset, then that dataset is plotted.

plot (*int xaxis*, *int yaxis*)
As the above plot, but the yAxis can also accept a list of any type of parameter as specified above

setImage (*DataSet*)
Copies the input dataset into the image dataset in the object. *Deprecated*

setImag (*int height*, *int width*, *double[] Data*)
Generates a dataset from the input variables and then copies it into the image dataset in the object. *Deprecated*

8.2 DataSet Class

The DataSet class is the core of all the fitting and visualisation architecture presented here. It is a N dimensional expendable container, which can be manipulated and visualised in many different ways. As was seen in the section on ScanFileHolders, the internals of these objects are datasets. In this section we will detail some of the uses of the dataset including manipulation and direct plotting.

8.2.1 Initial use of the dataset

The DataSet is the main way of handling sets of data in the GDA. There are many different ways of manipulating the dataset data. The following script demonstrates a few of these methods, as well as showing how to get the datasets out of a ScanFileHolder.

```
>>> data = ScanFileHolder()
>>> data.loadSRS(1)
>>> data.info()
ScanFileHolder object containing 26 lines of data per DataSet.
0    SampleYaw
1    detect1
2    detect2
>>> dataset1 = data[1]
>>> Plotter.plot("Data Vector",data[0],dataset1)
>>> dataset1.max()
1.0630006379056933
>>> dataset1.min()
-0.9609246487585549
>>> dataset1 -= dataset1.min()
>>> dataset1 /= dataset1.max()
>>> dataset1.max()
1.0
>>> dataset1.min()
0.0
>>> Plotter.plot("Data Vector",data[0],dataset1)
```

The idea here is to normalise the data between O and 1, and then to plot it out. This should Produce graphical output as shown below.

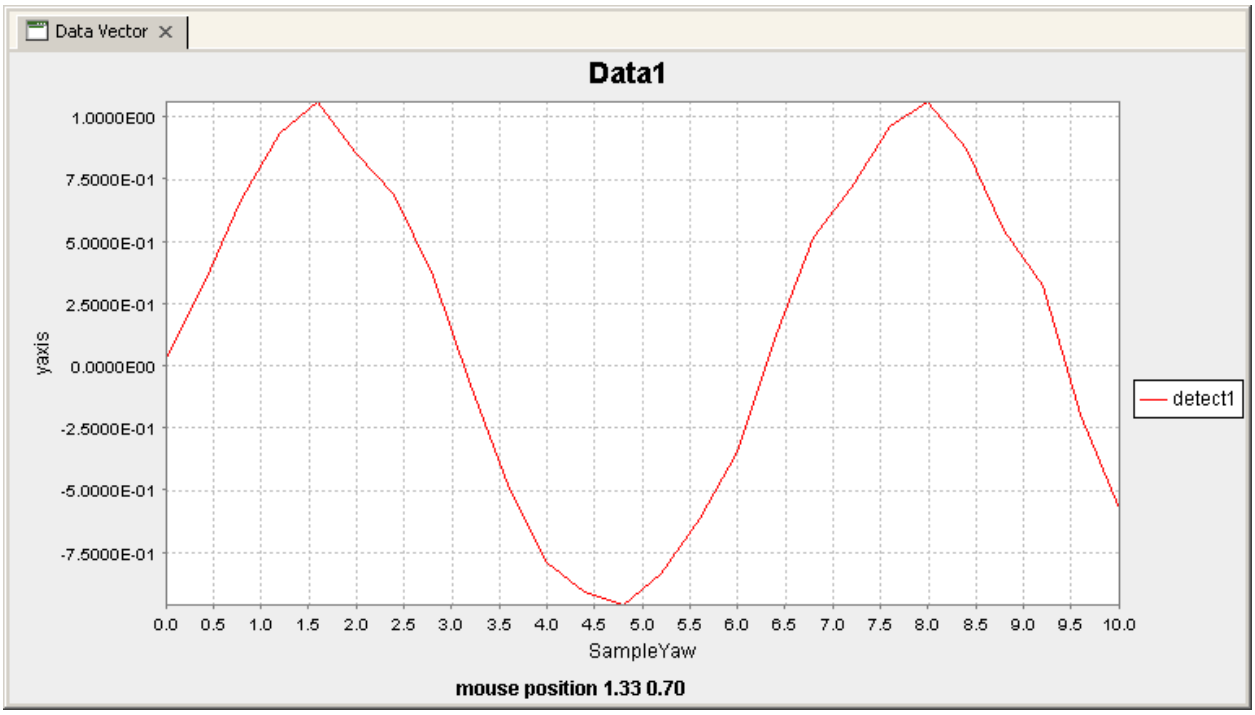
01 creates the object which will hold all the data we want to manipulate and visualise.

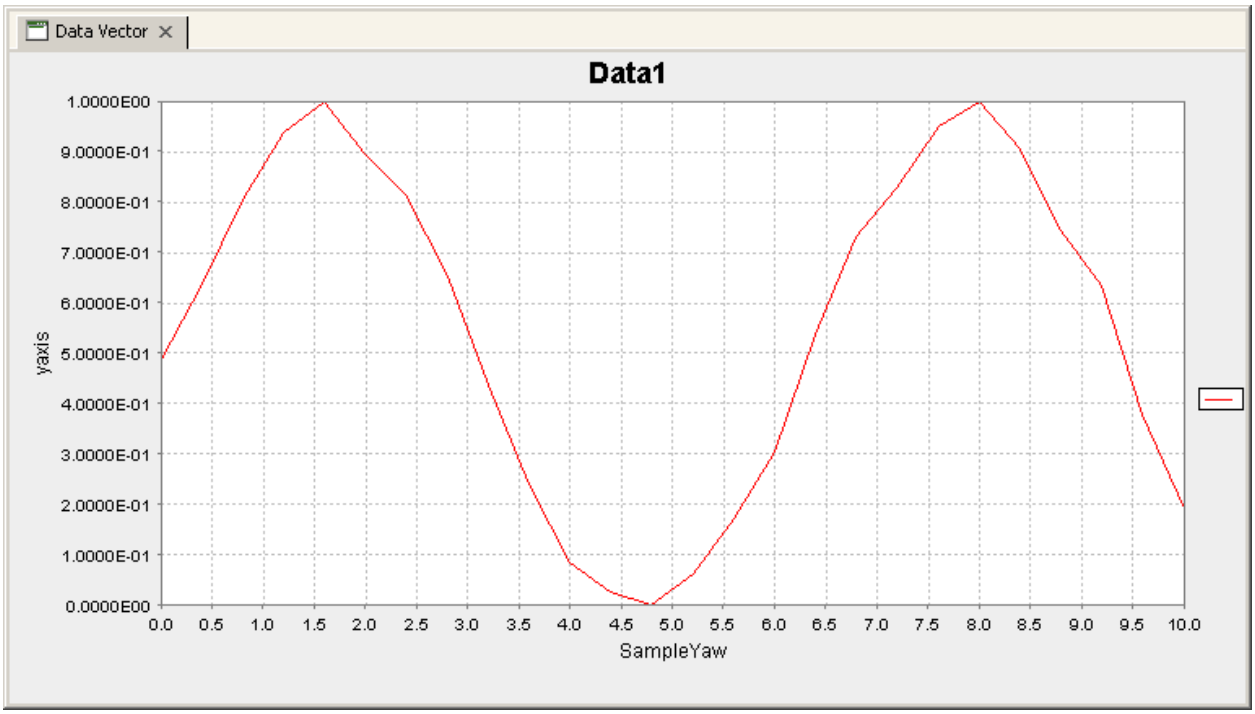
02 Loads in the scan that has the identifier "1.dat".

03 This command displays a an outline of the data inside the object.

04-07 This is the output from the previous command, showing information about what has been read in from the file.

08 This command creates a dataset from the information in the ScanFileHolder associated with the position 1 in the ScanFileHolder. In this case, this is the data which was collected by "detect1"


Data Vector
Data1
1.0000E00
7.5000E-01
5.0000E-01
2.5000E-01
0.0000E00
-2.5000E-01
-5.0000E-01
-7.5000E-01
yaxis
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
SampleYaw
detect1
mouse position 1.33 0.70


Data Vector
Data1
1.0000E00
9.0000E-01
8.0000E-01
7.0000E-01
6.0000E-01
5.0000E-01
4.0000E-01
3.0000E-01
2.0000E-01
1.0000E-01
0.0000E00
yaxis
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
SampleYaw

09 This command plots the information to the plotting window using the core functions which are wrapped by the ScanFileHolder class in the .plot() command. The first entry is the name of the panel to which the drawing should occur, the second is a dataset containing the x axis information, in this case gathered directly from the ScanFileHolder using the collection Jython methods used before, i.e. data[0]. The third value is the y axis, once again, this can be a dataset, or a list of datasets.

10-11 This uses one of the various datamining functions of the dataset, this returns the maximum element value in the dataset.

12-13 This command returns the minimum element value of the dataset.

14 This line makes use of the datasets a -= b operator which does the equivalent of a = a - b. In this case we are subtracting the minimum value of the dataset from every element of the dataset.

15 This line takes the dataset and divides every element by the maximum value of the dataset.

16-19 Running the previous commands to check that the max value is now 1.0 and the minimum value is now 0.0.

20 Replot the data, to show that it now lies between 0.0 and 1.0, and is still the same profile as previously.

8.2.2 Combinations of DataSets, and statistical analysis

Sometimes it can be useful to see the differences or similarities between two scans, or two detectors in the same scan. The following script highlights the noise between two different signals and evaluating that difference statistically.

```
1  >>> data = ScanFileHolder()
2  >>> data.loadSRS(1)
3  >>> data.info()
4  ScanFileHolder object containing 26 lines of data per DataSet.
5  0 SampleYaw
6  1 detect1
7  2 detect2
8  >>> dataset1 = data[1]
9  >>> dataset2 = data[2]
10 >>> Plotter.plot("Data Vector", data[0], [dataset1, dataset2])
11 >>> dataset3 = dataset1-dataset2
12 >>> Plotter.plot("Data Vector", data[0], [dataset1, dataset2, dataset3])
13 >>> dataset3.mean()
14 0.0067027887342955145
15 >>> dataset3.rms()
16 0.06266747949694666
17 >>> dataset3.skew()
18 0.5610499636907474
19 >>> dataset3.kurtosis()
20 -0.5355342349265815
```

This script looks at the difference between 2 detectors which are scanned in the same scan. This plots out a the data to make sure it is correct, and then to visualise the difference.

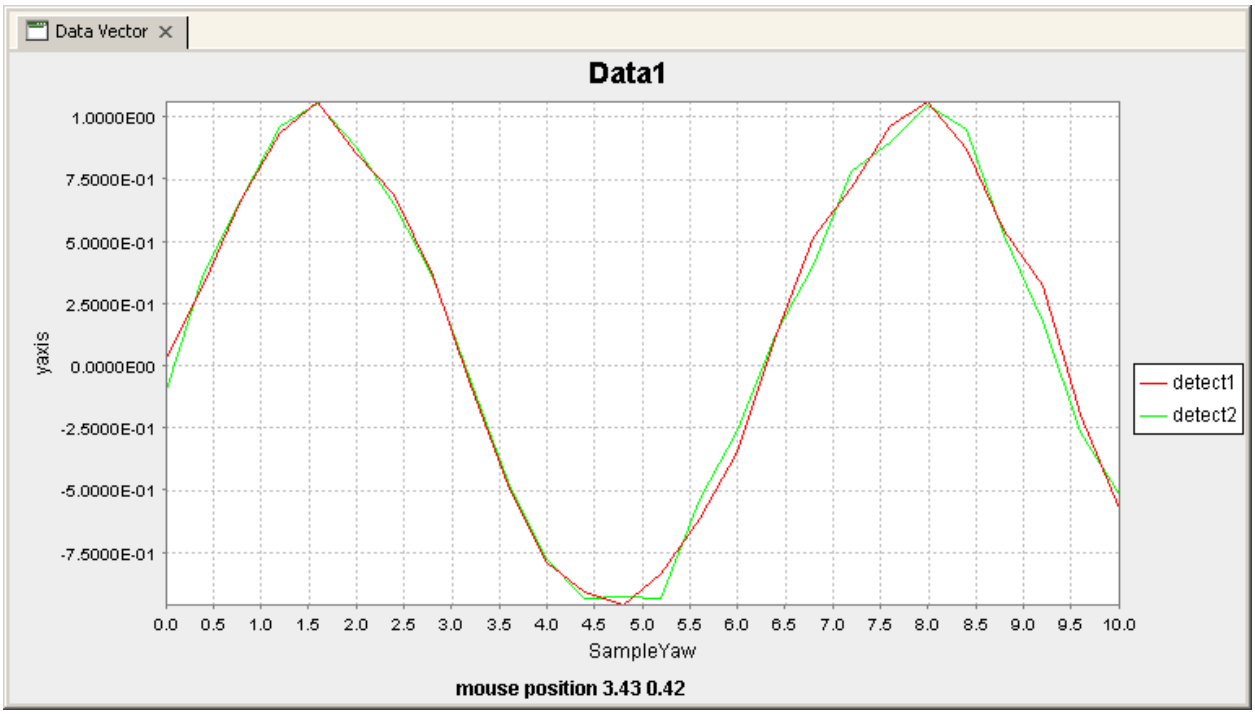
01 creates the object which will hold all the data we want to manipulate and visualise.

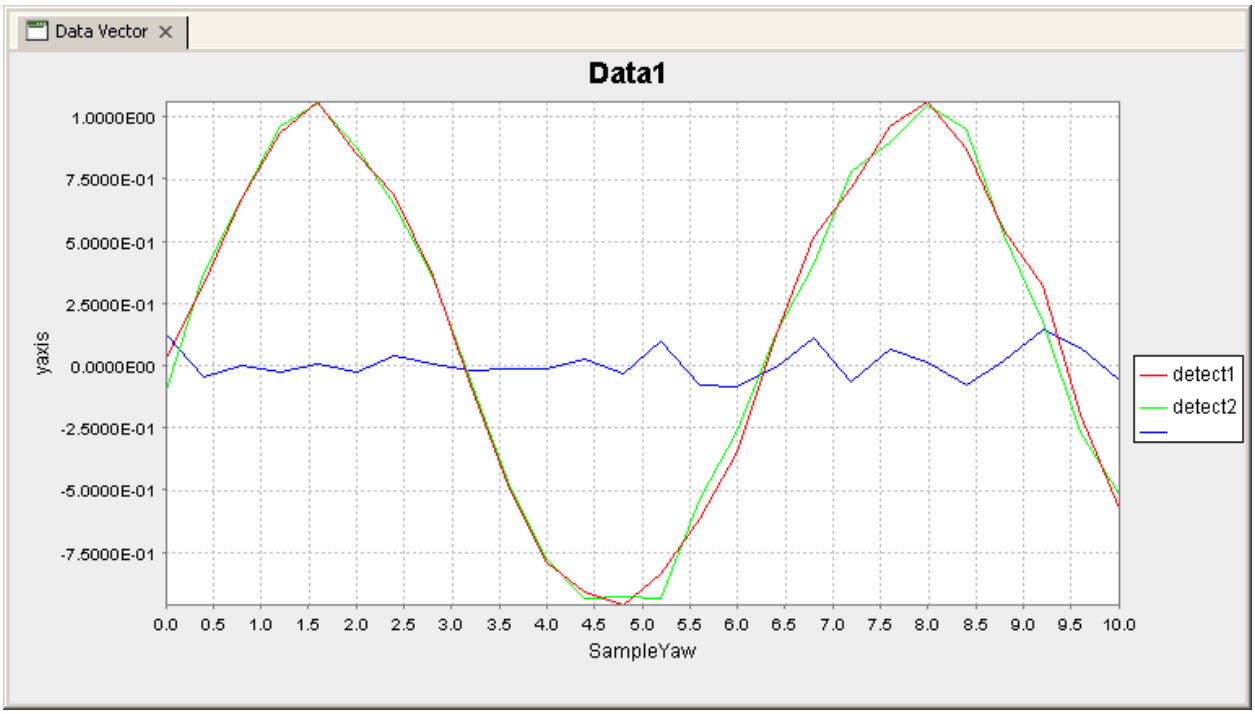
02 Loads in the scan that has the identifier "1.dat".

03-07 This command displays a an outline of the data inside the object.

08-09 These commands get the information from the ScanFileHolder corresponding to the "detect1", and "detect2" detectors.

10 This command plots both the datasets against the ScanFileHolders "SampleYaw" as this gives a nice x scale. Notice that both there are 2 y axis contained in the list given as the 3rd argument.


Data Vector
Data1
1.0000E00
7.5000E-01
5.0000E-01
2.5000E-01
0.0000E00
-2.5000E-01
-5.0000E-01
-7.5000E-01
yaxis
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
SampleYaw
detect1
detect2
mouse position 3.43 0.42


Data Vector
Data1
1.0000E00
7.5000E-01
5.0000E-01
2.5000E-01
0.0000E00
-2.5000E-01
-5.0000E-01
-7.5000E-01
yaxis
0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
SampleYaw
detect1
detect2

11 This command generates a third dataset which is the result of subtracting dataset1 from dataset2 element by element.

12 Plots the old data, along with the new dataset, which is the deviation of one detector from the other.

13-20 These lines perform some statistical analysis on the data inside the dataset, such as the mean value, the RMS, skew and Kurtosis.

8.2.3 DataSet Full Method Listing

Table6.7.ScanFileHolder Method Listing Method Inputs Outputs Description Constructor DataSet Makes a copy of the dataset given as the input Constructor double[] Generates a 1D dataset containing the data provided as the input Constructor int[] Generates a Dataset with the dimensionality of the length of the input array, and size per dimension of the input array value for example DataSet(2,3,4) would create a 3D dataset with sizes 2 in the first, 3 in the second and 4 in the third dimensions. Constructor int w, int h, double[] data Makes a new 2D dataset with height h, width w, and filled with the data. This is filled quickest along the width. Constructor int w, int h, int d, double[] data Makes a new 3D dataset with height h, width w, depth d and filled with the data. This is filled quickest along the width. then along the height. Constructor JAMA Matrix Creates a 2D dataset which has the same data and proportions of the input JAMA Matrix abs DataSet returns the Absolute values of each element of the dataset in a new Dataset which is returned centroid double returns the centroid value of the dataset, this is effectively the point along the dataset which is the centre of mass of all the values. chiSquared DataSet double This function compares the dataset element by element with the input dataset. The differences between them are then squared and summed, and this is the value that is returned. cos DataSet returns the cosine of every value in the dataset, as a new dataset sin DataSet returns the sine of every value in the dataset, as a new dataset exp DataSet returns the exponential of every value in the dataset, as a new dataset ln DataSet returns the natural log of every value in the dataset, as a new dataset log10 DataSet returns the log base 10 of every value in the dataset, as a new dataset pow double DataSet returns each value in the dataset raised to the power given to the function, as a new dataset norm DataSet returns a new dataset where all the values are normalised to between 0 and 1, and scaled appropriately lnnorm DataSet returns a new dataset where the natural log of all the values normalised to between 0 and 1, and scaled appropriately lognorm DataSet returns a new dataset where the log base 10 of all the values are normalised to between 0 and 1, and scaled appropriately max double returns the maximum value of the dataset. maxPos int returns the position of the maximum value of the dataset. min double returns the minimum value of the dataset. minPos int returns the position of the minimum value of the dataset. mean double returns the mean value of the dataset. rms double returns the Root Mean Squared value of the dataset. skew double returns the skew of the dataset. kurtosis double returns the kurtosis of the dataset. diff DataSet Calculates the differential of the dataset and returns it as a new dataset. This makes the assumption that all the points are 1.0 apart. diff int n DataSet Calculates the differential of the dataset and returns it as a new dataset. This makes the assumption that all the points are 1.0 apart. n is the number of points from each side that are taken as an average to reduce noise. diff DataSet DataSet Calculates the differential of the dataset and returns it as a new dataset. The input dataset is the x coordinates for the calculation. diff DataSet, int n DataSet Calculates the differential of the dataset and returns it as a new dataset. The input dataset is the x coordinates for the calculation. n is the number of points from each side that are taken as an average to reduce noise. disp Displays the datasets contents to the jython terminal doubleArray double[] Returns all the data in the dataset as a single double array doubleMatrix double[][] If the Dataset is 2D, this returns the data as an array of array of doubles getJamaMatrix double[][] If the Dataset is 2D, this returns the data as a JAMA Matrix get int[] double returns the data at the location specified by the input. set double value, int[] double sets the data at the location specified by the input with the input value. getDimensions int[] returns the dimensionality and size of the dataset as an array of integers

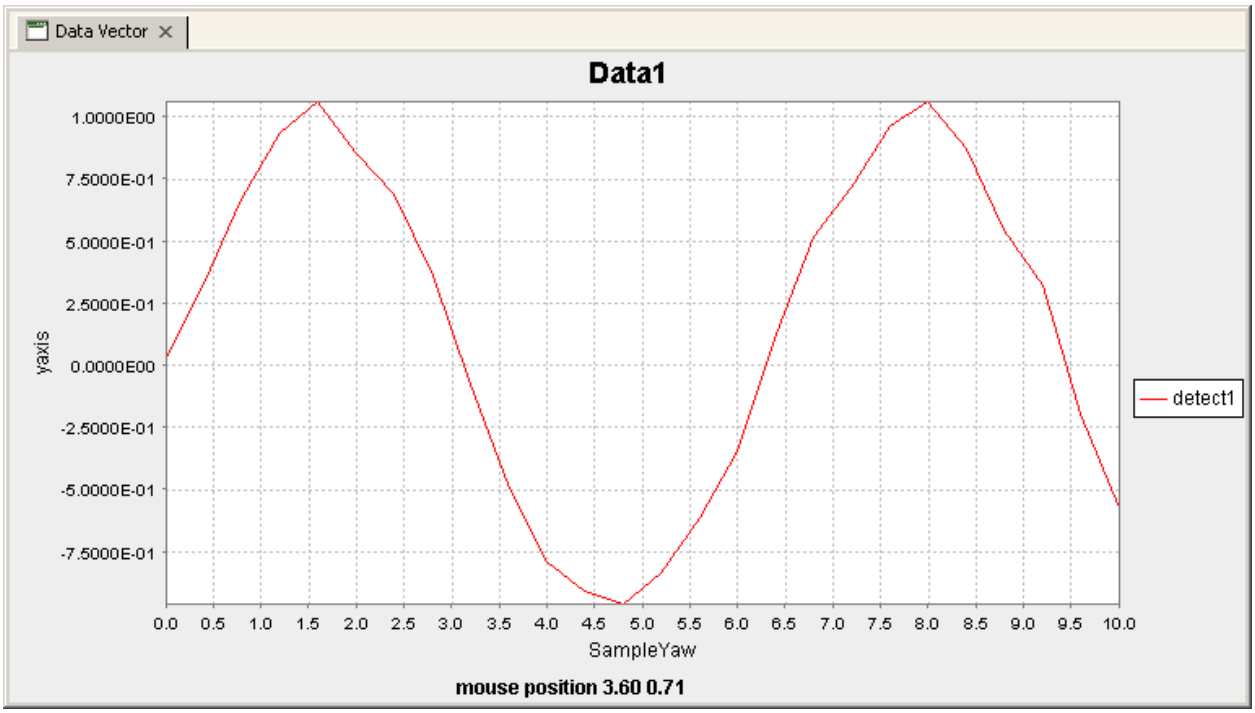
8.3 Peak Fitting

8.3.1 Basic Peak Fitting

A basic peak fitting routine in the Analysis toolkit is demonstrated in the following script.

```
>>>data = ScanFileHolder()
>>>data.loadSRS(1)
>>>data.plot(0,1)
>>>output = Fitter.plot(data[0],data[1]+1,GradientDescent(0.0001),[Gaussian(0.0,10.0,10.0,10.0),Gauss
>>>print output.disp()
7.883976029749007(0.0,10.0)
2.9464538192926537(0.0,10.0)
6.4733892793832455(-10.0,10.0)
1.56093364851906(0.0,10.0)
2.936788227366031(0.0,10.0)
6.420297623699959(-10.0,10.0)
>>>output[1].getValue()
2.9464538192926537
>>>output[1].getUpperLimit()
10.0
>>>output[1].getLowerLimit()
0.0
```

The images show below are of the base data, and then the fitted data. The green line represents the line of best fit through the function, with the other lines showing the individual functions which make up the fit. At the bottom of the plot there is a reference line, and some detail showing how far from the data the fit is. This provides a quick visual representation of the error in the fit.

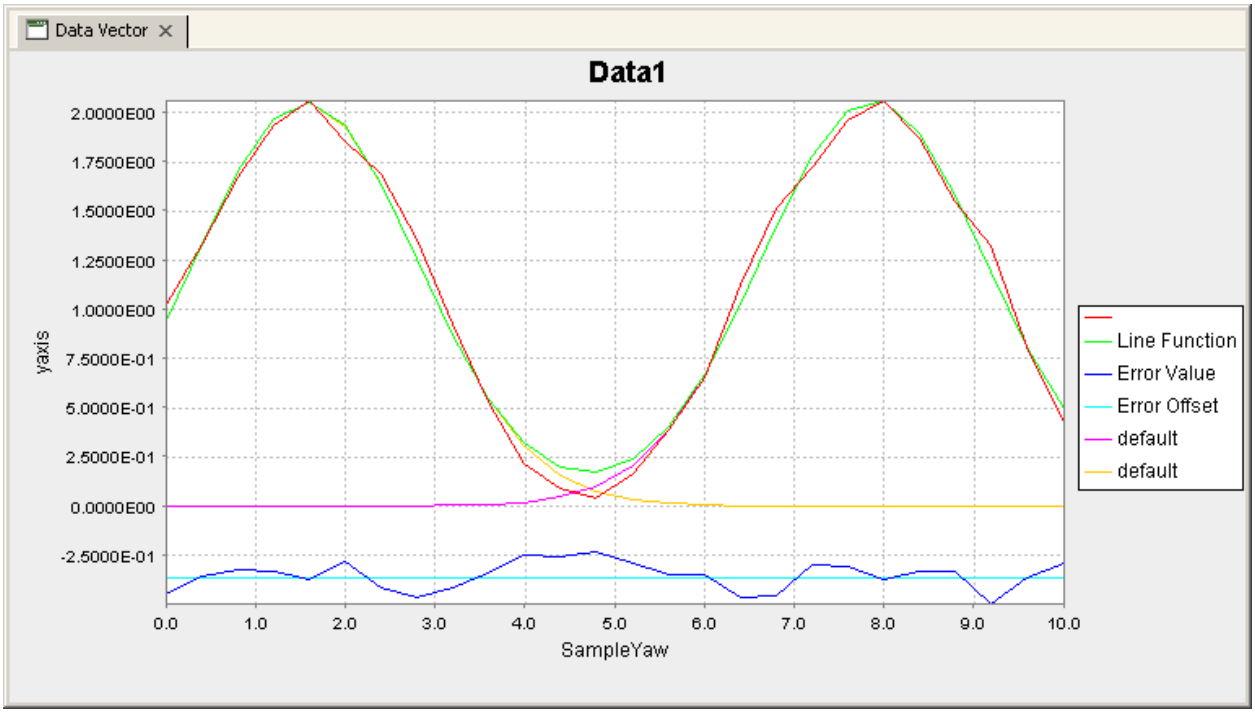


01 creates the object which will hold all the data we want to manipulate and visualise.

02 Loads in the scan that has the identifier "1.dat".

03 Plots the information that we want to fit the data against.

04 This is the command that performs the fitting, it is made up of several parts. This returns a set of values which represent each of the free parameters in the solution, and in this case is put into the object output. The first argument is the x data for the fit, and the second is the corresponding y data. The fit will be evaluated at every point specified by x values, and so if there are areas of dense data, they will be sampled more when the fitting



process is in action. The third entry is the optimiser that will be used for the process. The different optimisers are shown below. The final entry is a list of functions that will be fitted against. These are all added together to produce the final function.

05-11 This command, and the associated output shows the values after fitting for all the variable parameters in the fit. Each parameter is shown in turn and is followed by the bounds between which it can fall. These are in the same order that the items in the fit list were added in line 04. In this case, lines 06 to 08 are the first gaussians and lines 09-11 are the second. What the order of parameters means is displayed in the table later which describes the different line functions.

12-17 These commands show how individual pieces of data can be obtained from the object for use in other areas.

8.3.2 Fitting Methods

Table6.9.Fitting Methods Available Method Description MonteCarlo This method uses monte carlo methods to solve the problem. The Value entered when creating this is currently abstract, a smaller number is more accurate, and a larger number is quicker. 0.001 is a good start point for this. GradientDescent This method uses the gradient Descent method to optimise the problem. The differential of the configuration space is taken, and the direction found is stepped along by a certain amount. If the distance stepped causes the objective function to increase (a negative result) then the distance is reduced until a positive result is obtained. When the distance stepped to obtain a positive result is reduced to less than the argument value, the search ends. GeneticAlg This method uses differential evolution genetic algorithms to perform the search. This creates a group of agents which each perform a search over the configuration space to try to find good solutions. This is a slightly slower method, but performs much better on harder solutions. The value entered here is the stop criteria. When the objective function average of all the agents, is less than this value from the minimum agents objective function calculation, then the search will stop.

8.3.3 Fitting Functions

Table6.10.Fitting Functions Available Function Arguments Function Parameter Outputs Cubic minA, maxA, minB, maxB, minC, maxC, minD, maxD y=Ax^3+Bx^2+Cx+D A, B, C, D Gaussian MinPosition, MaxPosition, FWHM,

Area Position, FWHM, Area Lorentzian MinPosition, MaxPosition, maxFWHM, maxArea Position, FWHM, Area Offset minA, maxA y=A A PseudoVoigt minPos, maxPos, maxFWHM, maxArea Position, FWHM, Area Quadratic minA, maxA, minB, maxB, minC, maxC y=Ax^2+Bx+C A, B, C StraightLine minA, maxA, minB, maxB y=Ax+B A, B

CHAPTER

NINE

NEXUS DATA FILES

The DASC group are in the process of implementing the writing of NeXus files across all beamlines. For MX beamlines this will be as an archive format.

9.1 Overview

The structure of NeXus files is extremely flexible, allowing the storage both of simple data sets, e.g., a single data array and its axes, and also of highly complex data, e.g., the simulation results of an entire multi-component instrument. This flexibility is achieved through a hierarchical structure, with related data items collected together into groups, making NeXus files easy to navigate, even without any documentation. NeXus files are self-describing, and should be easy to understand, at least by those familiar with the experimental technique.

9.2 Using NeXus at Diamond.

9.2.1 Configuring the GDA to produce NeXus.

In order to configure the GDA to produce NeXus files there are a couple of configuration settings that will need to be set. Firstly, in the java.properties file ensure that the data writer's format is set to NexusDataWriter, i.e:

```
gda.data.scan.datawriter.dataFormat=NexusDataWriter
```

Next, we need to make sure that the GDA can find the relevant NeXus libraries that are required. So for any process that need to access NeXus files (this will normally only be the Command Server) we need to add the following argument to the startup command:

```
-Djava.library.path=$GDA_ROOT/lib/Linux-i386
```

where *GDA_ROOT* is defined to be wherever the GDA is installed. We are also assuming that we are running on a 32-bit Linux machine.

Finally, we need to make sure that the *LD_LIBRARY_PATH* also contains the library path (e.g. *$GDA_ROOT/lib/Linux-i386*). For the case of Microsoft Windows, we just need to make sure that the library directory (e.g. *%GDA_ROOT%/lib/win32*)is in the system path.

Important: If you use the standard GDA startup scripts then these modifications are already in place and all you will need to set is the java property.

9.3 relevant Java Properties

Below is a summary of the java properties that can be defined in order to configure some aspects of the DataWriter's behaviour.

Table 9.1: Nexus Java properties

Java Property name	default	possible	Description
gda.nexus.backend	HDF5	HDF5, XML	Which on disk format to use.
gda.nexus.beamlinePrefix	true	true, false	Whether to use the "iXX-" prefix for data files.
gda.nexus.createSRS	true	true, false	Whether to create an ASCII data file as well.

9.4 Useful Java Interfaces

If a scannable implements the `INeXusInfoWriteable` interface then it will have the ability to write additional information into the NeXus file within that scannables NeXus group which will usually be either a NXpositioner (scannables) or a NXdetector (detectors).

As an example here is the code from the `ScannableMotionBase` device:

```
public void writeNeXusInformation(GdaNexusFile file) {
    try {
        NeXusUtils.writeNexusDoubleArray(file, "soft_limit_min", getUpperGdaLimits());
        NeXusUtils.writeNexusDoubleArray(file, "soft_limit_max", getLowerGdaLimits());
        NeXusUtils.writeNexusDoubleArray(file, "tolerance", getTolerance());
        NeXusUtils.writeNexusInteger(file, "gda_level", getLevel());  // Non-official field
    } catch (NexusException e) {
        logger.debug("ScannableMotionBase: Problem writing additional info to NeXus file.");
    }
}
```

Also, here is an example from `CounterTimerBase`:

```
public void writeNeXusInformation(GdaNexusFile file) {
    try {
        NeXusUtils.writeNexusString(file, "description", getDescription());
        NeXusUtils.writeNexusString(file, "type", getDetectorType());
        NeXusUtils.writeNexusString(file, "id", getDetectorID());
    } catch (NexusException e) {
        logger.debug("CounterTimerBase: Problem writing additional info to NeXus file.");
    } catch (DeviceException e) {
        logger.debug("CounterTimerBase: Problem writing additional info to NeXus file.");
    }
}
```

You can see by implementing this interface, you are able to write also any information you want into the NeXus file.

9.5 Using NeXus Outside a Scan.

At the moment, the NeXus writing is mainly tied into the scanning mechanism. This is were most of the testing has taken place. It is possible to use the NexusDataWriter within a Jython script but this is considered 'experimental' at the moment.

9.6 Capturing Additional Information

9.6.1 Metadata

There are a number of items that will be captured by default on all beamlines. These are mainly items relating to the machine (source). All the metadata is read out using the GdaMetadata Class.

Basically, in order to record a particular value then you need to define a MetadataEntry with a specific name inside your server xml file. These can use any of the supported sources to actually get the data, e.g. scannable, EPICS pv, java property, ICAT value, etc...

At the moment the values are only captured at the start of a scan, this will be extended to allow for capture at various 'trigger' points, such as:

- Start of a scan.
- End of a scan.
- At both the start and end of a scan.
- At every scan point.
- At a specified time period (long term development)

9.6.2 Metadata items to NeXus Class Mapping

This section lists the mapping between various items within the NeXus classes and what metadata item needs to be called in order to populate them.

Table 9.2: NXsource

Field Name	Metadata Name	Datatype	Description
name	facility.name	String	Name
type	instrument.source.type	String	Facility type (e.g. "Synchrotron X-ray Source")
probe	instrument.source.probe	String	Radiation type (e.g. x-ray, IR, etc...)
mode	source.fillMode	String	synchrotron operating mode
facil-ity_mode	facility.mode	String	Facility running mode (e.g. "User", "Machine Day", etc...)
notes	instrument.source.notes	String	MCR messages
frequency	instru-ment.source.frequency	Double	Synchrotron Frequency in Hz
voltage	instrument.source.energy	Double	Synchrotron Energy in GeV
power	instrument.source.power	Double	Power
current	instrument.source.current	Double	Current
top_up	instrument.source.top_up	Boolean	Is the synchrotron in top_up mode ?

Table 9.3: NXmonochromator

Field name	Metadata Name	Datatype	Description
name	instrument.monochromator.name	String	Name
wavelength	instrument.monochromator.wavelength	Double	wavelength
energy	instrument.monochromator.energy	Double	energy

Table 9.4: NXbending_magnet

Field name	Metadata Name	Datatype	Description
name	instrument.bending_magnet.name	String	Name
bending_radius	instrument.bending_magnet.bending_radius	Double	Bending Radius
spectrum	instrument.bending_magnet.spectrum	NXdata	Spectrum
critical_energy	instrument.bending_magnet.critical_energy	Double	Critical Energy

Table 9.5: NXinsertion_device

Field name	Metadata name	Datatype	Description
name	instrument.insertion_device.name	String	Name
type	instrument.insertion_device.type	String	"undulator" or "wiggler"
gap	instrument.insertion_device.gap	Double	Gap
taper	instrument.insertion_device.taper	Double	Taper
phase	instrument.insertion_device.phase	Double	Phase
poles	instrument.insertion_device.poles	Integer	Number of Poles
length	instrument.insertion_device.length	Double	Length of Device
power	instrument.insertion_device.power	Double	Total Power delivered by device.
energy	instrument.insertion_device.energy	Double	Energy of Peak
bandwidth	instrument.insertion_device.bandwidth	Double	Bandwidth of Peak Energy
spectrum	instrument.insertion_device.spectrum	NXdata	Spectrum of insertion device.
harmonic	instrument.insertion_device.harmonic	Integer	Harmonic of Peak.

9.6.3 Writing arbitrary metadata to Nexus files

There are cases where it's useful to include arbitrary notes or fields in data files. To enable this, metadata fields can be specified along with a path for where in the file it should be written.

For instance to include a sample name field in the sample NXsample node, include a bean similar to the following in the Spring GDA configuration:

```
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="staticMethod" value="gda.data.scan.datawriter.NexusDataWriter.setMetadata"/>
    <property name="arguments">
        <map>
            <entry key="sample_name" value="sample:NXsample/name"/>
        </map>
    </property>
</bean>
```

This will include a name field in the sample group with the metadata value (read at the start of the scan) for the entry with the name *sample_name*. (NB metadata name does not have to match the entry in the nexus file)

The metadata to be included can be changed at run time via the Jython console:

```
>>> from gda.data.scan.datawriter import NexusDataWriter
>>> NexusDataWriter.updateMetadata({'xyz': 'sample:NXsample/abc'})
>>> # Scans from here will include xyz metadata
```

```
>>> NexusDataWriter.removeMetadata(['xyz'])
>>> # xyz will no longer be written to scan files
```

9.7 GDA Helper functions

With the *gda.data.nexus.NeXusUtils* class there are a number of static helper functions which make writing to a NeXus file a but less painful. Below there is a list of the most useful (but not all) functions.

openNeXusFile()
: Opens a NeXus file (Read-Write) and returns the file handlen.

openNeXusFileReadOnly()
: Opens a NeXus file (Read-Only) and returns the file handle.

writeNexusDouble()
: Writes an double into a field with a specified name at the current file position.

writeNexusDoubleArray()
: Writes an array of doubles into a field with a specified name at the current file position.

writeNexusInteger()
: Writes an integer into a field with a specified name at the current file position.

writeNexusIntegerArray()
: Writes an array of integers into a field with a specified name at the current file position.

writeNexusLong()
: Writes a long into a field with a specified name at the current file position.

writeNexusLongArray()
: Writes an array of longs into a field with a specified name at the current file position.

writeNexusString()
: Writes a String into a field with a specified name at the current file position.

writeNexusBoolean()
: Writes a boolean into a field with a specified name at the current file position.

CHAPTER

TEN

ECLIPSE GUI PREFERENCES

This document lists the various preferences defined by the GDA Eclipse Client

The common preferences are defined in the class uk.ac.gda.preferences.PreferenceConstants.

10.1 How to Specify Preferences

The file system location of a properties file containing default settings for plug-in preferences can be set in the following ways:

- use -Declipse.pluginCustomization=<path> as a VM argument to the Java VM
- set the property, -pluginCustomization, in the config.ini file in the appropriate configuration area

10.2 ScanPlot Preferences

- uk.ac.gda.client/gda.client.plot.colors

  Comma separated list of integers used to construct Color(int rgb). Def = PlotColorUtility.getDefaultColour(nr);

  Values converted to Integer using Integer.valueof(s,16)., i.e. using radix 16 e.g. FF0000 = red

```
e.g. uk.ac.gda.client/gda.client.plot.colors=FF0000,FF00,FF
```

- uk.ac.gda.client/gda.client.plot.linewidth

  Integer value for line width. Def = PlotColorUtility.getDefaultLineWidth(0):

```
e.g. uk.ac.gda.client/gda.client.plot.linewidth=1
```

- uk.ac.gda.client/gda.client.plot.linestyles

  Comma separated list of integers for line style. Def = PlotColorUtility.getDefaultStyle(nr):

```
e.g uk.ac.gda.client/gda.client.plot.linestyles=3,3,3,3,4,4,4,4
```

10.3 Preference to display text beside icons in GDA view

There is a preference to display text next to the relevant icons in the GDA views. To do so

- Click on the menu options Window -> Preferences
- GDA preferences will be displayed

- Select the checkbox that says - “Show text along with icons for menus in the toolbar”
- Click “OK” (GDA will prompt to restart itself - this is so that GDA can apply the relevant changes)

CHAPTER

ELEVEN

CONTRIBUTORS TO THE GDA PROJECT

The Generic Data Acquisition (GDA) software was initially developed at SRS Daresbury. In 2003 it was adopted by Diamond Light Source, who took over as the principal developer.

This section lists people who have contributed in some way (code, design, documentation) to the GDA project. Names are ordered alphabetically by surname.

Contributors to the most recent release

Mark Basham

Kristian Benning

Jonathan Blakes

Mark Booth

Peter Chang

Matthew Dickie

Silvia da Graca Ramos

Baha El Kassaby

Richard Fearn

Jacob Filik

Matthew Gerring

Markus Gerstel

Iain Hall

Paul Hathaway

Peter Holloway

Anthony Hull

Karl Levik

Charles Mita

James Mudd

Colin Palmer

Keith Ralphs

Chris Sharpe

Rob Walton

Kaz Wanelik

Michael Wharmby

Matthew Webber

Fajin Yuan, Past contributors

Jun Aishima

Alun Ashton

Simon Berriman

Oliver Buckley

Stuart Campbell

Josephine Chan

Chris Coles

Joachim Diepstraten

Paul Gibbons

Jonah Graham

Michael Kleyn

Phyo Kyaw

Graham Lee

Conor Lehane

Geoff Mant

Tracy Miranda

Vasanthi Nagalingam

William Newell

Bill Pulford

Xiaoxu Ren (Eric)

Deepa Rethinam

Tobias Richter

Irakli Sikharulidze

Duncan Sneddon

Ravi Somayaji

Richard Tyler

Richard Woolliscroft.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search