
GDA Developer Guide

Release 8.6

Jun Aishima
Mark Basham
Peter Chang
Joachim Diepstraten
Richard Fearn
Matthew Gerring
Paul Gibbons
Karl Levik
Geoff Mant
Vasanthi Nagalingam
Bill Pulford
Eric Ren
Tobias Richter
Duncan Sneddon
Robert Walton
Matthew Webber
Richard Woolliscroft
Fajin Yuan

July 01, 2010

CONTENTS

1	Introduction to the GDA	3
1.1	Example usage of the main Jython GDA commands	3
1.2	Example devices	4
1.3	Using the plotting functions in GDA	9
2	Writing new Device classes in Jython and Java	21
2.1	Introduction	21
2.2	The Scannable interface and ScannableBase classes	21
2.3	Description of the Scannable properties and relations between them	22
2.4	Add a new device to the server	23
2.5	Examples of other Scannable classes and tests in GDA	24
2.6	Demonstrate use of Scannable in terminal	25
3	Client GUI development	27
3.1	Introduction	27
3.2	Writing the Swing GUI component	27
3.3	Adding the new component to the GDA client	28
3.4	CORBAising the object	28
4	Remoting	31
4.1	CORBA in the GDA	31
4.2	Alternatives to CORBA	34
5	GDA configuration	37
5.1	Spring configuration	37
5.2	Logging	40
5.3	Metadata	41
6	GDA Demo	43
6.1	Basic commands	43
6.2	Other scannables	43
6.3	Default detectors	43
6.4	Beam focusing	44
7	Indices and tables	45

Contents:

INTRODUCTION TO THE GDA

For general background and detailed description of GDA, first read the ‘GDA Users’ Manual’ (in documentation/docs). This includes: an introduction to GDA, the GDA scripting environment, a description of scanning (data acquisition), advanced plotting techniques, and data analysis techniques.

A more detailed description of developing new components in GDA in Jython is provided in the ‘GDA Jython training course’ (in documentation/docs). This has practical examples of developing new components in Jython, scanning them, and plotting and analysing their output.

Here, we describe further examples of user-defined scannables in Jython, and scanning and manipulating them from the Jython terminal.

1.1 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.

1.2 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:

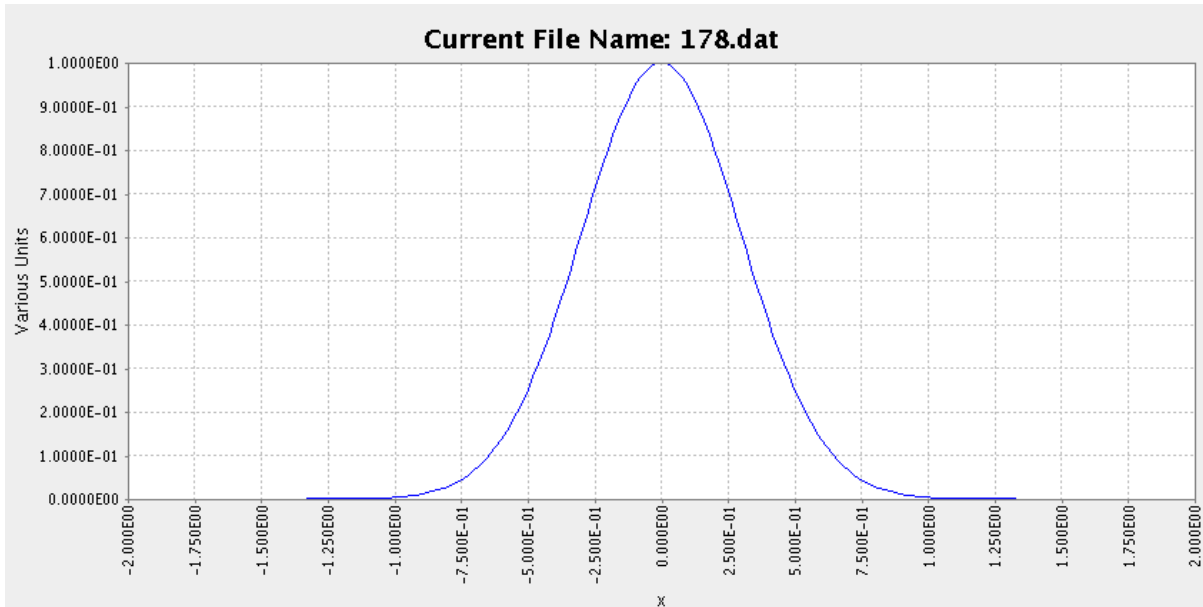
```
>>> help ScannableGaussian
>>> help ScannableSine
```

Make an instance of ScannableGaussian:

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

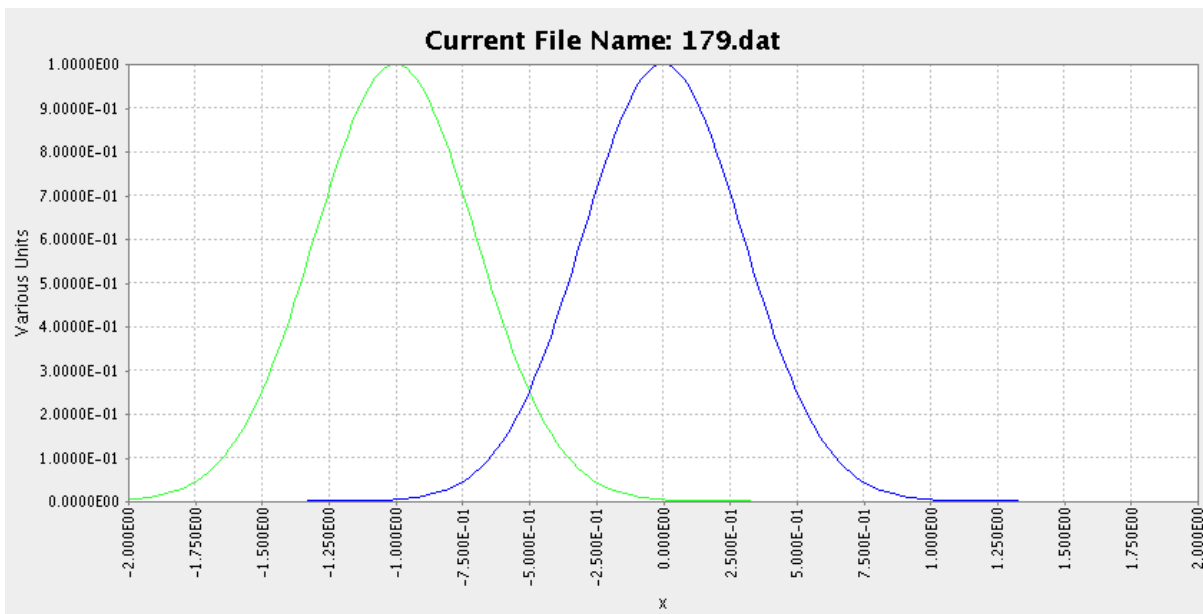
Scan it from -2 to 2 in steps of 0.02:

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



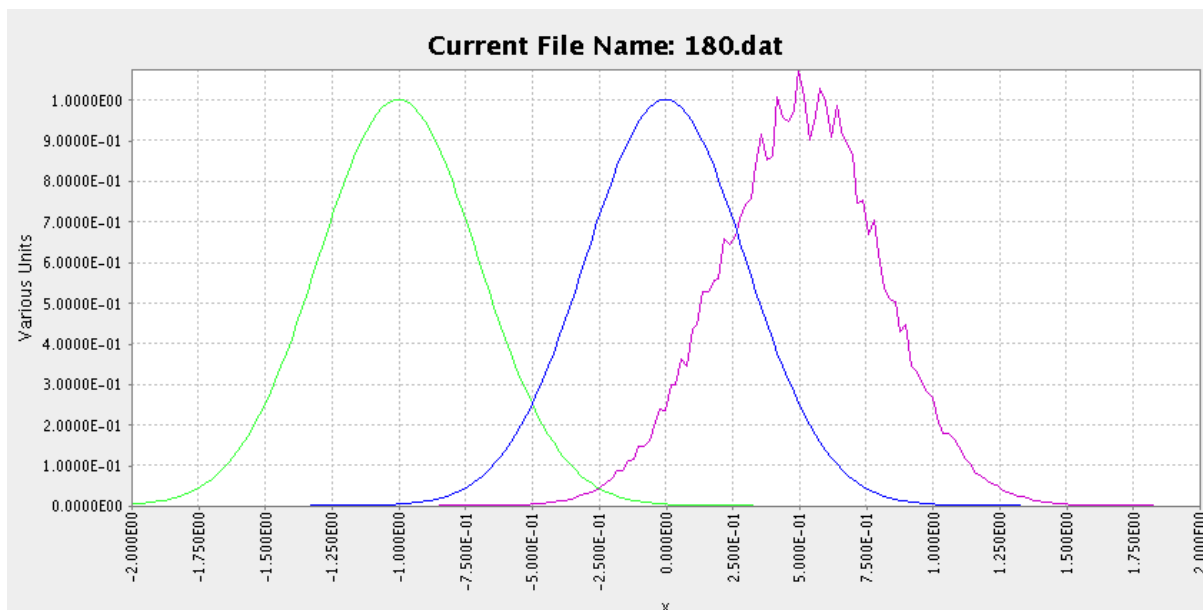
Change its centre to -1 and rescan:

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



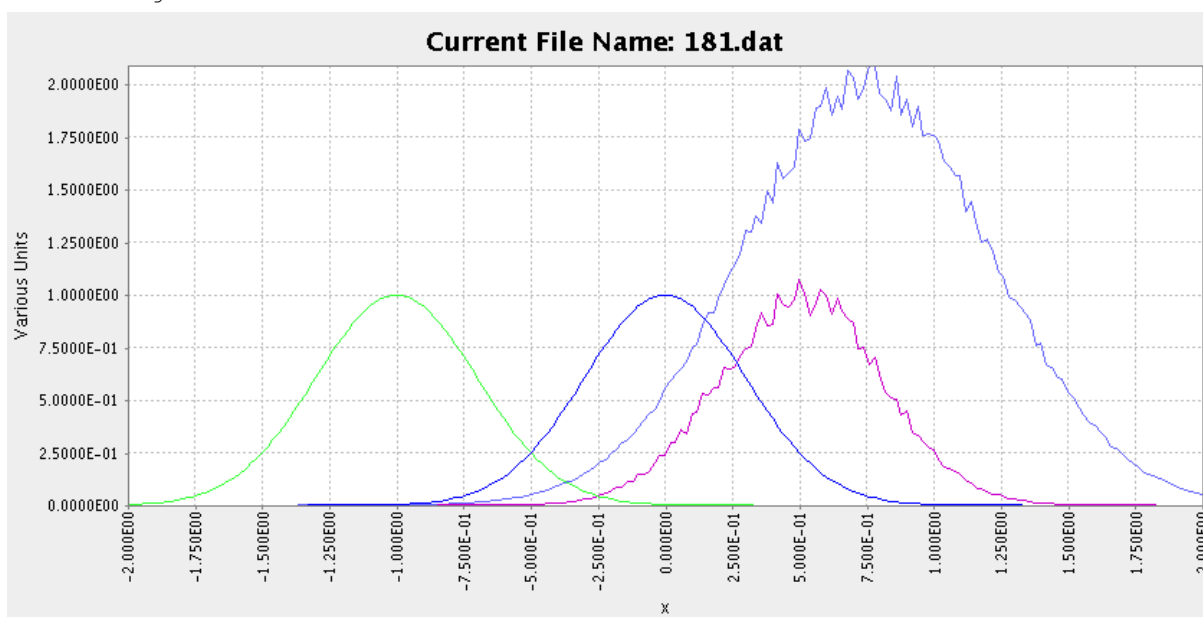
Move again, add some noise, and rescan:

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



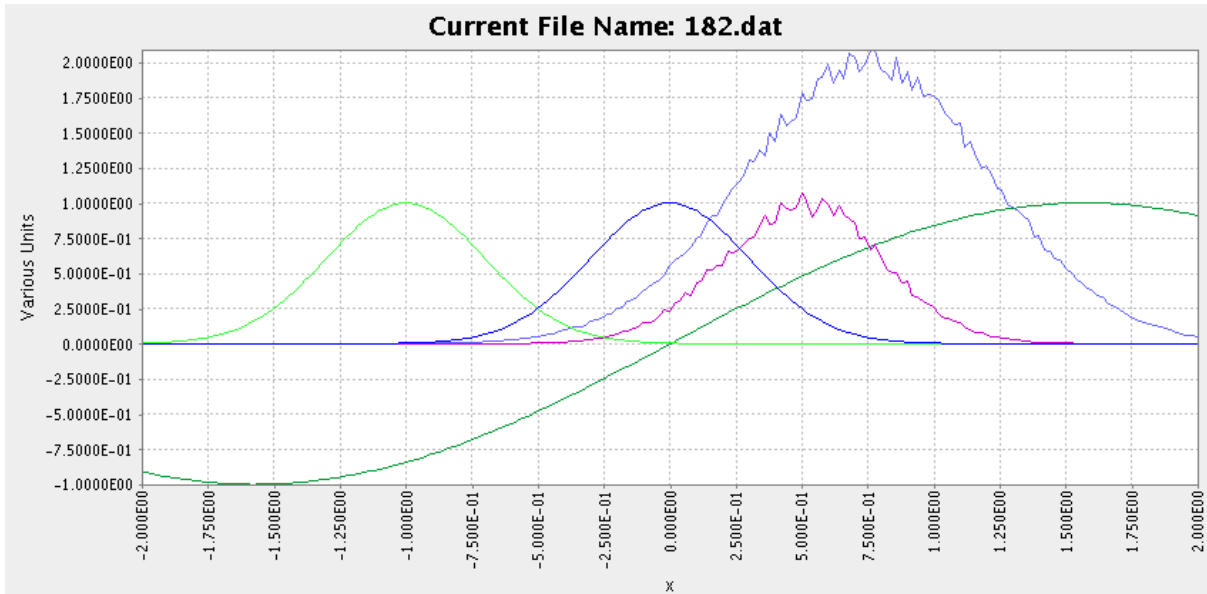
Make a new instance of ScannableGaussian, setting values for its additional optional properties, and scan it:

```
>>> 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
```



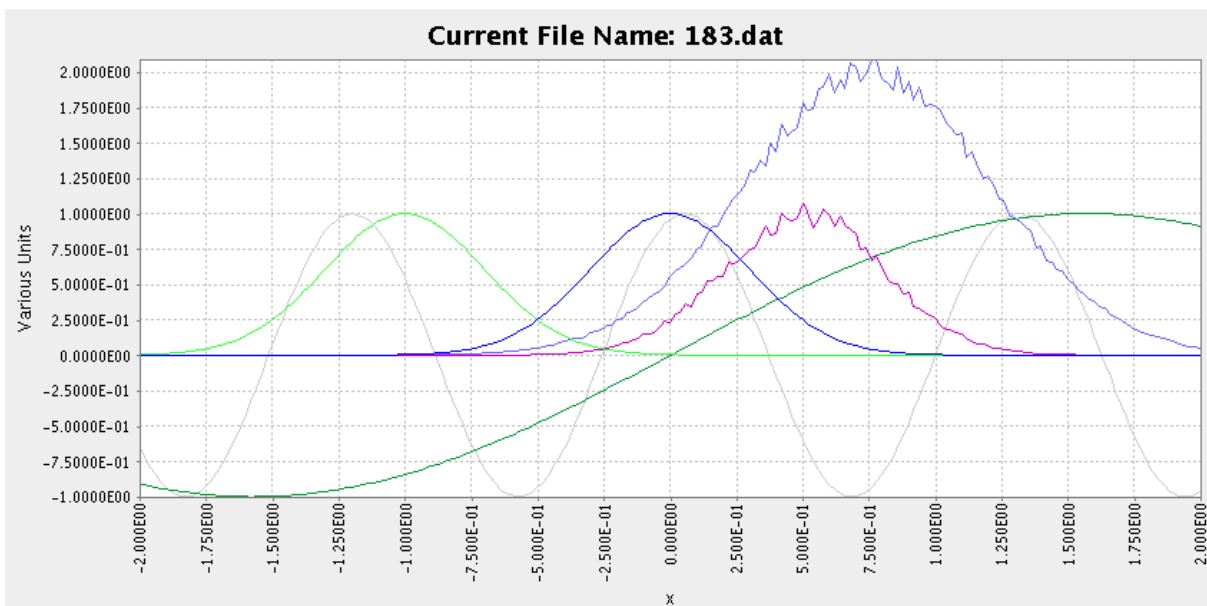
Make an instance of a ScannableSine class and scan it:

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



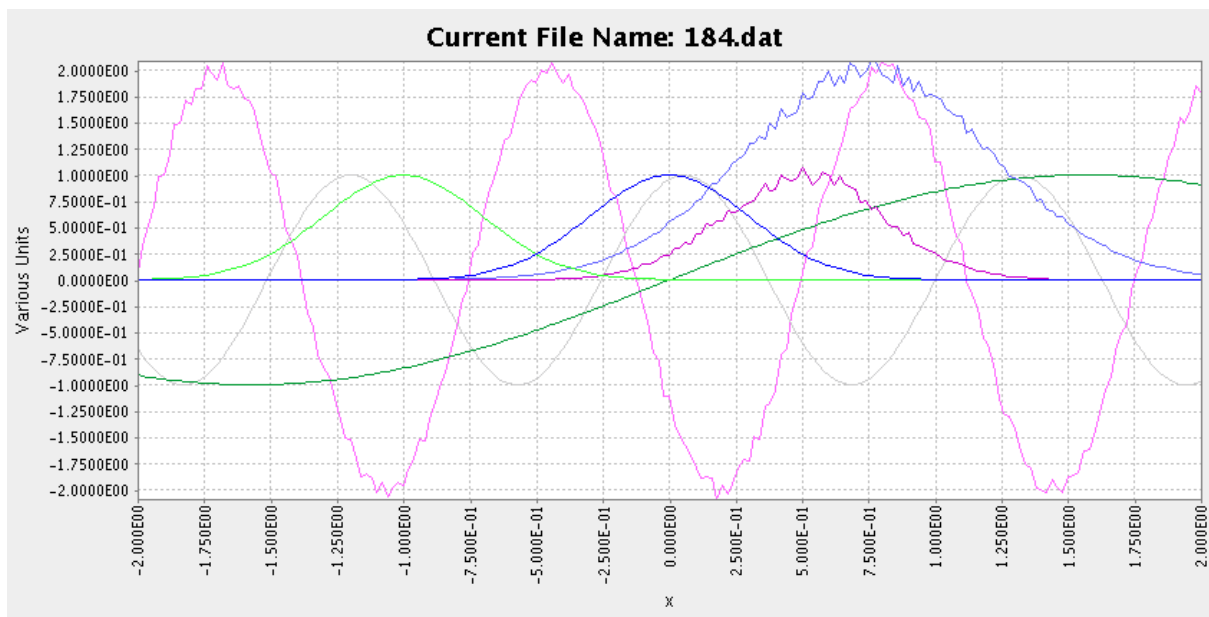
Change the period and phase of ss and rescan:

```
>>> 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 `documentation/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.

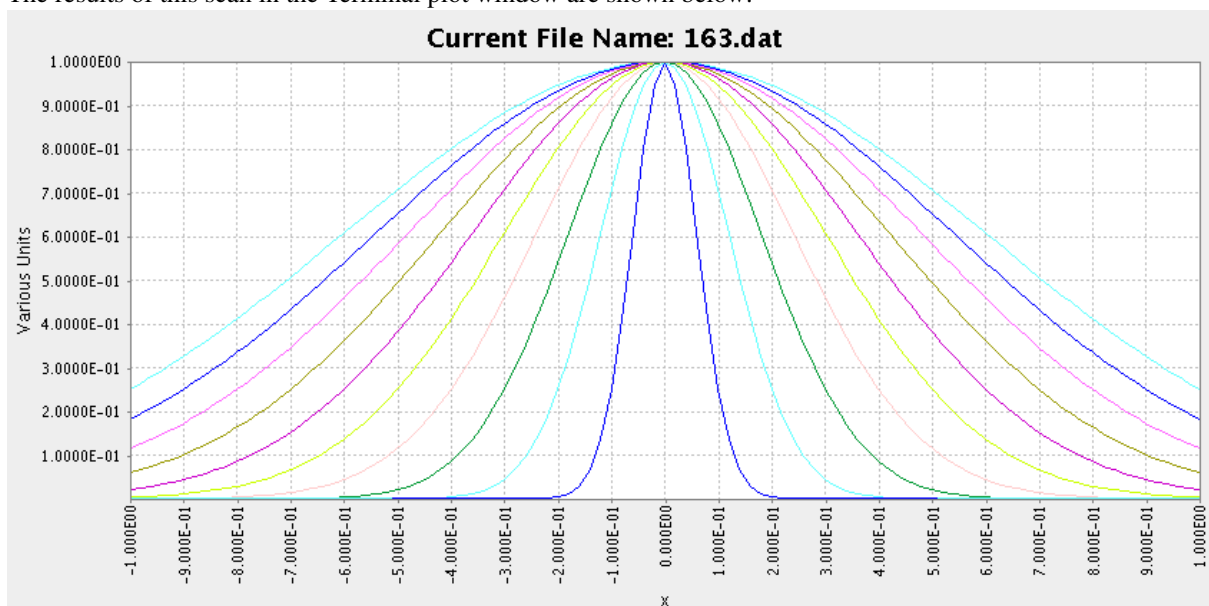
Instantiate a new `ScannableGaussianWidth` object:

```
>>> sgw = ScannableGaussianWidth('sgw', 0.0, scannableGaussian0)
```

Perform the nested scan:

```
>>> scan sgw 1. 10. 1 scannableGaussian0 0. 100. 10
```

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



1.3 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. For a detailed description of advanced plotting, refer to ‘Chapter 6. Plotting’ in the GDA Users’ Manual, and Section 6 ‘Data analysis and visualisation’ in the GDA Jython training course.

Here, we show a few basic plotting examples using the example Scannable classes in the module ‘scannable-Classes’.

Make new ScannableSine:

```
>>> del ss
>>> ss = ScannableSine('ss', 0.0, period=0.5)
```

Scan a scannable:

```
>>> scan ss -4.0 4.0 0.05
```

Make a new ScanFileHolder object (delete first, if already have ‘data’ object):

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

Read the last scan into the ScanFileHolder:

```
>>> data.loadSRS()
```

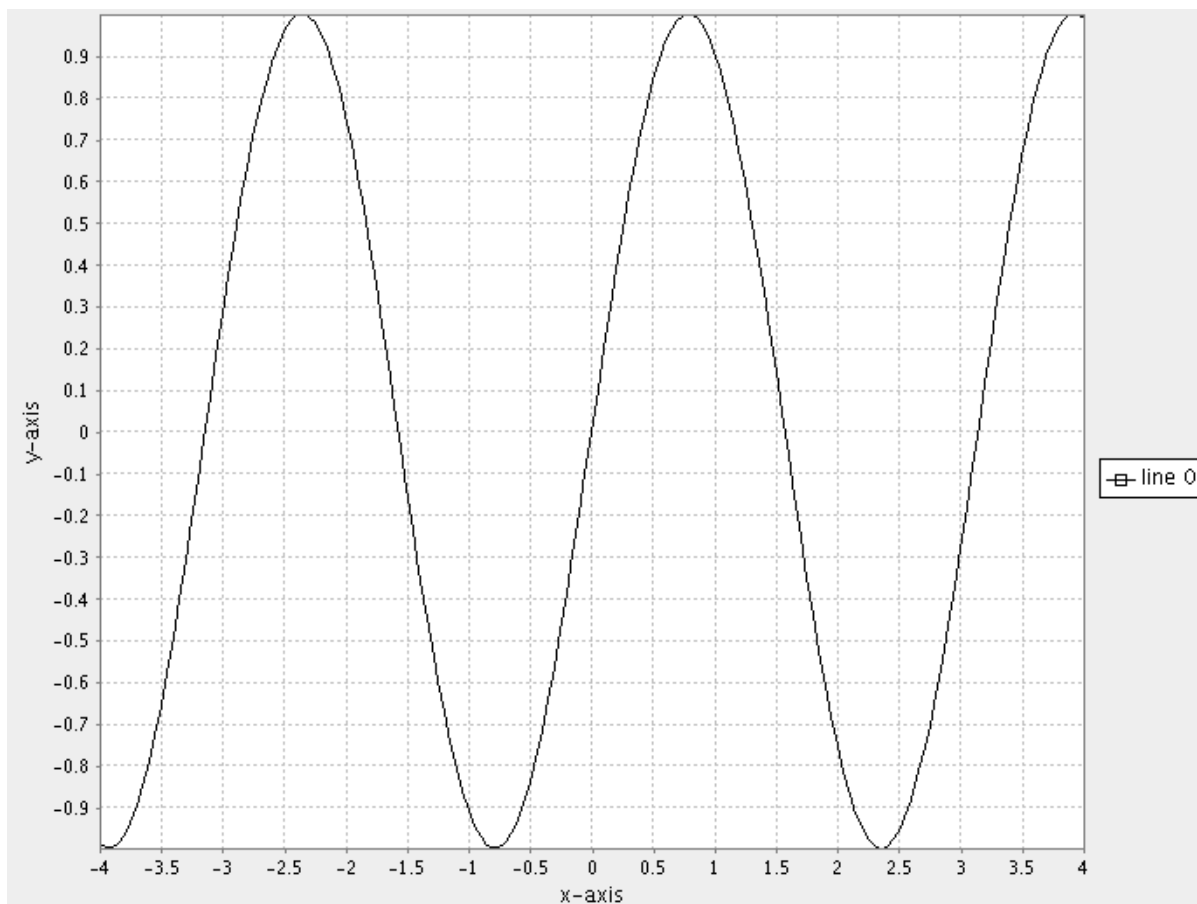
Print information about the scan:

```
>>> data.info()
```

Plot column 1 against column 0:

```
>>> data.plot(0,1)
```

The plot and associated functions are available in the ‘Data Vector’ view in the GDA client.



Individual columns of the scan can be accessed from the complete scan:

```
>>> dataset1 = data[1]
```

... and plotted against other data columns:

```
>>> Plotter.plot('Data Vector', data[0], dataset1)</nowiki>
```

Characteristics of individual data columns can be accessed using different functions:

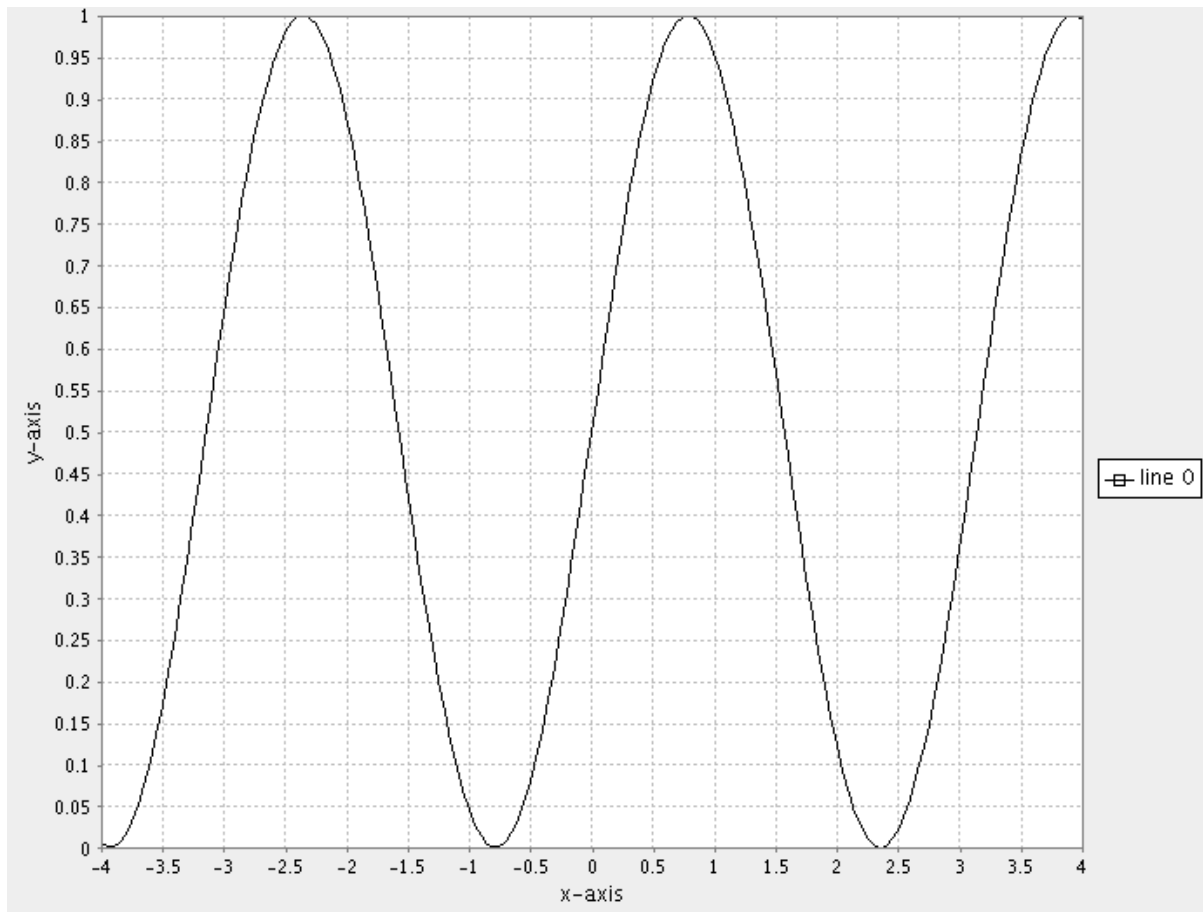
```
>>> dataset1.min()
-0.9999
>>> dataset1.max()
0.9999
```

Datasets can be transformed:

```
>>> dataset1 -= dataset1.min()
>>> dataset1.min()
0.0
>>> dataset1.max()
1.998
>>> dataset1 /= dataset1.max()
>>> dataset1.min()
0.0
>>> dataset1.max()
1.0
```

The two commands above result in the dataset being normalised to the range [0,1]. This is demonstrated by re-plotting the data:

```
>>> Plotter.plot('Data Vector', data[0], dataset1)
```



Generate double nested scan data:

Scan `scannableGaussian1` within a scan of `scannableGaussian0`. Both scans range from `-2.0` to `+2.0`, with a step of `0.2`. Both scans therefore consist individually of 11 data points:

```
>>> scan scannableGaussian0 -2.0 2.0 0.2 scannableGaussian1 -2.0 2.0 0.2
```

Read the scan into a `ScanFileHolder`, and print its information:

```
>>> data.loadSRS()
>>> data.info()
0 x
1 y
2 x
3 y
4 .....
```

Plot each scan independently:

```
>>> data.plot(data[0], data[1])
>>> data.plot(data[2], data[3])
```

Extract the signal ('y', i.e. Gaussian) from the first scan:

```
>>> test1 = data[1]
```

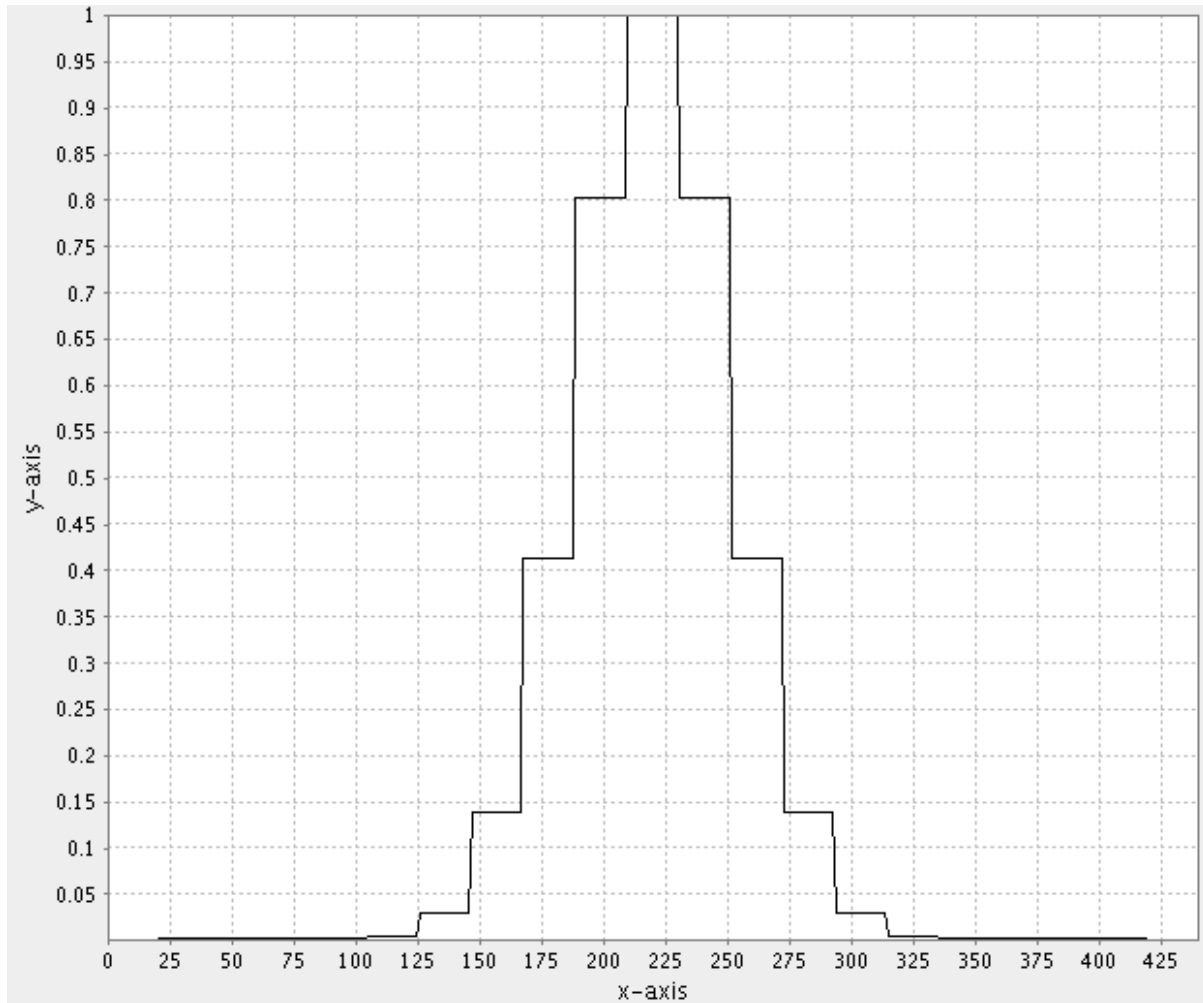
Resize the data to a square grid:

```
>>> test1.resize([21,21])
```

Note: The dimension arguments must correspond to the dimensions of the nested scan. Both the inner and our scans are from -2.0 to 2.0 in steps of 0.2; therefore each dimension consists of 21 points.

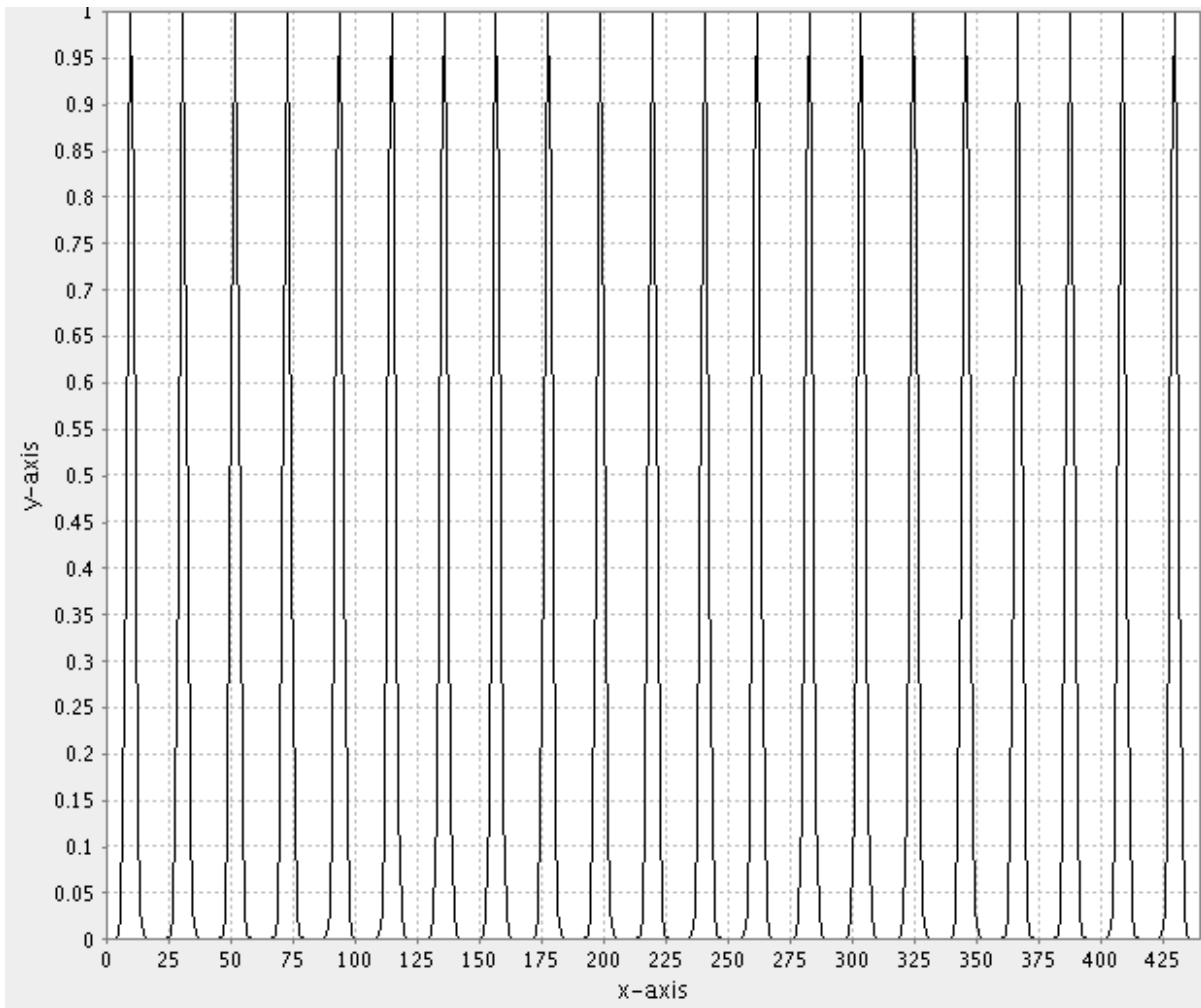
Plot the outer scan data over the range of the combined scan. This consists of 441 (21x21) points. The value of the outer scan increments every 21 points:

```
>>> data.plot(DataSet.arange(441), data[1])
```



Plot the inner scan data over the range of the combined scan. This consists of 441 (21x21) points. The inner scan data consist of 21 adjacent Gaussians:

```
>>> data.plot(DataSet.arange(441), data[3])
```



Plot an image:

```
>>> Plotter.plotImage('Data Vector', test1)
```



Do the same for the second scan:

```
>>> test2 = data[3]
>>> Plotter.plot('Data Vector', DataSet.arange(121), test2)
>>> test2.resize([11, 11]) </nowiki>
>>> Plotter.plot('Data Vector', DataSet.arange(121), test1)
>>> Plotter.plotImage('Data Vector', test2)
```



Plot the individual data sets over the complete scan range:

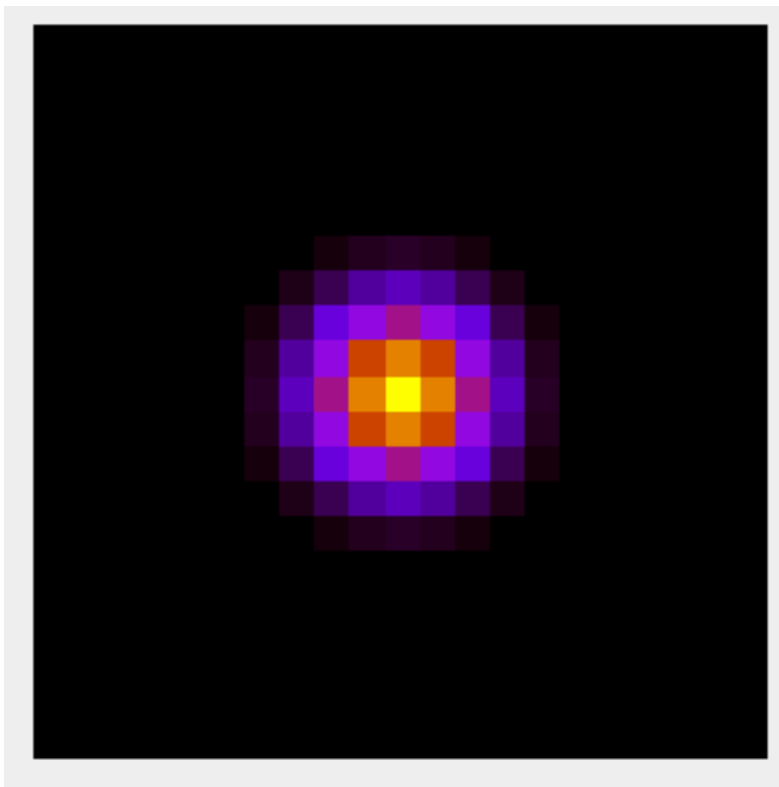
```
>>> data.plot(DataSet.arange(121), data[0])
>>> data.plot(DataSet.arange(121), data[1])
>>> data.plot(DataSet.arange(121), data[2])
>>> data.plot(DataSet.arange(121), data[3])
```

Alternative plotting command using Plotter:

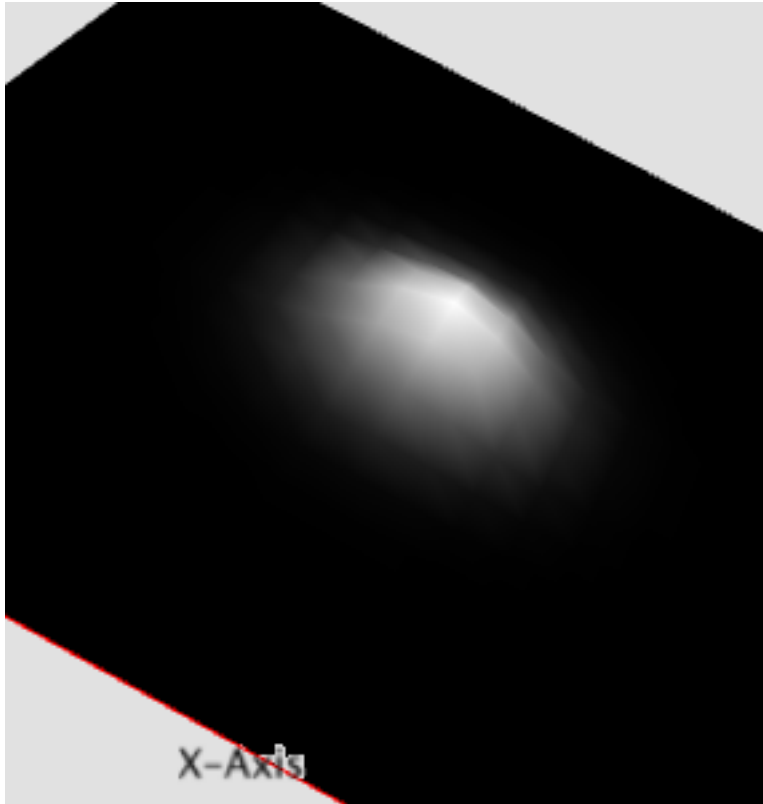
```
>>> Plotter.plot('Data Vector', DataSet.arange(121), data[0])
>>> Plotter.plot('Data Vector', DataSet.arange(441), data[1])
>>> Plotter.plot('Data Vector', DataSet.arange(121), data[0])
>>> Plotter.plot('Data Vector', DataSet.arange(441), data[3])
```

Make a new data set, the product of test1 and test2, and plot (resulting in a 2D Gaussian), both as flat image (heat map), and as (rotatable) 3D:

```
>>> test3 = test1 * test2
>>> Plotter.plotImage('Data Vector', test3)
```



```
>>> Plotter.plot3D('Data Vector', test3)
```

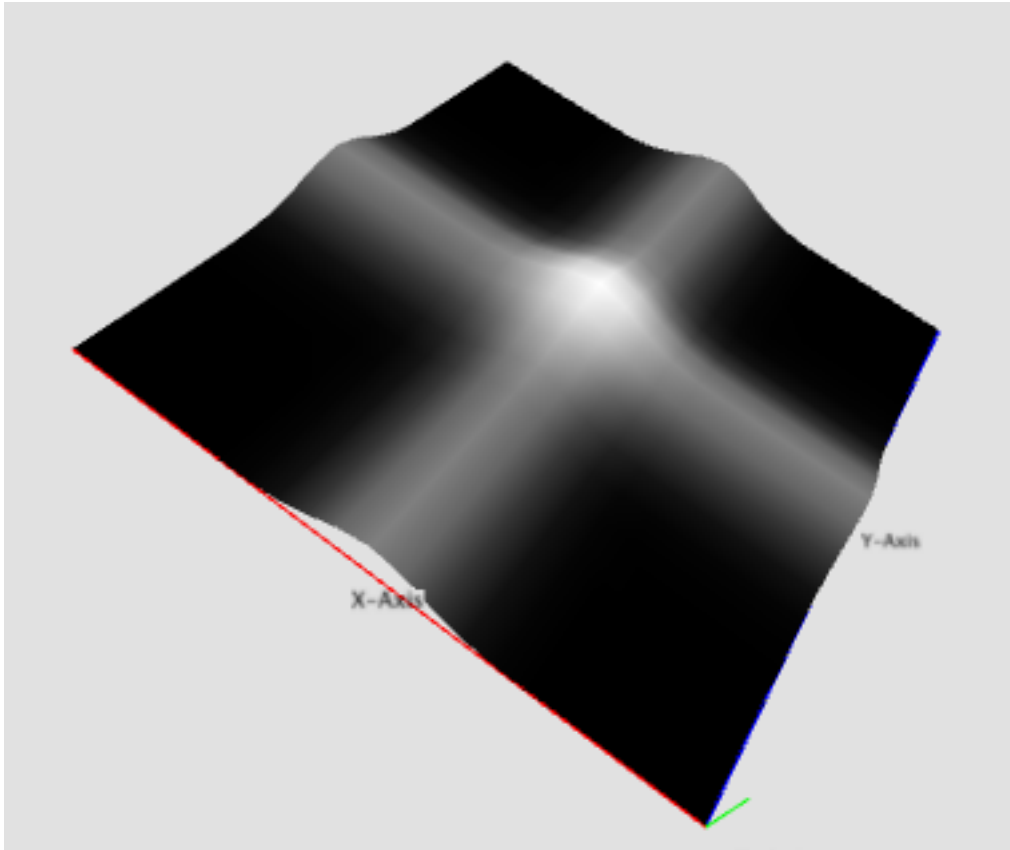


Make a different combination of data and plot:

```
>>> test4 = test1 + test2  
>>> Plotter.plotImage('Data Vector', test4)
```



```
>>> Plotter.plot3D('Data Vector', test4)
```

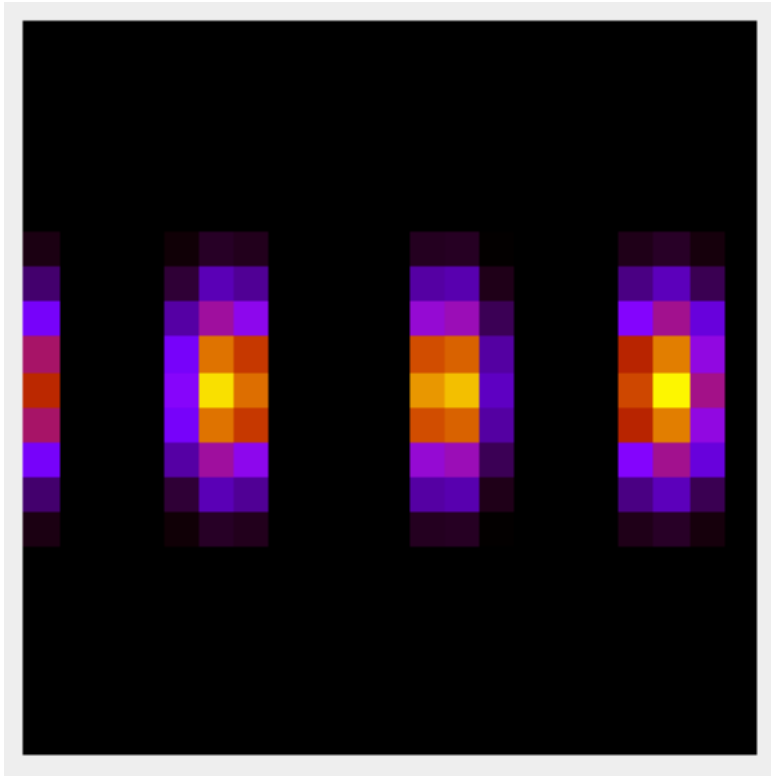


Plot the expanded / resized data sets:

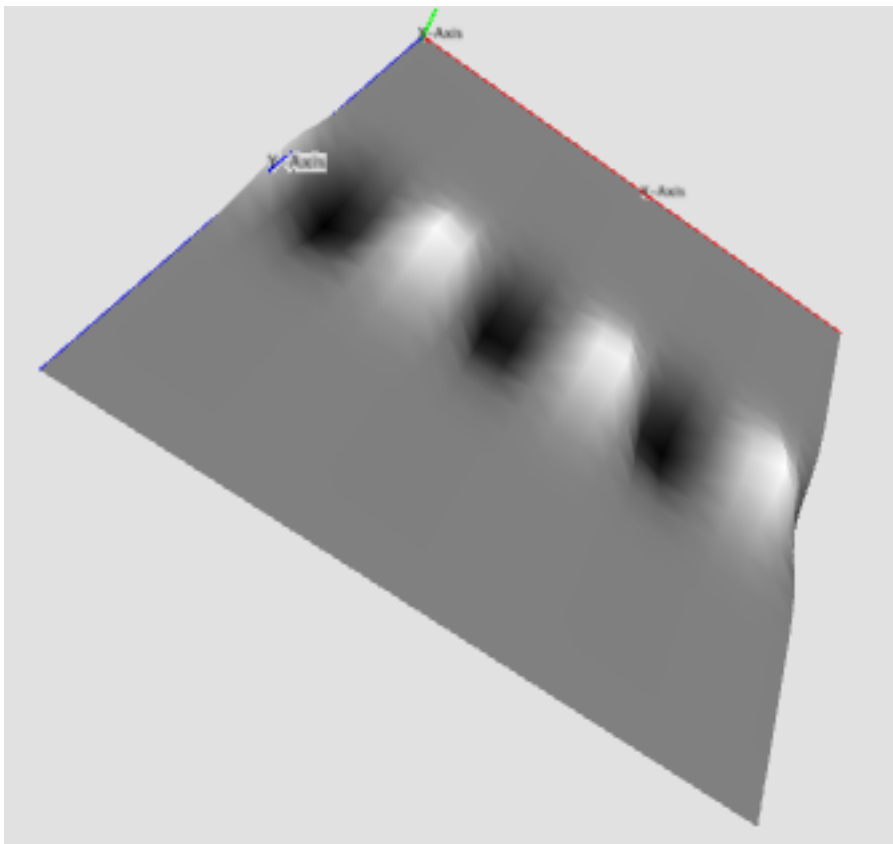
```
>>> Plotter.plot('Data Vector', DataSet.arange(121), data[1])
>>> Plotter.plot('Data Vector', DataSet.arange(121), data[3])
```

Do a nested scan of an outer Gaussian containing an inner sine:

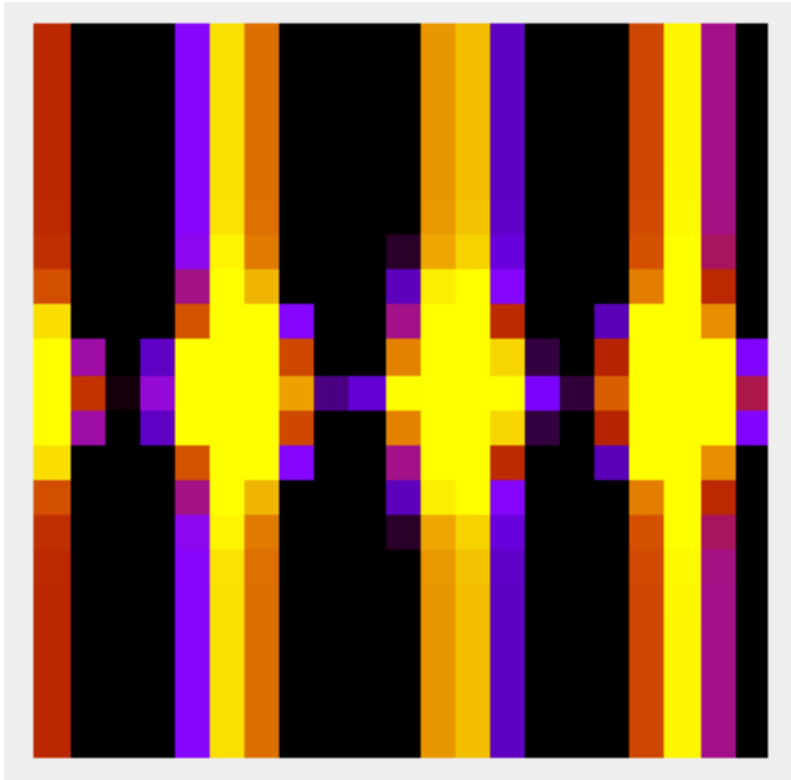
```
>>> scan scannableGaussian0 -2.0 2.0 0.2 scannableSine -2.0 2.0 0.2
>>> data.loadSRS()
>>> test1 = data[1]
>>> test2 = data[3]
>>> test1.resize([21,21])
>>> test2.resize([21,21])
>>> test3 = test1 * test2
>>> test4 = test1 + test2
>>> Plotter.plotImage('Data Vector', test3)
```



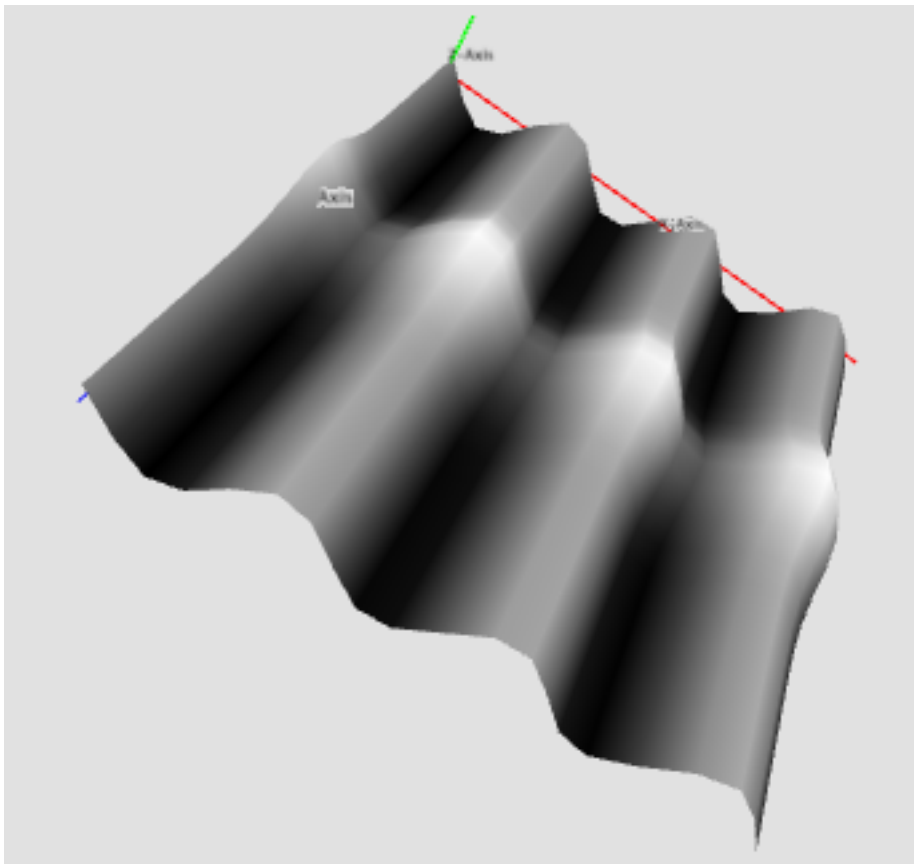
```
>>> Plotter.plot3D("Data Vector", test3)
```



```
>>> Plotter.plotImage("Data Vector", test4)
```



```
>>> Plotter.plot3D("Data Vector", test4)
```



WRITING NEW DEVICE CLASSES IN JYTHON AND JAVA

2.1 Introduction

New devices can be written using core classes in GDA. These can be written in either Jython or Java.

For both Jython and Java:

1. Define new devices in code
2. Load onto server (object server)

For Jython:

1. Define classes in Jython scripts that extend `PseudoDevice`
2. Load them into the object server by importing the Jython module, and make instances of the Jython-defined devices

For Java:

1. Write new devices in Java implementing different device interfaces. Here we illustrate by writing new Scannable devices
2. Import instances of the classes defined in Spring beans configuration files

To illustrate the process of developing new devices in Java, and incorporating them into GDA, we describe the process of developing several new devices that implement the Scannable interface. These devices are then included in the system by editing configuration files which are read by the server at startup. The devices can then be scanned and manipulated in GDA from the Jython terminal.

Developing software for new devices for GDA is a likely requirement at each site using GDA, to accommodate specific beamline components into the GDA software framework.

Users should first read “Chapter 5: Scanning” in the GDA Users manual for an introduction to the basic data acquisition techniques used in GDA. Below, we describe developing new classes which implement the Scannable interface. This will likely be required development at each site using GDA in order to accommodate specific beamline components into the GDA software framework.

2.2 The Scannable interface and ScannableBase classes

All Scannable classes implement the Scannable interface. A core base class implementing the Scannable interface is available in GDA as the class `gda.devive.scannable.ScannableBase`. New user-defined Scannable implementations should extend `ScannableBase`. Instances of these will then be visible in the GDA terminal after issuing the command ‘`ls Scannable`’.

The most important methods for a Scannable to implement are:

- getPosition()
- asynchronousMoveTo()
- isBusy()

Other fields in the Scannable that must be defined are:

- name
- initial position
- inputNames
- extraNames
- outputFormats
- units

A full description of the parameters available in a Scannable implementation is available in ‘Chapter 5: Scanning’ of the GDA Users Manual.

A test class that has static methods for constructing instances of several different types of ‘dummy’ or testable software Scannables is available in the documentation configuration src directory: org.myls.gda.device.scannable.ScannableClassGenerator. It has methods:

- generateScannableGaussian()
- generateScannableGaussian(Gaussian)
- generateScannableSine()
- generateScannableSine(SineWave)

This generator constructs instances of the two Scannable classes ScannableGaussian, and ScannableSine. These scannables classes differ in the value returned by getPosition(). For ScannableGaussian, the method returns the value of a Gaussian of the specified position, width and height 1, with additional noise if defined, at the specified x value:

```
@Override
public Object getPosition() throws DeviceException {

    // we assume the position is a double - it is only for testing
    double x = (Double) super.getPosition();
    double x2 = x - centre;
    double sigma = 0.425 * width // FWHM -> sd
    double noiseVal = height * (Math.random() * noise);
    double y = Math.exp(-(x2 * x2) / (sigma * sigma)) + noiseVal;
    return new Double[] { x, y };
}
```

2.3 Description of the Scannable properties and relations between them

(This material is derived from ‘Chapter 5: Scanning’ in the GDA Users’ manual; it is repeated here for convenience)

It is obligatory to set the values of several fields in the constructor of all Scannables. These obligatory fields are:

- name
- inputNames
- extraNames
- outputFormat

- `currentPosition`

The fields ‘`inputNames`’, ‘`outputNames`’, and ‘`outputFormat`’ together define what numbers this Scannable represents, what they are called, and the format for printing their values out to file or console.

The “‘`inputNames`’” array defines 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 empty (size 0) 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 `getRawPosition()` 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 is normally empty (size 0).

The “‘`outputFormat`’” array lists the formatting strings for the elements of both the `inputNames` and `extraNames` arrays. It is used when printing the output from the `rawGetPosition()` method to the console and logfiles.

Note: It is an absolute requirement that the length of the `outputFormat` array is the sum of the lengths of the `inputNames` and `outputNames` arrays for the Scannable to work properly.’”

2.4 Add a new device to the server

The new device is added to the server by defining it as a bean in a Spring beans configuration file. In the distribution, this file is ‘`server_beans.xml`’ in the ‘`xml`’ directory. This file can be consulted for the syntax used to define new object instances as beans in the Spring beans configuration file. The beans defined in this file are loaded into the object server at server startup, and can be accessed and manipulated by the GDA client.

Both getter and constructor dependency injection can be used. Each object on the server must have a ‘`name`’ property, which is its unique identifier in the server object namespace. As an example, we define several instances of the `ScannableGaussian` class using different bean definitions:

- `scannableGaussian0` — all properties set in the bean definition
- `scannableGaussian1` — only the properties of the Gaussian are set in the bean. Other properties such as input and extra names, and output formats are set to defaults in the Java constructor
- `scannableGaussian2` — the scannable is defined using a constructor argument which is a test Gaussian bean defined in the Spring configuration file. This demonstrates constructor dependency injection by Spring
- `scannableGaussian3` — no properties or constructor arguments are defined in the bean. The scannable is constructed using the default no argument constructor. All necessary properties are set to defaults in the Java class.

Similar examples are provided by several instances of the `scannableSine` class in the Spring configuration file:

- `scannableSine0` — the name and properties of the sine are set in the bean definition. Default values for other properties, such as input and extra names, and output formats, are defined in the Java class.
- `scannableSine1` — the properties of the sine are assigned to the object by a test sine bean defined in the bean configuration file (‘`testSineWave`’ bean)
- `scannableSine2` — no properties other than the name are defined in the bean definition. All other properties are set in the zero-argument constructor in the Java class.

2.4.1 Example: ScannableGaussian with setter injection

Fields of the `ScannableGaussian` are set as properties in the Spring beans configuration file, and default values defined. Atomic fields are defined with ‘`name`’ and ‘`value`’ attributes fields; array fields are defined using the ‘`list`’ tag:

```
<bean id='scannableGaussian1' class='org.myls.gda.device.scannable.ScannableGaussian'>
  <property name='name' value='simpleScannable1' />
  <property name='position' value='0.0' />
  <property name='inputNames'>
    <list>
      <value>x</value>
    </list>
  </property>
  <property name='extraNames'>
    <list>
      <value>y</value>
    </list>
  </property>
  <property name='level' value='3' />
  <property name='outputFormat'>
    <list>
      <value>%5.5G</value>
      <value>%5.5G</value>
    </list>
  </property>
  <property name='units'>
    <list>
      <value>mm</value>
      <value>counts</value>
    </list>
  </property>
</bean>
```

Now instantiate a ScannableGaussian using a predefined Gaussian Spring bean. Spring beans definition of a test Gaussian object:

```
<bean id='testGaussian' class='org.myls.gda.device.scannable.Gaussian'>
  <property name='testGaussian' value='testGaussian' />
  <property name='centre' value='0.0' />
  <property name='width' value='1.0' />
  <property name='height' value='1.0' />
  <property name='noise' value='0.1' />
</bean>
```

This test Gaussian bean can be used to create an instance of a ScannableGaussian using constructor injection with the test Gaussian as a constructor argument:

```
<bean id='scannableGaussian2'>
  <property name='name' value='scannableGaussian2' />
  <constructor-arg ref='testGaussian' />
</bean>
```

2.4.2 Exercise

Start with an empty server_beans.xml file, add Scannable components one by one, and test them in the GDA Jython console (requires server restart to incorporate the new components).

2.5 Examples of other Scannable classes and tests in GDA

- DummyMotor: from core: gda.device.motor.DummyMotor
- ScannableMotorTest: from core/test: gda.device.scannable.ScannableMotorTest
- TotalDummyMotor from core (used by test): gda.device.motor.TotalDummyMotor

2.6 Demonstrate use of Scannable in terminal

The new components are now available to be controlled from the GDA client.

2.6.1 Scan 1D

The example scanables can be scanned and manipulated from the Jython terminal in the GDA GUI.

Scan the example scannable `scannableGaussian0` from -2 to 2 in steps of 0.01:

```
>>> scan scannableGaussian0 -2.0 2.0 0.1
```

Change the width of `scannableGaussian0` from 1 to 2, and rescan:

```
>>> scannableGaussian0.setWidth(2)
>>> scan scannableGaussian0 -2.0 2.0 0.1
```

Change the centre of `scannableGaussian0` to -1.0 and rescan:

```
>>> scannableGaussian0.setCentre(-1)
>>> scan scannableGaussian0 -2.0 2.0 0.1
```

2.6.2 Nested scan

Import the demo scannable classes defined in the user‘q demonstration module `scannableClasses.py` (located in ‘documentation/users/scripts’, and viewable from from the JythonEditor view):

```
>>> import scannableClasses
>>> from scannableClasses import *
>>> sgw = ScannableGaussianWidth('sgw', scannableGaussian0)
>>> scan sgw 0.2 2.0 0.2 scannableGaussian0 -1.0 1.0 0.02
```

This nested scan has an outer scan which sets the width of the contained scannable Gaussian to different values from 0.2 to 2.0 in steps of 0.2. The inner scannable is then plotted for each width from -1.0 to 1.0 in steps of 0.02

CLIENT GUI DEVELOPMENT

3.1 Introduction

Having written new devices in Java and created instances of them on the server, they can now be examined and controlled from the interactive Jython interpreter. However, the GDA also allows developers to build custom graphical components for direct interaction with the server-side devices. These custom GUI components can be added to the core GDA client. This mechanism allows for a much more flexible means of examining and controlling new devices.

The process of extending the GDA client in this way requires two development steps:

- Writing the GUI component (currently in Swing)
- Setting up the communication between the client-side GUI component and the server-side device

These processes will be illustrated using an example of a simple power supply device, which has two states: 'On' and 'Off'.

3.2 Writing the Swing GUI component

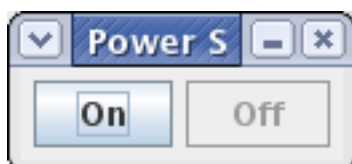
All custom GDA GUI components must extend the GDA class `AcquisitionPanel` (in the `uk.ac.gda.core` plugin, package `gda.gui`). This enables correct behaviour of the new component in the GDA client, and enables some of the components of client-server communication.

The example power supply GUI component extending `AcquisitionPanel` is in the documentation src tree, `org.myls.gda.gui.PowerSupplyPanel`. It comprises two `JButtons`, 'On' and 'Off'. Its behaviour is to toggle the state of the server-side power supply between the two states using the buttons. When the state of the underlying power supply is 'Off', the 'Off' button is disabled, and the 'On' button enabled. Clicking on the 'On' button changes the state of the server-side power supply to 'On' through CORBA-implemented client-server communication. The GUI component is registered as an observer of the power supply, and responds to any changes in state of the server-side device. In this example, the state of the server-side power supply has been changed by the user. The GUI component responds to the changed state by disabling the 'On' `JButton`, and enabling the 'Off' button.

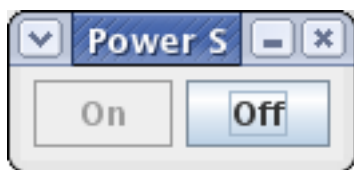
This behaviour is implemented in GDA using the `Observer` and `Observable` interfaces. The `PowerSupplyPanel` registers itself as an `Observer` of the server-side power supply, which sends changes in state back to the `PowerSupplyPanel`.

A standalone application which wraps the `PowerSupplyPanel` component is `org.myls.gda.gui.PowerSupplyTestFrame`, and can be run from the command line or from within Eclipse.

Appearance of the `PowerSupplyPanel` when the power supply is in state 'Off':



Clicking the 'On' button changes the state of the remote power supply to 'On'. This change is detected by the client-side panel, which updates to the set below:



3.3 Adding the new component to the GDA client

The new GUI component is added to the GDA client by defining an instance of it in the Spring beans configuration file 'client_beans.xml' in the documentation/xml directory. This configuration file is for including custom (non-core GDA) components to the GDA client. (Another client configuration file, 'client.xml', is used to define which of the predefined GDA graphical components to include in the client.)

The 'client_beans.xml' configuration file is a Spring beans container, and follows the Spring beans XML schema. The bean definition for the PowerSupplyPanel is:

```
<bean id='powerSupplyPanel' class='org.myls.gda.gui.PowerSupplyPanel'>
  <property name='name' value='gdaPowerSupplyPanel1' />
  <property name='psuName' value='gdaPowerSupply1' />
</bean>
```

The new view appears in the GDA client as the panel 'gdaPowerSupplyPanel1'.

3.4 CORBAising the object

See the corba manual next to this file for more specifics.

3.4.1 Writing the CORBA classes for GUI-server communication

Several generic CORBA classes are provided which external developers can extend. These include CORBA adaptors, Impl classes, and base classes.

3.4.2 Creating CORBA proxies for client-server communication

The new device is now available on the server and can be controlled from the Jython interpreter in the Scripting Terminal view. For example, scannable objects can be scanned, and the result of the scan displayed automatically in plot window.

However, it may be the case that a specialised GUI panel needs to be developed to control and simply the new device. In that case, communication must be set up between the client (GUI) and the object on the server. In GDA, this communication is CORBA-based.

CORBA proxies for custom devices are built using the Ant target 'make-corba-jar'. This target calls 'compile-corba-classes', which in turn calls 'compile-idl-definitions'. This invokes a CORBA idl2java compiler, in this case 'org.jacorb.idl.parser'.

Therefore, for each custom device that has been written, it is also necessary to define an IDL for that class.

An example of an IDL for a simple power supply device is:

REMOTING

4.1 CORBA in the GDA

4.1.1 Introduction

This section describes how to create a new object and CORBA-enable it - that is, create the additional files, interfaces and classes needed for the clients to interact with the object remotely.

4.1.2 Writing the Java code

Create the Java interface for the object

External hyperlinks, like `gda.device.detector.Phantom`.

Write an implementation of your interface

For example `gda.device.detector.phantom.PhantomV73`.

Note that if the interface is `some.package.Xxx`, the implementation should be in `some.package.xxx.SomeClass`.

In the case of the Phantom, the `PhantomV73` class fits into the Device/Scannable/Detector hierarchy, but an object implementing the `IPhantomV73Controller` interface is used to actually interact with the hardware (or a simulation of it).

4.1.3 Creating CORBA-specific files

Create an IDL that matches the Java interface

For example, `phantom.idl`

A few points:

- Note that whereas the Java interface is called Phantom, the CORBA interface is called CorbaPhantom.
- Java and CORBA types are different; for example, a Java int corresponds to a CORBA long.
- Input parameters must be prefixed with in; for example: `double getDemandVoltage(in long electrodeNumber) raises (device::corba::CorbaDeviceException);`

Compile the IDL to create CORBA classes

(It is no longer necessary to add the IDL file to an Ant script; the `make-corba-jar` target automatically picks up all IDLs.)

From the root of the GDA project, type:

```
$ ant make-corba-jar
```

Or if this fails, try:

```
ant -f build-classic.xml make-corba-jar
```

which will create a new `gda-corba.jar` that will include new classes for your object. For Phantom these classes include:

- `CorbaPhantomOperations` - interface containing the Phantom-specific operations (e.g. `setupForCollection`)
- `CorbaPhantom` - interface representing the CORBA version of Phantom; extends `CorbaPhantomOperations` plus some other CORBA interfaces
- `_CorbaPhantomStub` - implements `CorbaPhantom` and makes the CORBA remote requests
- `CorbaPhantomHelper` - various utility methods for working with `CorbaPhantom` objects

Write the CORBA implementation/adaptor classes

These classes must be located in the correct package so they are found.

- The interface for the device will be in `some.package.Xxx`.
- The `ImplFactory` requires the implementation class to be named `some.package.xxx.corba.impl.XxxImpl`.
- The `AdapterFactory` [`AdapterFactory`] requires the adapter class to be named `some.package.xxx.corba.impl.XxxAdapter`.

Implementation class

For Phantom, this is `PhantomImpl`.

The implementation class must extend your CORBA object's POA class (for Phantom, this is called `CorbaPhantomPOA`).

- The class needs two fields: * The real object - a Phantom in the case of the Phantom. * A POA field.
- You need a 2-arg constructor which takes the "real" object and the POA. `ImplFactory` will use this constructor.
- Each method that you implement should delegate to the "real" object; any exceptions must be converted into CORBA-specific exceptions (e.g. `DeviceException` to `CorbaDeviceException`). See `PhantomImpl` for examples of how to implement these methods.

Adapter class

For Phantom, this is `PhantomAdapter`.

The adapter class may extend other adapter classes but always needs to implement your Java interface (e.g. `Phantom`).

- The class needs three fields: * A CORBA object (e.g. a `CorbaPhantom` for the Phantom). * A `NetService`. * The object's name.

- You need a 3-arg constructor which takes a CORBA object, the object's name, and a NetService. Adapter-Factory will use this constructor.
- Each method that you implement should delegate to the CORBA object; any CORBA exceptions must be converted into corresponding non-CORBA exceptions (e.g. CorbaDeviceException to DeviceException). See PhantomAdapter for examples of how to implement these methods.

4.1.4 How the remote call works

Once the CORBA work has been done, the object can be used like this:

```
MyObject myObject = Finder.getInstance().find("My_Object_Name");
myObject.myMethod("foobar");
```

The way this is handled is as follows:

- myObject.myMethod("foobar") calls the corresponding method in the adapter.
- The adapter calls the CORBA stub.
- The CORBA stub makes the remote call across the network.
- On the server, the corresponding method in the implementation class is called by CORBA.
- The implementation class calls the "real" object.

4.1.5 Reference

Phantom.idl:

```
#ifndef _PHANTOM_IDL_
#define _PHANTOM_IDL_

#include <detector.idl>

module gda {
  module device {
    module detector {
      module phantom {
        module corba {

          /**
           * An interface for a distributed motor class
           */
          interface CorbaPhantom : device::detector::corba::CorbaDetector
          {
            void setUpForCollection(in long numberOfFrames, in long framesPerSecond, in long width, in long height)
              raises (device::corba::CorbaDeviceException);
            any retrieveData(in long cineNumber, in long start, in long count) raises (device::corba::CorbaDeviceException);
            string command(in string commandString) raises (device::corba::CorbaDeviceException);
          };

        };
      };
    };
  };
};
#endif
```

ImplFactory:

```
gda.factory.corba.util.ImplFactory
```

AdaptorFactory:

```
gda.factory.corba.util.AdaptorFactory
```

PhantomImpl:

```
gda.device.detector.phantom.corba.impl.PhantomImpl
```

PhantomAdaptor:

```
gda.device.detector.phantom.corba.impl.PhantomAdaptor
```

4.2 Alternatives to CORBA

4.2.1 Using RMI

Using a standard RMI exporter/proxy

For newly-written objects, RMI can be used to make those objects available over the network.

Spring's [RmiServiceExporter](#) can be used on the server side to make an object remotely available. It must be told which object is being exported, the name to export the object with, and the *service interface* - the interface defining the methods that should be available to clients. For example:

```
<!-- the object that is to be made remotely available -->
<bean id="server" class="...">
    ...
</bean>

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName" value="gda/ProsilicaServer" />
    <property name="service" ref="server" />
    <property name="serviceInterface" value="gda.images.camera.prosilica.server.ProsilicaImageSer
</bean>
```

On the client side, Spring's [RmiProxyFactoryBean](#) can be used to generate a proxy to the object on the server. It will create a proxy object that implements the service interface; each method makes a call to the remote object. For example:

```
<bean id="prosilica_server" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl" value="rmi://otherserver/gda/ProsilicaServer" />
    <property name="serviceInterface" value="gda.images.camera.prosilica.server.ProsilicaImageSer
    <property name="refreshStubOnConnectFailure" value="true" />
</bean>
```

The `refreshStubOnConnectFailure` property causes the client to reconnect to the server if, for example, the server is restarted. This allows a hot restart of the server without the need for the client to be restarted.

Note that using `RmiProxyFactoryBean` means that **every** call to a method in the service interface will result in a remote method invocation. This is not appropriate, for example, for objects that implement `IObservable` for eventing. See the next section for a solution to this.

There are currently a number of issues that prevent this mechanism from being used instead of CORBA for objects such as scannables:

- The 'remote interface' defined by the CORBA IDL files, and the adapter and implementation classes, often differ from the methods implemented by the 'real' object. An object exported using `RmiServiceExporter`, and a proxy automatically generated by `RmiProxyFactoryBean`, would not account for these differences.

- CORBA adapter and implementation classes often include additional logic not present in the ‘real’ object. They also sometimes carry out type conversion. Again, using the standard RMI exporter/proxy beans would not take these differences into account.
- CORBA adapter and implementation classes often carry out conversions between ‘real’ exception types (such as `DeviceException`) and CORBA-specific exception types (such as `CorbaDeviceException`). This means that the exceptions that a client needs to be prepared to handle are often quite limited. Using an automatically-generated RMI proxy would mean that the client may need to be modified to deal with other exception types.

However, if you are developing new objects and wish to invoke methods remotely, using this mechanism may be sufficient.

Using `GdaRmiProxyFactoryBean`

Spring’s `RmiProxyFactoryBean` will perform a remote method invocation for every method in an object’s service interface, including methods in the `IObservable` interface, if the server-side object implements that interface. To avoid this, `GdaRmiProxyFactoryBean` should be used instead.

`GdaRmiProxyFactoryBean` creates a proxy that handles `IObservable` method calls locally. The proxy is automatically connected to the CORBA event dispatch system, and registers to receive events related to the server-side object. It also maintains a client-side list of observers. When events are received by the client-side proxy, they will be dispatched to observers.

GDA CONFIGURATION

5.1 Spring configuration

5.1.1 FindableNameSetterPostProcessor

Putting this:

```
<bean class="gda.spring.FindableNameSetterPostProcessor" />
```

in your Spring XML file causes all `Findable` objects to have their `name` property set to be the same as the Spring `id`. Therefore you don't (except in a couple of special cases) need:

```
<property name="name" value="..." />
```

5.1.2 Making properties from `java.properties` available

Use this:

```
<context:property-placeholder location="file:${gda.config}/properties/java.properties" />
```

It allows you to use properties in your Spring XML files. For example:

```
<property name="hostname" value="${gda.images.camerahost}" />
```

5.1.3 Instantiating EPICS devices directly

For example:

```
<bean id="S1_top_motor" class="gda.device.motor.EpicsMotor">  
  <property name="pvName" value="BL04J-AL-SLITS-01:Y:PLUS" />  
</bean>
```

5.1.4 Instantiating using the EPICS interface “behind the scenes”

This is for those who don't like having PVs in their XML files ;-)

Put this somewhere in the Spring XML file (it doesn't need an ID):

```
<bean class="gda.configuration.epics.EpicsConfiguration">  
  <constructor-arg value="${gda.config}/xml/epics-interface.xml" />  
</bean>
```

Then do this:

```
<bean id="S1_top_motor" class="gda.spring.EpicsMotorFactoryBean">
  <property name="deviceName" value="S1.YP" />
</bean>
```

EpicsMotorFactoryBean is a Spring factory bean - the S1_top_motor object will actually be an EpicsMotor.

In addition to EpicsMotorFactoryBean, there is also EpicsMonitorFactoryBean and EpicsPositionerFactoryBean (they all need a deviceName).

5.1.5 Importing one file into another

```
<import resource="S1.xml" />
```

Effectively, the `<import>` is replaced with the contents of the imported file. All the beans are in the same Spring *context* (i.e. no need to duplicate the PropertyPlaceholderConfigurer, the FindableNameSetterPostProcessor, etc.).

5.1.6 Please use the `ref` attribute!!!

Instead of this:

```
<bean id="s1_bottom" class="gda.device.scannable.ScannableMotor">
  <property name="motorName" value="S1_bottom_motor" />
</bean>
```

you can do this:

```
<bean id="s1_bottom" class="gda.device.scannable.ScannableMotor">
  <property name="motor" ref="S1_bottom_motor" />
</bean>
```

Note the property is `motor`, not `motorName`, and this uses the `ref` attribute - which plugs the S1_bottom_motor motor into the s1_bottom object (so the ScannableMotor doesn't need to use the Finder to get the underlying motor - it's already wired up using Spring).

Since Spring has this dependency injection capability, there's no need to use the Finder in new classes - Spring can be used to do the wiring.

5.1.7 Making remote objects available through CORBA

You'll need this in your server-side configuration:

```
<corba:export namespace="stnBase" />
```

You need to declare the `corba` namespace by putting this at the top of the XML file:

```
xmlns:corba="http://www.diamond.ac.uk/schema/gda/corba"
```

and adding these entries to the `xsi:schemaLocation` attribute:

```
http://www.diamond.ac.uk/schema/gda/corba http://www.diamond.ac.uk/schema/gda/corba/gda-corba-1.0
```

Due to a limitation of Spring, property placeholders cannot be used in the namespace attribute when using `<corba:export />`. So this, for example:

```
<corba:export namespace="{gda.beamline.name}" />
```

will not work. (Property placeholders are typically resolved by a `PropertyPlaceholderConfigurer`, which is a `BeanFactoryPostProcessor` that operates on bean definitions in an application context. The `<corba:export />` element itself is not transformed into a bean definition: it uses the namespace value to add bean definitions for remote objects. It is not possible for the `PropertyPlaceholderConfigurer` to resolve placeholders used in the namespace attribute before that value is used to find remote objects.)

5.1.8 Importing remote objects from another object server

You'll need this in your client-side configuration:

```
<corba:import namespace="stnBase" />
```

As with `<corba:export />`, to use the `corba` namespace you need to declare it at the top of the XML file.

The good thing about using `corba:import` is that 'hidden' beans are added to the Spring context for all of the remote objects, so you can use them in any `ref="..."` attributes elsewhere in the file.

5.1.9 The `corba` namespace

If you use the `<corba:export>` or `<corba:import>` elements described above, you should add the schema for the `corba` namespace to the Eclipse XML Catalog, so that Eclipse can validate XML files containing these custom elements. To do this:

- Open the Eclipse preferences (Window → Preferences)
- Go to XML → XML Catalog
- Click "Add..."
- Enter the following details:
 - Location: click "Workspace..." and select `uk.ac.gda.core/src/gda/spring/namespaces/corba/gda-c`
 - Key Type: choose "Namespace Name"
 - Key: enter `http://www.diamond.ac.uk/schema/gda/corba/gda-corba-1.0.xsd`

Due to an issue with SpringSource Tool Suite, you may still get the following warning, which can be ignored:

```
Unable to locate Spring NamespaceHandler for element 'corba:export' of schema namespace
'http://www.diamond.ac.uk/schema/gda/corba'
```

5.1.10 `SingletonRegistrationPostProcessor`

```
<bean class="gda.spring.SingletonRegistrationPostProcessor" />
```

This registers certain objects you create in the Spring context as the application-wide singleton instances (e.g. the metadata).

(Objects in Spring XML files are, by default, singletons. In a perfect world, the metadata and other singletons could be injected into other objects, rather than the other objects calling `Whatever.getInstance()`. In practice it's difficult to do this because (1) there are too many objects that need the singletons; and (2) not all of those objects will be defined in the Spring XML file. It's good to define the objects in the Spring XML file, as this gives us complete control over their configuration, and means we can swap the real objects for mock objects. But this means we need to register those objects with some kind of registry.)

5.1.11 Property editors

`PropertyEditor` ([Javadoc](#)) is a standard Java interface concerned with converting text representations of property values into their ‘real’ types (among other things).

In Spring they are used to convert the text values used in Spring configuration files into the type required by the bean being instantiated. Spring has built-in support for many types already, but by putting this in your Spring configuration:

```
<import resource="classpath:gda/spring/propertyeditors/registration.xml" />
```

you will also be able to set properties of these types:

- `double[][]` - 2D double array
- `org.apache.commons.math.linear.RealMatrix` - Commons Math matrix

and any other types supported by the `PropertyEditors` listed in the `GdaPropertyEditorRegistrar` class.

5.1.12 Example Spring configuration

The Diamond I04.1 beamline uses Spring exclusively for its configuration. If you have access to the GDA Subversion repository, you can [view the I04.1 configuration](#). The Spring contexts for the two object servers are split into multiple XML files, which are all in the `xml/server` directory.

5.2 Logging

Logging messages can be generated not only by GDA classes, but also by third-party libraries such as Commons Configuration. GDA classes typically use the `SLF4J` API for logging. Log entries from code that uses Commons Logging or Log4j are redirected into `SLF4J` using two `SLF4J` bindings: *Commons Logging over SLF4J* and *Log4j over SLF4J*.

GDA uses `Logback` as the `SLF4J` implementation, so logging entries are passed from `SLF4J` to `Logback`.

5.2.1 Server-side logging configuration

The server-side logging configuration is used for object servers, and for the event server.

GDA has a default server-side logging configuration file, located in the `uk.ac.gda.core` plugin in the file `src/gda/util/logging/configurations/server-default.xml`.

A server-side logging configuration file for a particular GDA configuration can be specified using the `gda.server.logging.xml` property. The default server-side configuration will be applied first, followed by the custom configuration.

5.2.2 Client-side logging configuration

GDA has a default client-side logging configuration file, located in the `uk.ac.gda.core` plugin in the file `src/gda/util/logging/configurations/client-default.xml`.

A client-side logging configuration file for a particular GDA configuration can be specified using the `gda.client.logging.xml` property. The default client-side configuration will be applied first, followed by the custom configuration.

5.2.3 Using property placeholders in Logback configuration files

You can make properties defined in `java.properties` available for use in a Logback configuration file by adding the following element to the top of the file (inside the `<configuration>` element):

```
<property file="${gda.config}/properties/java.properties" />
```

(Use of `${gda.config}` works here because `gda.config` is a system property.)

You can then use property placeholders elsewhere in the file. For example:

```
<appender name="SOCKET" class="ch.qos.logback.classic.net.SocketAppender">
  <RemoteHost>${gda.logserver.host}</RemoteHost>
  <Port>${gda.logserver.port}</Port>
  ...
</appender>
```

5.3 Metadata

5.3.1 ICAT

The `icat` property of the `GdaMetadata` instance should be defined as follows:

```
<bean class="gda.data.metadata.icat.IcatConnectionDetails">
  <property name="name" value="icat" />
  <property name="url" value="jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(HOST=${oracle.host})) (PRO
  <property name="user" value="${icat.username}" />
  <property name="password" value="${icat.password}" />
  <property name="shiftTolerance" value="1440" />
  <property name="icatClassName" value="gda.data.metadata.icat.DLSIcat" />
</bean>
```

(`oracle.host`, `oracle.port`, `icat.username`, and `icat.password` are properties defined in the `java.properties` file.)

The `DLSIcat` class provides connectivity to the ICAT database. There is an alternate class in the `uk.ac.gda.core` plugin called `XMLIcat` which uses an XML file as a database. This is primarily for use in unit testing or offsite demonstrations, but could also be used by other facilities if they do a database dump into that format.

GDA DEMO

6.1 Basic commands

To get help:

help

scannable = software abstraction of angles, slits, energy, temperature probe, detector...

pos – show current positions of all scannables

e.g. pos x, pos y, pos z

shows extended syntax – no brackets

Move: pos x 10

ls – look at objects of certain types

e.g. ls Motor

Easy to write dummy scannables, e.g. x/y/z, for testing

6.2 Other scannables

t shows time since initialisation

dt shows time since last data point captured

w waits for specified time. e.g. to wait 2 seconds:

```
>>>pos w 2
```

all single-value position so far

multi-input – can move to multi-value position – e.g. pos mi [2, 3]

multi-extra – read-only output values – e.g. pos me

can combine – mie – one input, two (read-only) outputs – pos mie 4

6.3 Default detectors

list_defaults

add_default pil

remove_default pil

6.4 Beam focusing

fwhm = full width half maximum

minimise fwhmarea = area of spot on detector in pixels

Scan to show the images being plotted:

```
>>> scan f 430 600 20 pil 20
```

To display the images: images plotted on “Data Vector” panel

to focus on region of interest:

```
peak2d.setRoi(50, 50, 150, 150)
```

6.4.1 wide scan

```
>>> scan f 430 600 20 pil 20 peak2d
```

(finds 490 as the minimum)

data plotted as it's collected

6.4.2 finer scan

::

```
>>> go minval
>>> rscan f -20 20 2.5 pil 20 peak2d
```

(finds 482.5 as the minimum)

6.4.3 get feature details

::

```
>>> minval
```

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*