2 Introduction

The JFFTW package is a partial simple JNI interface to the C-callable FFTW3 library. The FFTW3 library is an extensive and highly optimised FFT library by Matteo Frigo and Steven Johnson from the Department of Applied Mathematics, MIT, This library has multiple internal algorithms and gives optimal performance on a vast range of processor types, and more significantly gives $N\log_2(N)$ performance for *any* length of FFT, including prime numbers, and not just for highly factorisable lengths. It is also very flexible designed to take FFT of almost any data configuration, with one, two, three dimensional transform of complex or real data packed in variety of formats.

This JAVA package implements a sub-set of the library and in particular concentrates on one-off FFTs and does not save `plans` between calls. For multiple FFTs this is computationally sub-optimal but vastly simplifies the calling procedure. In practise most JAVA programs are interactive and efficiency is not really an issue, if you want to take vast numbers of FFTs of large amount of data, you should use C and make direct calls to the FFTW3 library.

This package has two interfaces, these being:

1. `DataArray` class, and its extending classes of `RealDataArray` and `ComplexDataArray`. This is for user who want the simplest use and are happy to write their programs to use theses supplied classes to handle their data. Such users should read-up about the `Complex` class and `DataArray`, and should not need to get their *hands dirty* with `FFTW` at all.

   The `DataArray` classes also have extensive access methods with sanity checking of the supplied parameters. This does however have a significant efficiency overhead, but does increase the probability of your program actually working!

2. `FFTW` class and its extending classes `FFTWReal` and `FFTWComplex` operates on raw `double` arrays where the user is responsible for pack and understanding what they send to the underlying library. This is more efficiency and flexible, but rather harder work and more error prone, especially for real-data FFTs. To make life a bit easier there is also a `ArrayUtil` class help with some of the array manipulation.

Since the JAVA interface calls an underlying native C-library, getting the parameters wrong can result is some *interesting* effects, usually the whole JRE environment crashing in a messy heap! This is really a feature of C-code, and while the JAVA interface tries to trap the most common errors, a *dedicated* programmer will manage to to create such crashes, which come under the class of *probable user error!.*; so don't report them until you are *really certain* you have the call right!

The full JAVADOC and examples are available from the package HOMEPAGE.

3 Use of Package

On the local CPLab systems, `jfftw` is already in your `CLASSPATH` so to use it you just need to put

```
import jfftw.*;
```

at the top of your source code. In addition you *must* set the *Unix* environmental variable `LD_LIBRARY_PATH` so that the low-level interface is loaded. This is best done by adding the line

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

to your `.bashrc` file[1]. You can test that this has worked by typing

```
echo LD_LIBRARY_PATH
```

at the command line. If you fail to do this you will get an `UnsatisfiedLinkException` when you try and run your program.

For people wanting to use this on other machines, you can download the package source from the package homepage, unpack and install. There are elementary instructions in the included `README` file aimed at LINUX systems; other will follow, if I ever have the effort and inclination!

3.1 Speed and Size

The Fourier transform is an intrinsically numerically intensive algorithm and taking very large FFTs with tax even the largest of system. The *Fast* bit of FFT is when compared to the direct implementation of the discrete digital Fourier transform, which even for modest amount of data, is unusably slow.

JAVA is as resource extravagant with memory as with other things and by default (on Unix) system has a maximum memory heap of 256 Mbytes. By the time JAVA has it slice, this is sufficient to take one two-dimensional complex FFT of approximately 1500×1500; any larger[2] resulting in `OutOfMemoryError` run-time error. The memory maximum can be increase with the `-Xmx<nnn>M` flag, so that

```
java -Xmx512M MyJavaApp
```

with run `MyJavaApp` with a 512Mbyte memory heap.

The question "how big and how fast" is limited by two things, firstly raw floating point CPU speed, but more importantly memory bandwidth since the data being transformed is accessed almost continually in (almost) random order. To get any sensibly performance it is essential that all the data is resident in physical memory and is not being swapped to/from disc. On most normal desktop systems with IDE hard drives, swapping data to disk is totally catastrophic[3] and can result in FFT taking 100 of times longer than predicted while simultaneously rendering the system unusable due to swap-thrashing[4]. If you find this occurs; its obvious the disc activity light is solidly "on" and the system does not respond to the keyboard or mouse!; then try shutting down all other memory hungry processes, for example Firefox, OpenOffice etc. If this does not help, then your only solution is to fit more physical memory.

[1] This first time you do this you must logout and login again!

[2] Java code with extensive graphics will fail before this.

[3] Systems with SCSI fair a bit better.

[4] All the system is doing is swapping the data to/from disc and is not able to do anything else.

4 `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.

4.1 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.

DataArray Class

The DataArray class simplify the calling and data handling in the jfftw package, and via its extending classes of RealDataArray and ComplexDataArray supports one, two and three dimensional real and complex data sets. Internally the data is held as one-dimensional array of double[], with, for Complex types, being held in *interleaved* format, however the support method *hide* much of the complex access and indexing problems; it also does extensive array bound checking and sanity checking of parameters. This does however come at a considerable cost in efficiency. People worried about[5] should consider the lower level FFTW interface.

DataArray is itself is declared as abstract, with all the user access via it extending classes, which deal with real or complex data.

ComplexDataArray Class

This is an extending class to deal with Complex data in both real and Fourier space. This is by far the simplest method of use where you get *exactly what you expect.* Novice users are *very strongly* encouraged to use this class even if their data in real space is real only; remember the Fourier transform of real data is complex.

Constructors

This class has the following constructors to create ComplexDataArrays, these being,

1. ComplexDataArray(int width) create a one-dimensional ComplexDataArray of specified width with all elements set to zero.
2. ComplexDataArray(int width, int height) create a two-dimensional ComplexDataArray of specified width×height with all elements set to zero.
3. ComplexDataArray(int width, int height, int depth) create a three-dimensional ComplexDataArray of specified width×height×depth with all elements set to zero.
4. ComplexDataArray(ComplexDataArray data) creates a new ComplexDataArray being a copy or clone of the specified ComplexDataArray.
5. ComplexDataArray(RealDataArray data) creates a new ComplexDataArray of the same dimensions of the RealDataArray with its data used to set the *Real Part* of the complex elements. The RealDataArray must be in real space or an IllegalArgumentException is thrown.
6. ComplexDataArray(int w, int h, int d, double[] da) creates a ComplexDataArray of specified dimensions and used the supplied one-dimensional double[] array as the internal data buffer which is assumed to be formatted correctly. This will fail with IllegalArgumentException if da.length != 2*w*h*d.

   Note for two-dimensional data set d=1 and for one-dimensional data set h=1 and d = 1.

In addition there is the clone() method,

[5] If you are really worried about efficiency, call FFTW direct from "C".

- `ComplexDataArray clone()` which gives a deep clone including a clone of the underlying data.

Getters

The basic getters to get information about the internals of the class and are:

1. `getSpace()` get the space, either `FFTW.REAL` or `FFTW.FOURIER`.
2. `int getWidth()` get the width in pixels.
3. `int getHeight()` get the height in pixels, will be 1 if one-dimensional DataArray.
4. `int getDepth()` get the depth in pixels, will be 1 if one or two-dimensional DataArray.
5. `String toString()` gets information about array type, width, height, depth and current space as a formatted `String`.
6. `boolean getNormalisation()` whether automatic normalisation on inverse Fourier is active, default is `true`, see details under `fourier()`.
7. `int getConversion()` gets to Complex to double conversion flag, see details under `getDouble()` below.
8. `int length()` get the length of the internal `double` data buffer.
9. `double[] getDataBuffer()` get the internal `double` data buffer for direct access of the interleaved array. Experts only here, if you are calling this, seriously consider using the direct `FFTW` interface.
10. `FFTWComplex getFFTW()` get the underlying `FFTWComplex` object again not normally called by users.

For all setter and getters there are *three* version of each to deal with one, two and three dimensional underlying structures. They all have the same structure being,

1. `get<value>(int i)` in one-dimensions for `i < getWidth()`
2. `get<value>(int i, int j)` in two-dimensions. with `i < getWidth()` and `j < getHeight()`.
3. `get<value>(int i, int j, int k)` in three-dimensions. with `i < getWidth()`, `j < getHeight()` and `k < getDepth()`.

with array subscript checking. Only the one-dimensional version will be documented here, but all three versions are available.

The getters to access the data as complex are:

1. `Complex getComplex(int i)` get the `i` complex element in one-dimensions for `i < getWidth()`

while the corresponding getter to get the data as `double`s are

1. `double getDouble(int i)` get the `i` Complex as a `double`, element in one-dimensions for `i < getWidth()`

The `Complex` to `double` conversion is controlled by the conversion flag, which must be one on the six possible defined under `Complex.getDouble()`. This defaults to `Complex.MODULUS` so giving the modulus of the Complex.

Setters

The control setters are

1. `setSpace(int space)` manually set the space to either `FFTW.REAL` or `FFTW.FOURIER`. A creation this defaults `FFTW.REAL` and is then automatically controlled by calls to `fourier()`.

   Note this does *not* perform a Fourier transform, see `fourier()`.

2. `setNormalisation(boolean n)` controls the normalisation on inverse Fourier transforms, but default set to `true`.

3. `setConversion(int flag)` set the Complex to `double` conversion used by `getDouble()`. Defaults to `Complex.MODULUS`, and can be set any of the flags defined in `Complex`.

4. `setDimensions(int w, int h, int d)` set the dimensions used to access the data array *without sanity checking*. Normally used in conjunction with `setDataBuffer` with extreme care!

5. `setDataBuffer(double db[])` set the internal databuffer to the specified `double[]` array. This *must* be consistent with the current dimensions and of the correct structure. If you are using this, really think about using the direct `FFTWComplex` classes.

As with the getters above, only the one-dimensional version is documented here, but there are exact, and obvious equivalence for two and three dimensional again with full array bound checking.

1. `void setComplex(int i, double a, double b)` set the `i` elements in one-dimensions with real `a` and imaginary `b`.

2. `void setComplex(int i, Complex c)` set the `i` elements in one-dimensions with Complex `c`.

and the corresponding `double` are

1. `setDouble(int i, double a)` set the real part of `i` element.

where is all cases the *imaginary* element is not altered.

All the `setComplex` and `setDouble` have array bounds checking and will throw `ArrayIndexOutOfBoundsExcept` if out of bounds.

Data Element Manipulation

There are a set of method to modify the current values of the data elements. Only the one-dimensional versions are listed here but there are exact equivalent for two and three dimensions.

1. `add(int i, double a)` add `double a` to the current `i` element in one-dimensions.
2. `add(int i, double a, double b)` add $a+\imath b$ to the current `i` element in one-dimensions.
3. `add(int, Complex c)` add `Complex c` to the current `i` element in one-dimensions.
4. `mult(int i, double a)` multiply the current `i` element by `a`.
5. `mult(int i, double a, double b)` multiply the current `i` by element by $a+\imath b$.
6. `mult(int i, Complex c)` multiply the current `i` by element by `Complex c`.
7. `conjugate(int i)` take the conjugate of the `i`, this will be ignored if the data is real.

Array Data Manipulation

These methods act on the whole array. These methods use well optimised code so, especially for large array, should be used rather than writing your own using `getComplex()` and `setComplex()`.

1. `conjugate()` forms the complex conjugate by negating the imaginary parts of all elements. In data is real this is ignored.
2. `mult(double a)` multiplies all elements by a scalar.
3. `mult(double a, double b)` multiplies all elements by a complex specified at two doubles. This will fail for real only data.
4. `mult(Complex c)` multiple all elements by a `Complex`. This will fail for real only data.
5. `mult(DataArray d)` multiplier's the current array by the specified `DataArray` on a point by point basis. Both `DataArray` *must* be same dimensions and may be type `RealDataArray` or `ComplexDataArray`.
6. `RealDataArray getRealDataArray(int conversion)` returns the current `ComplexDataArray` as a `RealDataArray` (in real space), where where the Complex to double conversion is controlled by `int conversion` which can be any of the `Complex` conversion flag.
7. `RealDataArray getRealDataArray()` as above, but uses the internal conversion flag set by `setConversion(int flag)` method.

The Fourier Methods

Now finally to the Fourier methods,

1. `void fourier()` takes the normal Fourier transform overwriting the current data with its Fourier transform. If the data is in *real* space a *forward* transform will be taken and the space switched to *fourier* space. If the data is in *fourier* the inverse will be take. In both cases the `space` flag accessed via `getSpace()` is correctly set.

   The default is to normalise the Fourier transform on *inverse* only, so in one-dimensions for a complex samples signal $f(i)$ of length N is, the forward one-dimensional transform given by,

   $$F(k) = \sum_{i=0}^{N-1} f(i) \exp\left(-\imath 2\pi \frac{ik}{N}\right)$$

   where $\imath = \sqrt{-1}$ and the inverse transform is

   $$f(i) = \frac{1}{N} \sum_{k=0}^{N-1} F(k) \exp\left(\imath 2\pi \frac{ik}{N}\right)$$

   This behaviour can be altered by `setNormalisation(boolean n)` which if set to `false` then *no normalisation* will occur, which is the FFTW default.

2. `void centreFourier()` takes to *centred Fourier* transform with the zero frequency shifted to the centre $(N/2)$ element, for more generally `width/2,height/2,depth/2` element, where all dimensional are assumed by be *even*. The shift is done by pre-multiplication by a ± 1 checker pattern using method `checker()` so is reversible, with two calls to `centreFourier()` returning the data to its original state[6]. This method also behaves exactly like `fourier()` with the same forward/inverse and normalisation rules.

3. `RealDataArray powerSpectrum(boolean logPower)` returns the power spectrum, being the square modulus of the Fourier transform as a `RealDataArray`. If the current `DataArray` is in *real space*, a `centreFourier()` is applied automatically, and the data left in Fourier space. The `boolean logPower` controls whether the power (modulus squared) or $\log(|F(k)|^2 + 1)$ is returned.

For examples of how to use this class see the package HOMEPAGE

5 `RealDataArray` class

Most input data is real, be that signals in one-dimensional, images in two-dimensions or crystal structures in three-dimensions. However despite the input being real the Fourier transform will always be *Complex*, in this this case with Hermition symmetry. Since the Fourier transform is unitary then we should have the same number of data elements in real and Fourier space; however due to the packing scheme used within the FFTW library the Fourier space representation take *slightly* more space than the real. This can be overcome in C by over-allocation of memory in real space, but with JAVA's strict array-bound checking, this is much more difficult. Therefore this interface does *not* implement *in-place* real transforms but always returns

[6] This checker scheme only works for even array sizes.

the transforms results in newly allocated memory. This is more expensive in memory allocation, but from initial timing, does not appear to have a significant computational penalty[7]. The `RealDataArray` class tries to handle this in an intelligent way without the used being aware of what is happening, but at times care has to be taken.

Data packing in Real and Fourier Space

In real space the data is packed in a `double[]` array in *row order* so a three-dimensional data set is held in an array of length `width*height*depth`. However in Fourier space *slight more than half* of the Fourier space is needed and the data is packed in *interleaved* format in *Complex* array of dimensions `width/2+1,height,depth`. See the section on `FFTWReal` for detailed description of why this occurs.

Constructors

All the constructors are in real space, these being,

1. `RealDataArray(int width)` one-dimensional `RealDataArray` of length `width`.
2. `RealDataArray(int width,int height )` two-dimensional `RealDataArray` of size `width` by `height`.
3. `RealDataArray(int width, int height, int depth)` three-dimensional `RealDataArray` of size `width`, `height` by `depth`.
4. `public RealDataArray(RealDataArray da)` forms a `RealDataArray` taking all parameters from the specified `RealDataArray`. This constructor will also work for `da` being in Fourier space.
5. `RealDataArray(int w, int h, int d, double[] data)` forms `DataArray` of specified dimensions with the specified `double[]` data buffer. Only the size of the `double[]` array is checked with *must* be `w*h*d`.

Getter

The basic information getters is *identical* to the `ComplexDataArray` getters except,

1. `getWidth()` gets the width of the data as declared in *real space*.
2. `getCurrentWidth()` gives the current width of the data, which is real space is the same as `getWidth()` but in Fourier space will be `getWidth()/2 + 1`. See section on Fourier data structure for details of why.
3. `int length()` length of current databuffer, this *will* change between real and fourier space.
4. `double[] getDataBuffer()` will get the *current* databuffer but its location, and size *will change* between real and fourier space.

[7] It is considerably faster to take real transforms than to consider real data as being complex with the imaginary part being zero.

The data getters have the same call as for `ComplexDataArray`, but do different things in real and Fourier space,

1. `double getDouble(int i), getDouble(int i, int j), getDouble(int i, int j, int k)`
   (a) *Real Space* gets data element over the range given by `getWidth(), getHeight(), getDepth()`.
   (b) *Fourier Space* get the `double` value of the complex element using the current conversion flag. Here the range is `getCurrentWidth(), getHeight(), getDepth()`.
2. `Complex getComplex(), getComplex(int i, int j), getComplex(int i, int j, int k)`
   (a) *Fourier Space* get the `Complex` value of the element in the range is `getCurrentWidth(), getHeight(), getDepth()`.
   (b) *Real Space* gets the `Complex` version of type real data with the imaginary part set to zero. Here the range is `getWidth(), getHeight(), getDepth()`.

Setters

The control setter are identical to those for `ComplexDataArray`, except remember now since real and fourier space have a different structure, but `setSpace()` does *not* change the structure of the data, so careless use of this will almost always be disastrous!.

The data setters are again the same at for `ComplexDataArray` but again as the with the getters, when in real space the *width* of the data is `width`, and should be set by `setDouble()` while in Fourier space it is `width/2 + 1` and is normally set by `setComplex()`. In both cases the method `getCurrentWidth()` will always return the correct width.

5.0.1 Data Element Manipulation

Again the methods and parameters are identical to the `ComplexDataArray` expect as with the GETTERS and SETTERS the data *width* is different in real and Fourier space.

5.0.2 Array Data Manipulation

Here again, all all the methods are the same with the same parameters, but:

1. `conjugate()` does noting if in real space. There is no error flagged.
2. `mult(DataArray d)` the sizes of `d` *must* the current `DataArray` so *must* be different for real and fourier space. Also if the current `RealDataArray` is in real space `d` *must* also be a `RealDataArray` in real space.
3. `RealDataArray getRealDataArray()` if in real space, the current `DataArray` will be returned unaltered, but if in fourier space it will return a *new* `RealDataArray` in real space of being the `getCurrentWidth()` of current `DataArray`.

The Fourier Methods

Again these look superficially the same as the `ComplexDataClass` but extra care has to be taken.

1. `fourier()` takes the normal Fourier transform, but in fourier space there is *half* the frequency range, with the other half given by the Hermition symmetry of Fourier transforms of real data. The *first* index range is therefore *halved*[8]. If the *width* of the input data is N then the Fourier width will be $N/2+1$, and also element 0 *and* $N/2$ will be *real only*.

2. `centreFourier()` will shift the zero frequency exactly as for the `ComplexDataArray`, again being valid only for even sizes data.

3. `RealDataArray powerSpectrum(boolean logpower)` returns a `RealDataArray` of the size of the Fourier transform, so with the first index halved.

[8] Due to the FFTW packing there is actually one apparent *extra* element.

jfftw Package

The FFTW classes give lower level access library and are accessed through the two extending classes being FFTWComplex for complex-to-complex transforms and FFTWReal for real-to-complex transforms. These classes operate directly on double[] arrays and the user is responsible for all data packing and unpacking.

Using these classes is more efficient than the DataArray classes mainly since there is no array range checking and the internals are almost totally free of setter/getter indirections. However all FFTs are still executed as *one-off* with no re-use of FFTW plans. This still makes then much less efficient that real "C-code"

FFTWComplex Class

The extending class take complex FFT where the data is packed in one of two possible formats, these being:

1. *interleaved* where the complex data is packing in a single one-dimensional double[] array with complex pairs in adjacent elements with real values are in the *even* elements and imaginary in the *odd* elements.
2. *split* where the real and imaging data is packed in two-dimensional double[][] array where for the i^{th} complex value the real part is held in element [0][i] and the imaginary part in element [1][i].

In both cased multi-dimensional data is held in the the same array dimensions as detailed in the calls.

Constructors

There are four possible constructors, these being:

1. FFTWComplex() default constructor that initialises the package with default settings and loads the native interface library.
2. FFTWComplex(boolean systemWisdom) as default constructor but also loads the system-wide *wisdom* file, advanced section for details.
3. FFTWComplex(File wisdomFile), as default constructor but also loads personal *wisdom* file by File.
4. FFTWComplex(String wisdomFile), as default constructor but also loads personal *wisdom* file by file name.

Most users will either use default constructor, or will load system wide *wisdom* file, if it is available.

One-Dimensional Complex FFTs

There are two possible method to take one dimensional FFT.

```
double[] oneDimensional(double data[], int dirn, boolean overwrite)
```

for *interleaved* data array, where,

- `data` is a one-dimensional *interleaved* `double` array that contains the *complex* data with real and imaginary parts in adjacent elements.
- `dirn` integer to specify the direction of transform, `FFTW.FORWARD` for forward transform, `FFTW.BACKWARD` for inverse. These constants are just 1 and -1 respectively.
- `overwrite` boolean that determines if the FFT will overwrite the input data or will be returned in a new double array without changing the input `data` array.
- returns a `double[]` array containing the FFT of the `data`, again of length $2N$ with the real and imaginary parts in adjacent elements, so a that

  $$F_R(k) = \texttt{data}[2*\texttt{k}] \quad \text{and} \quad F_I(k) = \texttt{data}[2*\texttt{k}+1]$$

  if `overwrite` is `true` this will be in the same array as the input data, so overwriting it, while if `false` it will be in a newly allocated `double[]` array.

The corresponding call for *split* data is:

```
double[][] oneDimensional(double[][] data, int dirn, boolean overwrite)
```

where

1. `data` is a two-dimensioal *split* `double[][]`array with real parts in elements `[0][i]` and the corresponding imaginary part in `[1][i]`.
2. `dirn` direction ± 1 as above.
3. `overwrite` controls overwrite as above.
4. returns a `double[][]` of same size as input containing the FFT in locations,

   $$F_R(k) = \texttt{data}[0][\texttt{k}] \quad \text{and} \quad F_I(k) = \texttt{data}[1][\texttt{k}]$$

   If `overwrite` is true this be the same array as the input and the elements will be overwritten, while if `false` a new `double[][]` will be created and the input array unaltered.

What is really calculates, for a complex samples signal $f(i)$ of length N is, the forward transform given by,

$$F(k) = \sum_{i=0}^{N-1} f(i) \exp\left(-\imath 2\pi \frac{ik}{N}\right)$$

where $\imath = \sqrt{-1}$ and the inverse transform is

$$f(i) = \sum_{k=0}^{N-1} F(k) \exp\left(\imath 2\pi \frac{ik}{N}\right)$$

so the transforms are *not* normalised, so for signal $f(i)$ if you take *forward* followed by *inverse*, the result will be $N f(i)$.

Two-Dimensional Complex FFTs

There is a two method to take two dimensional FFT, being again for *interleaved* and *split* data being

```
double[] twoDimensional(int width, int height, double data[],
                int dirn, boolean overwrite)
```

for *interleaved* data where the parameters are:

- `width` the primary dimension of the two-dimensional data.
- `height` the secondary dimension of the two-dimensional data.
- `data` one-dimensional `double[]` array holding the complex data with real and complex parts in adjacent elements. This array *must* be of length `2*width*height`. For the two-dimensional samples function $f(i,j)$ the values must be located in

  $$f_R(i,j) = \texttt{data}[2*(\texttt{j}*\texttt{width}+\texttt{i})] \quad \text{and} \quad f_I(i,j) = \texttt{data}[2*(\texttt{j}*\texttt{width}+\texttt{i})+1]$$

- `dirn` direction ± 1 as in the one-dimensional case.
- `overwrite` boolean that determines if the FFT will overwrite the input data or will be returned in a new double array without changing the input `data` array.
- returns a `double[]` array containing the FFT of the `data`, again of length `2*width*height` with the real and imaginary parts in adjacent elements, so a that

  $$F_R(k,l) = \texttt{data}[2*(\texttt{l}*\texttt{width}+\texttt{k})] \quad \text{and} \quad F_I(k,l) = \texttt{data}[2*(\texttt{l}*\texttt{width}+\texttt{k})+1]$$

  if `overwrite` is `true` this will be in the same array as the input data, so overwriting it, while if `false` it will be in a newly allocated `double[]` array.

For *split* data format the call is

```
double[][] twoDimensional(int width, int height, double[][] data, int
                    dirn, boolean overwrite)
```

with parameters,

1. `width` and `height` giving width and height of array as above.
2. `data` a two-dimensional of size $[2][\texttt{width}*\texttt{height}]$

   $$f_R(i,j) = \texttt{data}[0][\texttt{j}*\texttt{width}+\texttt{i})] \quad \text{and} \quad f_I(i,j) = \texttt{data}[1][\texttt{j}*\texttt{width}+\texttt{i}]$$

As the the one-dimensional case, there is no normalisation, so that *forward* then *inverse* transform will result in data being scaled by `width*height`.

Three-Dimensional Complex FFTs

There is a single method to take two dimensional FFT, this being:

```
double[] threeDimensional(int width, int height, int depth, double data[],
                          int dirn, boolean overwrite)
```

where the parameters are:

- `width` the primary dimension of the three-dimensional data.
- `height` the secondary dimension of the three-dimensional data.
- `depth` the third dimension of the three-dimensional data.
- `data` one-dimensional `double[]` array holding the complex data with real and complex parts in adjacent elements. This array *must* be of length `2*width*height*depth`. For the three-dimensional samples function $f(i,j,k)$ the values must be located in

$$\begin{aligned} f_R(i,j,k) &= \texttt{data}[2*(\texttt{k}*\texttt{width}*\texttt{height}+\texttt{j}*\texttt{width}+\texttt{i})] \\ f_I(i,j,k) &= \texttt{data}[2*(\texttt{k}*\texttt{width}*\texttt{height}+\texttt{j}*\texttt{width}+\texttt{i})+1] \end{aligned}$$

- `dirn` integer to specify the direction of transform, `FFTW.FORWARD` for forward transform, `FFTW.BACKWARD` for inverse. These constants are just 1 and -1 respectively.
- `overwrite` boolean that determines if the FFT will overwrite the input data or will be returned in a new double array without changing the input `data` array.
- returns a `double[]` array containing the FFT of the `data`, again of length `2*width*height*depth` with the real and imaginary parts in adjacent elements, so a that

$$\begin{aligned} F_R(k,l,m) &= \texttt{data}[2*(\texttt{m}*\texttt{width}*\texttt{height}+\texttt{l}*\texttt{width}+\texttt{k})] \\ F_I(k,l,m) &= \texttt{data}[2*(\texttt{m}*\texttt{width}*\texttt{height}+\texttt{l}*\texttt{width}+\texttt{k})+1] \end{aligned}$$

  if `overwrite` is `true` this will be in the same array as the input data, so overwriting it, while if `false` it will be in a newly allocated `double[]` array.

As the the one-dimensional case, there is no normalisation, so that *forward* then *inverse* transform will result in data being scaled by `width*height*depth`.

`FFTWReal` Class

The extending class take real FFT where the data is packed in one-dimensional `double[]` arrays.

Constructors

There are four possible constructors, these being:

1. `FFTWReal()` default constructor that initialises the package with default settings and loads the native interface library.

2. `FFTWReal(boolean systemWisdom)` as default constructor but also loads the system-wide *wisdom* file, advanced section for details.

3. `FFTWReal(File wisdomFile)`, as default constructor but also loads personal *wisdom* file by `File`.

4. `FFTWReal(String wisdomFile)`, as default constructor but also loads personal *wisdom* file by file name.

Most users will either use default constructor, or will load system wide *wisdom* file, if it is available.

Taking FFT of real-only data is much more complicated than for complex data since although the real space data is real, the Fourier space data is complex with Hermition symmetry. Since the Fourier transform is unitary then we should have the same amount of data in real and Fourier space, however due to the packing scheme in FFTW the Fourier space representation take *slightly* more space. This can be overcome in C by over-allocation of memory in real space, but with JAVA strict array-bound checking, this is much more difficult. Therefore this interface does *not* implement *in-place* real transforms but always returns a the transform results in newly allocated memory. This is more expensive in memory allocation, but from initial timing, does not appear to have a significant computational penalty[9].

One-Dimensional Real FFT

There are two methods, one for *forward* transforms and one for *inverse*. The forward transform is taken by

```
double[] oneDimensionalForward(double realArray[])
```

where the parameters are

- `data` one-dimensional `double[]` array holding the data one data point per element, so for real signal $f(i)$ then $f(i) = \texttt{data[i]}$, where for N samples the length of the array $\texttt{data.length} = N$.

- returns a one-dimensional array of containing $N/2+1$ complex components with real and imaginary parts in adjacent elements.

  $$F_R(k) = \texttt{return}[2*\texttt{k}] \quad \text{and} \quad F_I(k) = \texttt{return}[2*\texttt{k}+1]$$

  for $k = 0,\ldots,N/2$, so for N even, there is an extra component than would be expected, see below for explanation.

  This returned array is in a different location that the `data` array which in unaffected by the transform.

The inverse transform is taken by

```
double[] oneDimensionalBackward(double complexArray[])
```

[9] It is considerably faster to take real transforms than to consider real data as being complex with the imaginary part being zero.

where the parameters are

- `complexArray` which is a one-dimensional `double[]` array consisting of $N/2+1$ complex pairs with real and imaginary parts in adjacent elements holding the Hermition Fourier transform data. The length of this array is thus $N+2$ elements.

  See `oneDimensionalForward` for the exact format of this array. It is *usual* to use this method to *inverse* transform data that was forward transformed by `oneDimensionalForward`, if this is not the case, read and understand the symmetry properties of real Fourier transforms *very carefully* before trying to use this.

- return the inverse Fourier transform in a `double[]` array of length N. This array is in a different memory location than `complexArray` which in unaffected by this transform.

Again for efficiency there is no normalisation with respect to the sample lengths, so *forward* followed by *inverse* will result in data of $N\,f(i)$ where N is the number of samples.

Symmetry of One-Dimensional Real Fourier Transforms

We know that the Fourier transform of a real signal is Hermition symmetry, so for real $f(i)$ then

$$F_R(-k) = F_R(k) \quad \text{and} \quad F_I(-k) = -F_I(k)$$

but the discrete Fourier transform as also cyclic of period N, the length of the transform, so we have that,

$$F_R(N-k) = F_R(k) \quad \text{and} \quad F_I(N-k) = -F_I(k)$$

so to form data in the range $0,\ldots,N-1$, then we need $F_R()$ and $F_I()$ to be in the range $0,\ldots,N/2$ inclusive, so each having $N/2+1$ elements. This initially looks wrong, but if we also note that $F(0)$ and $F(N/2)$ are *real only*[10], then we see we have $N/2+1$ real values and $N/2-1$ imaginary values that depend on $f(i)$, so making N values with two *stray* zeros.

Clearly it is possible to relocate the $F_R(N/2)$ element to the $F_I(0)$ location, which is always zero, so giving $N/2$ complex elements as expected, but this is somewhat *unnatural* since the first complex element of the Fourier array is a rather odd mixed term. Given this issue, the authors of FFTW have opted to return the Fourier transform of real signal of N samples as a complex array of $N/2+1$ with two of the imaginary values always zero. This does however mean that the real space and Fourier space take up different amount of array space, thus this problems with with *in-place* transforms noted above.

Two-Dimensional Real FFT

As with the one-dimensional case, there were are two method, one for *forward* and one for *inverse* transforms. The forward transform is,

```
double[] twoDimensionalForward(int width, int height,
                               double realArray[])
```

where the parameters are,

[10] since $\sin(0) = \sin(\pi) = 0$

- `width` primary dimension of the two-dimensional array.
- `height` secondary dimensional of the two-dimensional array.
- `realArray` one-dimensional `double[]` array of length `width*height` containing the data with for sampled image $f(i,j)$ located at

  $$f(i,j) = \texttt{realArray}[\texttt{j}*\texttt{width}+\texttt{i}]$$

  with one sample per element.
- return is a `double[]` array containing complex samples in adjacent pairs with `wft*height` samples where $\texttt{wft} = \texttt{width}/2+1$.

  $$F_R(k,l) = \texttt{return}[2*(\texttt{l}*\texttt{wft}+\texttt{k})] \quad \text{and} \quad F_I(k,l) = \texttt{return}[2*(\texttt{l}*\texttt{wft}+\texttt{k})+1]$$

  where $\texttt{k} = 0,\ldots,\texttt{width}/2$ inclusive and $\texttt{l} = 0,\ldots,\texttt{height}-1$.

The corresponding *inverse* transform is

```
double[] twoDimensionalBackward(int width, int height,
                                double complexArray[])
```

where the parameters are

- `width` primary dimension of the two-dimensional data in *real space*.
- `height` secondary dimensional of the two-dimensional data in *real space*.
- `complexArray` one-dimensional `double[]` array containing the Hermition complex transform data as returned by `twoDimensionalForward` with `wft*height` complex samples where $\texttt{wft} = \texttt{width}/2+1$.
- return a `double[]` array of size `width*height` being the two dimensional real space function with one samples per element located at

  $$f(i,j) = \texttt{realArray}[\texttt{j}*\texttt{width}+\texttt{i}]$$

Again, for efficiency there is no build-in normalisation in either transform.

Symmetry of Two-Dimensional Fourier Transforms

Here we have Hermition symmetry in two-dimensions, so for a real $f(i,j)$ of size $M \times N$ then we have

$$\begin{aligned} F_R(-k,-l) &= F_R(k,l) \\ F_R(-k,l) &= F_R(k,-l) \\ F_I(-k,-l) &= -F_I(k,l) \\ F_I(-k,l) &= -F_I(k,-l) \end{aligned}$$

but $F(k,l)$ is now cyclic of period M in the k direction and N in the l directions, so we have that

$$\begin{aligned} F_R(M-k,N-l) &= F_R(k,l) \\ F_R(M-k,l) &= F_R(k,N-l) \\ F_I(M-k,N-l) &= -F_I(k,l) \\ F_I(M-k,l) &= -F_I(k,N-l) \end{aligned}$$

so as for the one-dimensional case we only need *half* the Fourier data, with the other *half* given by the symmetry conditions. The *half* returned by FFTW is

$$F(k,l) \quad \text{for } k = 0,\ldots,M/2 \text{ and } l = 0,\ldots,N-1$$

so for M being even, then a total of $(M/2+1)N$ complex points. The reason for the apparent *extra* points are exactly as explained for the one-dimensional case, where here $F(0,l)$ and $F(M/2,l)$ are always real only, but are both returned with zero imaginary parts rather than the packing the two real parts into one complex element. This makes unpacking, and processing the Fourier data *slightly!* easier.

Three-Dimensional Real FFT

Following on the three-dimensional case, there were are two method, one for *forward* and one for *inverse* transforms. The forward transform is,

```
double[] threeDimensionalForward(int width, int height,
                depth, double realArray[])
```

where the parameters are,

- `width` primary dimension of the three-dimensional array.
- `height` secondary dimensional of the three-dimensional array.
- `depth` the third dimension of the three-dimensional array.
- `realArray` one-dimensional `double[]` array of length `width*height*depth` containing the data with for sampled image $f(i,j,k)$ located at

  $$f(i,j,k) = \texttt{realArray}[\texttt{k}*\texttt{width}*\texttt{heightj}*\texttt{width}+\texttt{i}]$$

  with one sample per element.
- return is a `double[]` array containing complex samples in adjacent pairs with `wft*height`*depth samples where $\texttt{wft} = \texttt{width}/2+1$.

  $$F_R(k,l,m) = \texttt{return}[2*(\texttt{m}*\texttt{wft}*\texttt{height}+\texttt{l}*\texttt{wft}+\texttt{k})]$$

  $$F_I(k,l,m)) = \texttt{return}[2*(\texttt{m}*\texttt{wft}*\texttt{height}+\texttt{l}*\texttt{wft}+\texttt{k})+1]$$

  where $\texttt{k} = 0,\ldots,\texttt{width}/2$ inclusive and $\texttt{l} = 0,\ldots,\texttt{height}-1$.

The corresponding *inverse* transform is

```
double[] threeDimensionalBackward(int width, int height,
            int depth, double complexArray[])
```

where the parameters are

- `width` primary dimension of the three-dimensional data in *real space*.
- `height` secondary dimensional of the three-dimensional data in *real space*.
- `depth` third dimensional of the three-dimensional data in *real space*.
- `complexArray` one-dimensional `double[]` array containing the Hermition complex transform data as returned by `twoDimensionalForward` with `wft*height*depth` complex samples where $\texttt{wft} = \texttt{width}/2+1$.
- return a `double[]` array of size `width*height` being the two dimensional real space function with one samples per element located at

$$f(i,j,k) = \texttt{realArray}[\texttt{k}*\texttt{width}*\texttt{height}+\texttt{j}*\texttt{width}+\texttt{i}]$$

Again, for efficiency there is no build-in normalisation in either transform.

Wisdom Files and Information

FFTW *wisdom* is information on the optimal FFT calculation scheme to use on the particular computer for the particular size of FFT being performed. The optical calculation method will depend on the details of the local computer, being processor type, memory size available, memory bandwidth, disc speed, and even, on a multi-user system, what other jobs are running. For each system a system wide average *wisdom* is normally located in a text file being `/etc/fftw/wisdom`[11]. This file contains information about all the standard sized one and two dimensional FFTs. Similar special *wisdom* files can be generated anywhere and loaded. The management methods are:

- able to load either system of special *wisdom* file with the constructor parameters.
- `boolean loadWisdom()` loads the current system wisdom file. Returns `true` is successful.
- `boolean loadWisdom(File file)` loads a specified wisdom file by `File`. Returns `true` is successful.
- `boolean loadWisdom(String wisdom)` Loads *wisdom* from a JAVA STRING. Returns `true` is successful.
- `void clearWisdom()` removes all *wisdom* information and deallocated the memory space it was using.
- `boolean exportWisdom(File file)` exports the current loaded *wisdom* information to a `File`. Returns `true` is successful.

[11] Normally created by `fftw-wisdom`, an application supplied with FFTW

- `String getWisdom()` gets the current loaded *wisdom* as a `String`.
- `static String readWisdom(File file)` reads a *wisdom* file into a `static String`.
- `static boolean writeWisdom(String wisdom, File file)` write a *String* containing] *wisdom* information to a output `File`. Returns `true` is successful.

6 `ArrayUtil` class

The `ArrayUtils` class is a series of static methods to manipulate `double[]` arrays in the formats used by `jttfw`. These utilities are used extensively within `DataArray`, but are also very useful if you want to use the lower level `FFTW` class methods.

Interleave and split methods

These methods convert between interleaves array where complex data is held in adjacent elements of a one-dimensional array, and split arrays where real and imaginary data is held in separate arrays.

1. `static double[] interleave(double[] real, double[] imag)` takes two `double[]` arrays containing real and imaginary parts and returns a new interleaved array with real and imaginary values in adjacent elements. If `imag` is either `null` or of length 0, it is treated as an array on zeros.
2. `static double[] interleave(double[][] data)` takes two-dimensional split array with `[0][i]` holding the real value and `[1][i]` imaginary value of the i^{th} element and returns a new interleaved array of length `2*data[0].length`.
3. `static double[][] split(double[] data)` split an interleaved `double[]` array into two split array, returned as a `double[2][]` array with `[0][i]` holding the real values and `[1][i]` imaginary.
4. `static double[][] split(double[] real, double[] imag)` forms a two-dimensional split array from two array of real and imaginary parts. If `imag` is `null` or of length 0, then it is assumed to be zero. Note the input arrays will *not* be copied to a new memory space.

Getting and Setting `Complex` elements

The following `static` methods allows the various array formats to be accessed using the `Complex` class. The available methods are:

1. `static Complex getComplex(double[] data, int i)` get the i^{th} complex element of an interleaves array where the real and imaginary parts are held in adjacent elements.
2. `static void setComplex(double[] data, int i, Complex z)` set the i^{th} of an interleaved array with specified `Complex`.
3. `static Complex getComplex(double[] real, double[] imag, int i)` get the i^{th} complex element from a split array with the real and imaginary parts held in two separate arrays.
4. `static void setComplex(double[] real, double[] imag, int i, Complex z)` set the i^{th} of a split array with real and imaginary in two separate arrays with specified `Complex`.

5. `static Complex getComplex(double[][] split, int i)` get the i^{th} complex element from a split array with the real and imaginary parts held in a two-dimensional split array.

6. `static void setComplex(double[][] split, int i, Complex z)` set the i^{th} of a split array with real and imaginary in a two-dimensional split arrays with specified `Complex`.

Whole Array Manipulations

There are a limited range of method to perform whole array manipulations all of which use direct array access with minimum of setter/getter overhead.

1. `static void conjugate(double[] data)` forms complex conjugate of interleaved array by negating imaginary parts.

2. `static void conjugate(double[][] split)` forms complex conjugate of split array.

3. `static void mult(double[] data, double a)` multiplies interleaved array by a scalar.

4. `static void mult(double[][] split, double a)` multiplies a split array by a scalar.

5. `static void mult(double[] data, double a, double b)` multiplies an interleaved array by a complex specified as two doubles.

6. `static void mult(double[] data, Complex c)` multiplies an interleaves array by a `Complex`.

7. `static void mult(double[][] split, double a, double b)` multiplies a split array by a complex specified as two doubles.

8. `static void mult(double[][] split, Complex c)` multiplies a split array by a `Complex`.