DIA Support Code

Basic Library Use

The support package for DIA is called `signal` and handles one-dimensional signals and two-dimensional images. Each of which can either be real or Complex. This document contains the most basic use with the details in the JAVADOC at www.ph.ed.ac.uk/ wjh/java/docs/dia/.

To use the package you must import it, so at the top of all calling classes, there must be:

```
import uk.ac.ed.ph.signal.*;
```

in addition you *must* set a *Unix* environmental variable to locate the native library by setting the `LD_LIBRARY_PATH` variable. This is best done by adding the line,

```
export LD_LIBRARY_PATH=/ifp/java/lib/
```

to your `.bash_profile` in your home directory which is executed every time you login[1]. You can test that this works by typing:

```
echo LD_LIBRARY_PATH
```

at the command line.

If you want to use Complex data, most users will also want to also import `jfftw` package, which defines a useful `Complex` class by also adding:

```
import jfftw.*;
```

at the top of your program(s). The class is detailed towards the back of this document.

2 Dealing with Images

Images are processed through the `Image` classes using either `RealImage` to work with real images, or `ComplexImage` to work with complex images.

Note: when taking Fourier transform it is most convenient to convert a real image into a complex image and work with the `ComplexImage` class. Taking Fourier Transform of real images is rather complicated, and is not documented here, see the `jfftw` documentation at www.ph.ed.ac.uk/ wjh/teaching/Java/fft/ for details.

2.1 Real image using `RealImage` class

Constructors: there are three common constructors for the `RealImage` class being:

1. `RealImage(int width,int height)` creates a real image of size `width` by `height` pixels initialised to zero.

[1] When you first do this you will need to logout and back in again.

2. `RealImage(RealImage im)` creates a new `RealImage` which is a clone of the specified RealImage `im` including a copy of the image data.

3. `RealImage(ComplexImage im, int flag)` creates a `RealImage` where the data is copied from the specified ComplexImage `im` using the data conversion `flag`. This `flag` is as taken by he `Complex` method `getDouble(int flag)` in section 4.

in addition there is a clone method of

1. `RealImage clone()` which returns a clone of the current RealImage.

Reading Images

The simplest route to read images is via the static method to `RealImage` being

1. RealImage.readImage(String file) which will read a image from `file` depending on its extension. The accepted extensions are:

   - `pgm` portable grey map monochrome images in raw or asci format; this is the format used for project test images.
   - `jpg, jpeg, gif, bmp, png` image types. If the image is full colour the *green* image is read. See `realImageIO` for better control if you need it.

Getter Methods

There are a range of getter method to get image and pixel information the most used are

1. `int getWidth()` get the width (or x-dimension) of the image.

2. `int getHeight()` get the height (or y-dimension) of the image.

3. `double getDouble(int i, int j)` get the value of the i, j pixel.

4. `double[] getDataBuffer()` get the whole image buffer as a one-dimensional array.

There are also other getter method to `getRow(int j)` and `getColumn(int i)` as a `RealSignal` and also `getBufferedImage()` to get the image data as a `BYTE_GRAY BuffererImage` which can be painted into other JAVA objects. See the JAVADOC for details.

Setter Methods

The only basic user setter is to set and individual pixel, this being

1. `void setDouble(int i, int j, double value)` to set the `i,j` pixel to `value`.

For the other setter methods, see the JAVADOC.

2.1.1 Statistics of RealImages

Image statistics are calculated and returned by the `RealStatistics` object which can either be constructed externally and attached to an `RealImage` (for advanced use), or calculated and returned by the `RealImage` method.

1. `void calculateStatistics()` which causes the statistics to be calculated, or re-calculated if called for a second time. This will only work in real space, and pixels set to `Double.NaN` will be ignored.

2. `RealStatistics getStatistics()` which returns the `RealStatistics` object associated with the `RealImage`.

The actual statistics are then obtainable from the *getter* methods of `RealStatistics` being

1. `double getMin()` get the minimum pixel value.

2. `double getMax()` get the maximum pixel value.

3. `double getMean()` get the mean image value.

4. `double getPower()` get power of image being $\langle |f(i,j)|^2 \rangle$.

5. `double getVariance()` get image variance.

6. `String toString()` get all the statistics in a formatted `String`.

2.1.2 Displaying Images

The simples scheme to display images is via the `DisplayImage` class that display images and gives basic on-screen manipulation such a re-sizing and zooming. This has a single constructor being

- `ImageFrame(String title)` where `title` is the text that appear in the top bar.

an image can then be added by one of the following method

1. `void addImage(RealImage im)` which adds the image using the image statistics from min and max, if they are set, otherwise they will be automatically calculated.

2. `void addImage(BufferedImage bi)` add any JAVA displayable `BufferedImage`, experts only.

on adding an image the frame is automatically set visible. There are a set of buttons along the top, the most useful is SAVE which will save the displayed image in either `gif`, `jpeg`, `png` or `bpm` format.

2.2 Complex image using `ComplexImage` class

Constructors: there are three common constructors for the `ComplexImage` class being:

1. `ComplexImage(int width,int height)` creates a complex image of size `width` by `height` pixels initialised to zero.

2. `ComplexImage(ComplexImage im)` creates a new `ComplexImage` which is a clone of the specified ComplexImage `im` including a copy of the image data.

3. `ComplexImage(RealImage im)` creates a `ComplexImage` where the real part of the pixel values is taken from the specified `RealImage`.

in addition there is a clone method of

1. `ComplexImage clone()` which returns a clone of the current ComplexImage.

There is no direct method to read images, they need to be read as a `RealImage` and then converted.

Getter Methods

The `getWidth()`, `getHeight()` and `getDataBuffer()` methods are identical to `RealImage`, the pixel data methods are

1. `Complex getComplex(int i, int j)` get value of the `i,j` pixel as a `Complex`.
2. `double getDoubleValue(int i, int j, int flag)` get he value of the `i,j` pixel as a `double` where the conversion is controlled by `flag` as taken by the `Complex` method `getDouble(int flag)` in section 4.

Setter Methods

There are two basic setter for pixel data, these being

1. `void setComplex(int i, int j, Complex c)` set pixel `i,j` with value of complex `c`.
2. `void setComplex(int i, int j, double a, double b)` set pixel `i,j` with value of $\mathtt{a}+\imath\mathtt{b}$.

There are also other getter method to `getRow(int j)` and `getColumn(int i)` as a `ComplexSignal`, see JAVADOC for details.

Fourier Methods

There are three basic

1. `void fourier()` take in-place Fourier transform overwriting the data with $(0,0)$ is *top/left*. If data is the *real* space, a forward transform is take, if in *fourier* space it is the inverse.

   Normalsiation occurs on inverse transform only.

2. `void centreFourier()` as `fourier()` but the Fourier space $(0,0)$ is shifted to the centre of forward and real space $(0,0)$ shifted back the *top/left* on inverse.
3. `RealImage powerSpectrum(boolean logPower)` returns the Power Spectrum as a `RealImage`. If `logPower` is true, the pixel values are the $\log(F(l,l)^2+1.0)$ otherwise they are $|F(k,l)|^2$.

   If the current image is is *real* space then `centreFourier()` is called leaving the image in *Fourier* space. If the current mage is already in *Fourier* space it is unaltered.

Note there are also similar methods for `RealImage` but the results are rather more complex and difficult to understand.

Display Method

There are no specific method to display or save complex images, they need to be converted to a `RealImage` first. The method

- `RealImage getRealImage(int flag)` where `flag` is the conversion type from Complex method `getDouble(int flag)`.

is the most obvious call.

3 `Complex` class

Any work with Fourier Transform involves complex numbers, which in this package are dealt with by the `Complex` class. This holds a complex at two `double`s being the real and imaginary part. This class has a set of support methods to perform most common complex arithmetic operations. This `Complex` class has been written with *ease of use* in mind and not efficiency, given this performance with modern version of `JDK` appear to be very respectable.

Constructors

There are four constructors for the Complex class being:

1. `Complex()` to create a Complex with real and imaginary parts set to zero.
2. `Complex(double a, double b)` to create a Complex with real set to `a` and imaginary set to `b`.
3. `Complex(double a)` to create a Complex with real set to `a` and imaginary set to zero.
4. `Complex(Complex c)` to create a Complex with real and imaginary parts being set by `c`.

In addition this class implements `Cloneable`, with method

- `Complex clone()` which return a clone of the current `Complex` where both real and imaginary parts are copied.

3.1 Setters

The internal variables can be set by following methods:

1. `void set(double a, double b)` set real part to `a` and imaginary part to `b`.
2. `void set(Complex c)` sets the real and imaginary parts to the same value as the real and imaginary parts of `c`.
3. `void setReal(double a)` sets the real part to `a`, and does not alter the imaginary part.
4. `void setImag(double b)` sets the imaginary part to `b`, and does not alter the real part.
5. `void setPolar(double rho, double theta)` sets real and imaginary parts with polar coordinates, so setting

$$\text{real} = \rho\cos(\theta) \quad \text{and} \quad \text{imag} = \rho\sin(\theta)$$

6. `void setExpi(double theta)` sets the Complex to $\exp(\imath\theta)$.
7. `void setRandomPhase(double m)` set the Complex number to have specified modulus `m` but the phase set randomly in the range $0 \rightarrow 2\pi$, with a flat random distribution given by `Math.random()`.
8. `void setInvalid()` sets both real and imaginary parts to `Double.NaN`.

Getters

The following methods to get information about the `Complex`, these being:

1. `double getReal()` return the real part of the Complex.
2. `double getImag()` return the imaginary part of the Complex.
3. `double modulusSq()` returns the modulus square of the Complex.
4. `double modulus()` returns the modulus of the Complex. This uses the simple $\sqrt{a^2+b^2}$ scheme for speed since, as of JAVA 1.5 `Math.hypot` is very very slow.
5. `double phase()` returns the phase of the Complex as defined for complex $a+\imath b$ by `Math.atan2(b,a)`.
6. `double logPower()` return the log of the power, which for number $a+\imath b$ is defined as

$$\log\left(a^2+b^2+1.0\right)$$

   where the the $+1.0$ prevents $-\infty$ for a square modulus of zero.

7. `double getDouble(int flag)` returns a `double` converted from the Complex as defined by the `flag` with

`flag`	Value
`Complex.REAL`	Real part
`Complex.IMAG`	Imaginary part
`Complex.MODULUS`	Modulus
`Complex.MODULUS_SQUARED`	Modulus squared
`Complex.PHASE`	Phase
`Complex.LOG_POWER`	Log power

8. `boolean inNaN()` returns `true` if either real or imaginary parts set to `Double.NaN`, see `setInvalid()`.
9. `String toString()` returns Complex as a formatted `String` of the for `[a.aaaa,b.bbbb]` where the format of the each number is controlled by `setFormatString` below.
10. `void setFormatSting(String fmt)` sets the format String used in `toString()` used by `String.format()`. the default is `"%g"`.

The internal real and imaginary `double` are `protected` being only available to extending classes; see the JAVADOC for details.

Three Parameter Arithmetic

These methods perform the basic Complex arithmetic and return a new Complex object without affecting either the current Complex or the parameter(s). These are:

1. `Complex plus(Complex c)` add Complex `c` to current Complex.

2. `Complex plus(double a, double b)` add Complex specified by real and imaginary parts.
3. `Complex plus(double a)` add real value to the real part, imaginary kept from current.
4. `Complex minus(Complex c)` subtract Complex `c` from current Complex.
5. `Complex minus(double a , double b)` subtract Complex specified by real and imaginary parts.
6. `Complex minus(double a)` subtract real value from real part, imaginary kept from current.
7. `Complex from(Complex c)` subtract current Complex *from* specified Complex `c`.
8. `Complex from(double a , double b)` subtract current Complex *from* Complex specified by real and imaginary parts.
9. `Complex from(double a)` subtract current Complex *from* real, imaginary will be *negative* of current.
10. `Complex mult(double a)` multiply current Complex by double `a`.
11. `Complex mult(Complex c)` multiple current Complex by specified Complex `c`.
12. `Complex mult(double a, double b)` multiply current Complex by specified Complex given as real and imaginary parts.
13. `Complex multConj(Complex c)` multiple current Complex by conjugate of specified Complex `c`.
14. `Complex over(double a)` divide current Complex by a double `a`.
15. `Complex over(double a, double b)` divide current Complex by Complex specified at two doubles.
16. `Complex over(double a, double b)` divide current Complex by specified Complex `c`.
17. `Complex under(double a)` divide the specified double `a` by the current Complex.
18. `Complex under(Complex c)` divide the specified Complex `c` by the current Complex.

Since these method always return a new `Complex` they can be chained for example,

```
Complex a = new Complex(3.0,4.5);
Complex b = new Complex(2.0,6.0);
Complex d = new Complex(7.0,10.0);

Complex d = a.mult(b).plus(c);
```

will give $d = a \times b + c$ without altering the contents of `a,b,c`.

Two Parameter Arithmetic

These methods perform "in place" arithmetic on the current Complex and are much more computational efficient that the three parameter versions since they do not allocate new memory. Here there is a less extensive set, these being:

1. `addTo(Complex c)` add `c` to the current Complex.
2. `addTo(double r, double i)` add complex $a + \imath b$ to current Complex.
3. `multBy(double a)` multiply current complex by real `a`.
4. `multBy(Complex c)` multiply current complex by `c`.
5. `void multBy(double a, double b)` multiply current complex by $a + \imath b$.

As the code is developed, additional method will be added which will and documented in the JAVADOC.