Note 157: The AIPS++ FFTServer Class

How to use the FFTServer class - definitions and tutorial

Anthony G. Willis

1 Introduction

The AIPS++ `FFTServer` class provides application programmers with a very flexible tool for doing one, two and three dimensional Fast Fourier Transforms. This document describes the public methods for this class and how to use them.

This class does in-place Fourier transforms, but methods are provided which save the input array and create a new output array for those programmers who do not want their input data destroyed.

The actual FFT routines used by the `FFTServer` tool are written in FORTRAN and use the widely available public domain NCAR FFTPACK code available from Netlib.

2 Frequency Domain Data Structures

The final goal of the radio astronomer is usually to obtain some type of image of a piece of the sky. When we Fourier Transform an image composed entirely of real numbers (an example would be an image of the Stokes parameter I) to the frequency, or UV domain, we find that the UV grid is Hermetian: i.e. a complex data point at -U, -V is the complex conjugate of the data point at +U, +V. Consequently we only need to sample half the UV grid in order to obtain information about the entire UV domain. However, if we have a complex image (e.g. the Stokes combination Q + i U) its Fourier Transform is not Hermetian, and we must sample the full UV domain.

2.1 Hermetian UV grids

Since we only need to sample half the UV domain in order to obtain complete information about the properties of a hermetian UV grid we store the data for a Hermetian grid differently from that of a non-Hermetian UV grid. This section describes the layout of a Hermetian UV grid that is currently used by the AIPS++ classes `FFTServer` and `GridTool`. We will describe the format using a 2-dimensional UV grid as an example. This grid will be transformed to create a real image of size M x N. The format of the uv grid is shown in Figure 1.

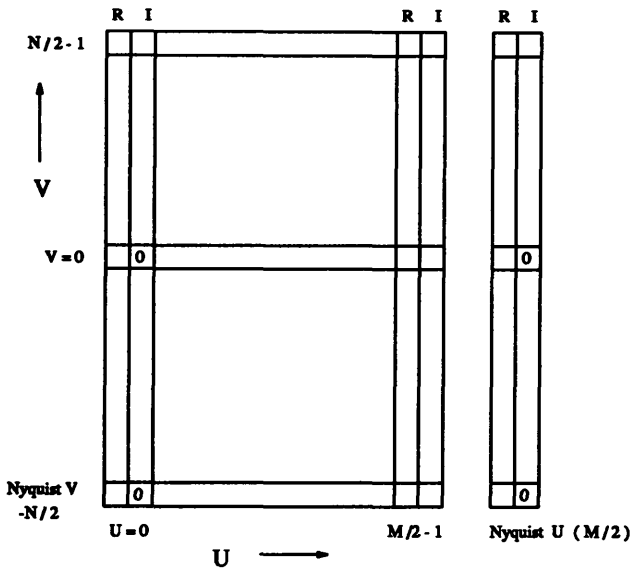


Figure 1: The UV Grid

The U frequency values are sampled over the range U = 0 to U = M / 2. Here M/2 is the Nyquist frequency in U. Since each sample point has both a real and an imaginary value, we require M / 2 + 1 complex numbers along this axis, or 2 * (M / 2 + 1) floating point numbers consisting of real/imaginary pairs. We do not need to sample in the range - (M / 2 - 1) to -1 because of the hermetian symmetry of a uv frequency domain containing Fourier components of a real image.

The V frequency values are sampled over the range V = - N / 2, ..., -1, 0, 1, ..., N /2 - 1. Here, - N / 2 is the Nyquist frequency in V. Therefore we have a total of N sample points along this axis.

The dimensions given above show that if, for example, we are doing an FFT to create a 1024 x 1024 real image, the required size of the uv grid is actually 1026 x 1024. However, as the figure suggests, we can split the grids up into two sub grids, one containing U data points up to M /2 -1, and a separate array containing the Nyquist U point at M / 2. That way most of the uv domain data can be carried around in an array which is the same size, for example 1024 x 1024, as that of the final real image.

Although Figure 1 formally represents a 2 dimensional frequency grid, note that the frequency components of a 1 dimensional complex vector in u space can be represented by by the row of data situated at V = 0. A three dimensional cube with third dimension size Z, would require a series of grids like that shown for each of the sample points -Z / 2, ..., -1, 0, 1, ..., Z /2 - 1.

Normally, any complex arrays that you create from sampled data should not have any values at the Nyquist frequency. If you do have data at the Nyquist frequency, this is an indication that you probably have not set up your data sampling interval correctly. For example, if you are creating

uv grids with the **gridtool**, it will warn you when data is placed in the uv grid at the Nyquist frequency.

An object of class **FFTServer** contains a private internal array which is used to store the extra column of Nyquist data. However, we can still transport the Nyquist data along with the main body of uv data by storing the complex data in a "packed" format. This "packed" format makes use of the fact that the imaginary data elements at zero frequency and at the Nyquist frequency all have value 0. Thus, for example, in the case of our two dimensional uv plane, we can store the real values of the data at (Nyquist U, 0) in the imaginary location at frequency (0,0) and we can store the real value at (Nyquist U, Nyquist V) in location (0, Nyquist V). The off-axis data points along the U = 0 and U = Nyquist axes have conjugate symmetry so we can store the (U = Nyquist, V > 0) data in the locations (U = 0, V > 0), but later compute the overwritten data values using conjugate symmetry properties. By doing so, we contain all necessary complex data information for the range 0 to Nyquist, in an array the same size as the transformed real image. The packing operations for a two dimensional array are illustrated in Figure 2.

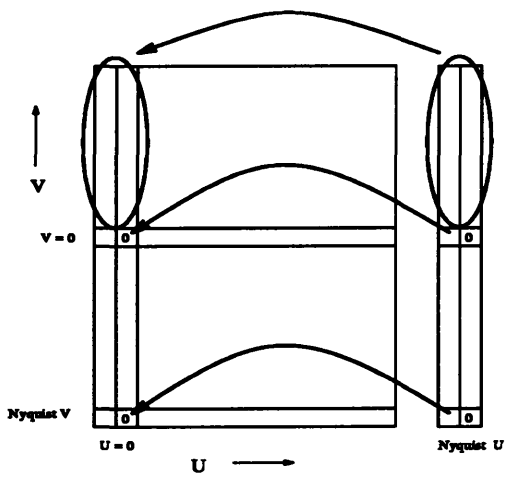


Figure 2: Packing the UV grid

The FFTServer tool expects to get complex frequency data in packed form and unpacks this data just before performing a complex to real transform. The Nyquist data are unpacked and stored in the internal private Nyquist array. This array, along with the unpacked main array containing the majority of the uv data, is passed to the low level FORTRAN routines which perform the fft. The unpacking operations for a two dimensional matrix are illustrated in Figure 3.

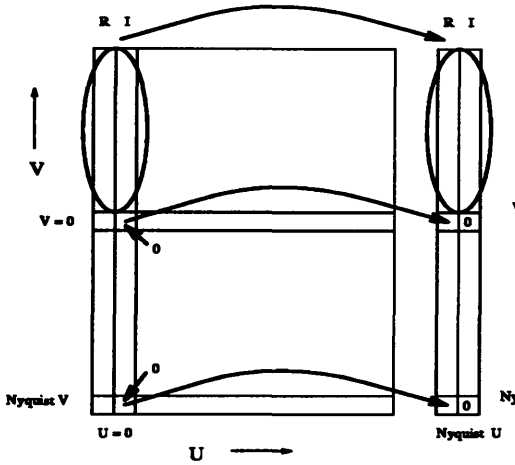


Figure 3a: Unpack - Step 1

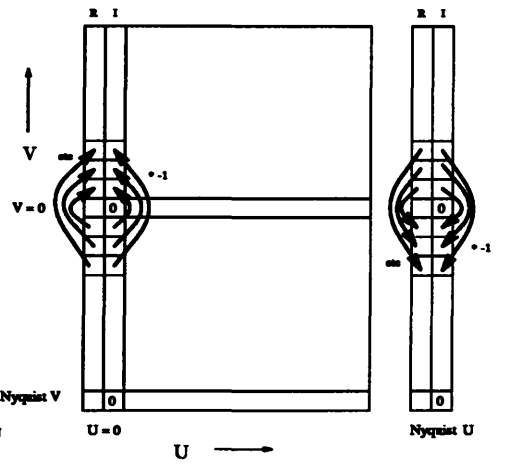


Figure 3b: Unpack - Step 2

Most FFTs expect to get frequency data in a form something like 0, 1, ..., Nyquist, ..., -1. Our uv grid (Figure 1) has its V frequencies ordered as Nyquist, ..., -1, 0, 1, ..., Nyquist - 1. So, if we want to do a Fourier transform from the uv frequency domain to the image domain, we must reorder the uv grid shown in Figure 1 by flipping the top and bottom halves of the main array and the Nyquist array. This operation is done by the **exchangeUV** method described in Chapter 5 and shown in the diagram below.

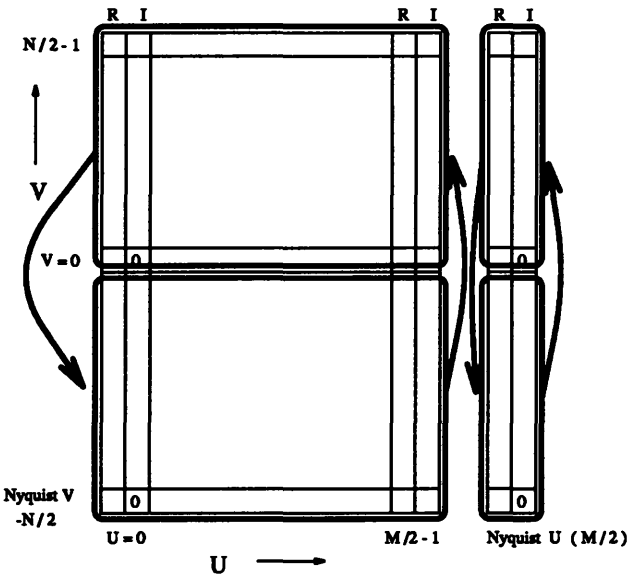


Figure 4: Exchange UV

The real image that emerges from an FFT will have the point that defines the phase reference position situated in the lower left hand corner of the image. However, since radio astronomers normally define the phase reference position to be situated in the centre of the field, we must flip the quadrants of the image that emerges from the fft to obtain a picture that makes sense to most of us. This `flipImage` method which implements this operation is illustrated for the 2 dimensional case in Figure 5. Our particular sequence of operations means that the phase reference pixel in

the image will be located at point M / 2 +1, N / 2 + 1, or 513, 513 for an image with 1024 x 1024 points.

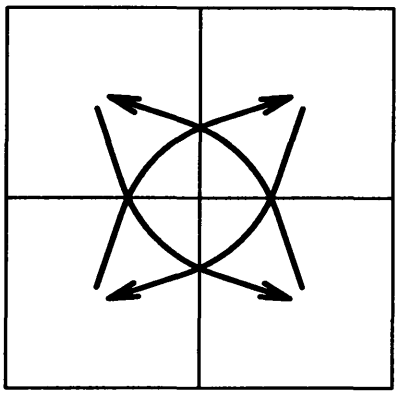


Figure 5: Flip Image

So the steps that occur inside the fft method for going from the frequency domain to the image domain are

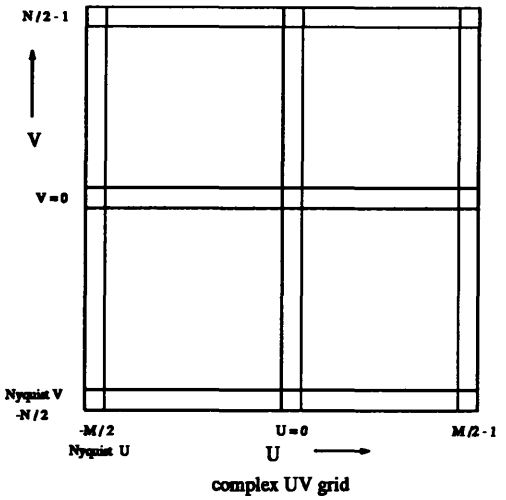
```
- unpack uv frequency data
- exchange the uv data to arrange frequencies in correct order
- do fft from uv to image domain
- flip image that comes out of fft
```

If we want to do an FFT from the image domain to the frequency domain we perform these steps in reverse order.

```
- flip image so phase reference point is first point in to fft
- do fft from image domain to uv frequency domain
- exchange the uv data to arrange frequencies with zero frequencies in
    centre of the uv grid
- pack uv frequency data
```

2.2 Full Complex UV grids

In the case of a complex UV grid without Hermetian symmetry, we must sample and store data covering the full UV domain, not just half of it. In this case, where we will have a final complex sky image with M x N complex numbers, the FFTServer expects to get a UV grid set out as shown in the following figure:



If we flip the quadrants of this UV grid, then the data in U will be ordered from 0 ... M /2 - 1, Nyquist, ... -1 and the data in V will be ordered from 0 ... N /2 - 1, Nyquist, ... -1. As we discussed in the previous section this is the sequence expected by most FFT algorithms. Then we can perform a full complex <> complex FFT to go to the image domain. The complex image that comes out of the FFT has its phase reference position situated in the lower left hand corner of the image just as did the real image described in the previous section. So we must again flip the quadrants of the image to obtain one with its phase reference point at the centre. So the steps that occur inside the full complex fft method for going from the frequency domain to the complex image domain are

```
- flip UV grid so zero frequency data are at the origin
- do fft from complex uv to complex image domain
- flip complex image that comes out of fft
```

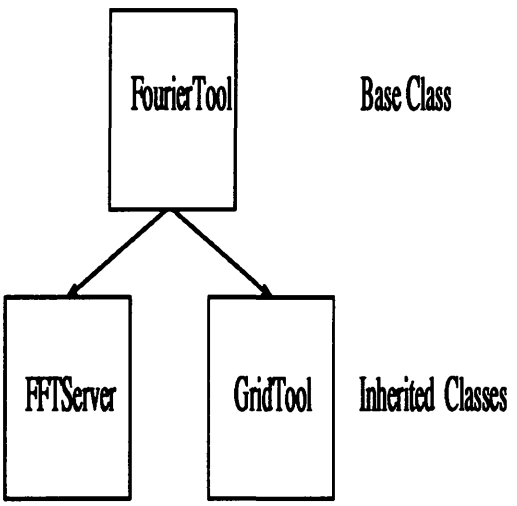
If we want to do an FFT from the complex image domain to the complex frequency domain we perform these steps in reverse order.

```
- flip complex image so phase reference point is first point in to fft
- do fft from complex image domain to uv frequency domain
- flip the quadrants of the uv data to arrange frequencies with
    zero frequencies in centre of the uv grid
```

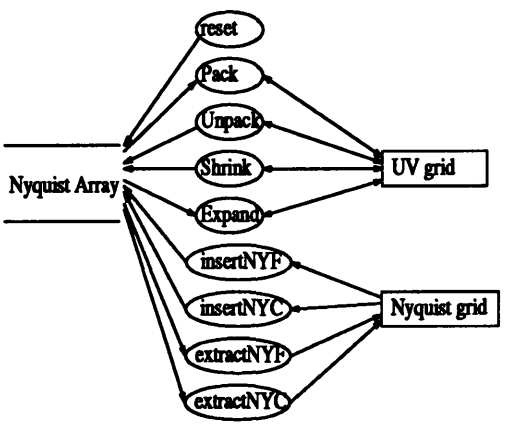
3 Base Class Methods

3.1 Class `FourierTool`

The `FFTServer` class inherits from a base class called `FourierTool`. `FourierTool` is the class which contains the Nyquist data array used in the case of Hermetian UV grids and has methods to handle this array. I suggest that any classes which handle Fourier domain data inherit from this base class so that they have predefined methods to handle the Nyquist array. At present the `FFTServer` and `GridTool` classes inherit from this base class. A diagram of the relationship between the `FourierTool`, `FFTServer` and `GridTool` classes is shown below.



The public class methods of `FourierTool` allow the applications programmer to manipulate the Nyquist U data that are stored inside the Nyquist array that is internal to the class. The functional relationships of the `FourierTool` class are illustrated in the following diagram.



Class FourierTool

3.2 FourierTool Methods

This section gives a detailed description of how to use the methods defined in the **FourierTool** base class. Examples of using these methods are shown in Chapter 6.

- `void pack(Array<T> & uv_grid);`
- `void pack(Array<S> & uv_grid);`

where T can be one of **float** or **double**, and S can be one of **Complex** or **DComplex**. (We actually have two overloaded templated **pack** functions, but obviously, to the applications programmer they look like one function.)

This operation takes the Nyquist data stored internally inside an **FourierTool** object and stores it in the uv array for transport elsewhere.

- `void unpack(Array<T> & uv_grid);`
- `void unpack(Array<S> & uv_grid);`

where T can be one of **float** or **double**, and S can be one of **Complex** or **DComplex**. This operation extracts the Nyquist data packed into a uv array, and stores it internally inside the **FourierTool** object. It then overwrites that part of the uv grid which was being used to store the Nyquist data with the appropriate data from the complex conjugate part of the uv grid.

The figures in the previous chapter illustrate the packing and unpacking operations in a graphical format.

- `void reset();`

This method resets the internal Nyquist array to have a value of zero.

- `const Array<T> & extractNYF();`
- `const Array<S> & extractNYC();`

where T can be one of **float** or **double**, and S can be one of **Complex** or **DComplex**. These methods copy the internal Nyquist array into an external array that can be viewed or manipulated by the programmer.

- `const Array<T> & insertNYF();`

- `const Array<S> & insertNYC();`

where T can be one of `float` or `double`, and S can be one of `Complex` or `DComplex`. These methods copy an externally defined array of data into the the internal Nyquist array. They can be used to insert a Nyquist array back into a `FFTServer` onject after the array has been modified for some reason.

- `void expand(Array<T> &);`
- `void expand(Array<S> &);`

where T can be one of `float` or `double`, and S can be one of `Complex` or `DComplex`.

These methods allow you to attach the contents of the internal Nyquist array on to the end of an unpacked UV array so that the entire range of data for frequencies in U from 0 to Nyquist are stored in one array. The U dimension of the array will be increased by 2 for the case where T is either `float` or `double`, and by 1 for the case S is either `Complex` or `DComplex`.

Note that in order to perform this operation an internal temporary array whose size is that of the final output array will be used.

- `void shrink(Array<T> &);`
- `void shrink(Array<S> &);`

where T can be one of `float` or `double`, and S can be one of `Complex` or `DComplex`.

These methods take an array which contains Nyquist data at the end of the U dimension, copy the Nyquist data into the internal Nyquist array, and then delete the Nyquist data from the input array. The U dimension of the array will be decreased by 2 for the case where T is either `float` or `double`, and by 1 for the case S is either `Complex` or `DComplex`.

Note that in order to perform this operation an internal temporary array whose size is that of the final output array will be used.

An expanded array of this type should NOT be used as an initializer for the FFTServer class.

4 General Purpose Methods

This chapter describes the **FFTServer** methods which will most likely be used by the typical applications programmer. Some more esoteric methods which are not likely to be used very often, or are called internally by the methods described here, are discussed in the next chapter.

- `FFTServer();`

  `FFTServer<T, S> (Array<T> &);`

  `FFTServer<T, S> (Array<S> &);`

  `FFTServer<T, S> (IPosition &);`

where `T` can be one of `float` or `double`, and `S` can be one of `Complex` or `DComplex`. Not all combinations are possible: if `T` is `float` then `S` must be `Complex`. If `T` is `double` then `S` must be `DComplex`.

These are the constructors for the class. The FFTserver must know what size array it is working with so that an internal array for storing Nyquist data is set up with the correct dimensions. Once you have initialized an FFTServer for an array of a particular size, you can use the same FFTServer on other arrays of the same size without having to reinitialize. The constructor also initializes an internal work array with sine and cosine values. The initialization of the work array allows you to save some cpu time if you are going to do a series of one-dimensional FFTs on vectors that are all the same size. The array type used for the initialization can be any of Vector, Matrix, Cube, or generic Array. You can also initialize the FFTServer by means of an IPosition vector which contains the shape of the input or output real image.

Here is a simple example of creating an **FFTServer** object.

```
Matrix<float> abc(128,128);
FFTServer<float,Complex> fft(abc);
```

This example created an object called `fft` which will handle ffts for any real 128 x 128 two dimensional array or any complex 64 x 128 matrix.

An equivalent initialization could be done by

```
Matrix<float> abc(128,128);
IPosition Shape(abc.ndim());
Shape = abc.shape();
FFTServer<float,Complex> fft(Shape);
```

The next example set up an **FFTServer** object for handling a `Complex` array.

```
Matrix<Complex> abc(128,128);
FFTServer fft_complex<float,Complex> (abc);
```

This example creates an object called **fft_complex** which will handle ffts for any 128 x 128 two dimensional Complex array. (Since the internal initialization of the FFTServer mostly employs real, rather than complex numbers, this initialization will also allow you to transform 256 x 128 real arrays)

- **~FFTServer();**

The class destructor. It is currently trivial.

- **void fft(Array<T> & data, int dir);**

This function performs in-place real to complex and complex to real fft's, depending on the value of **dir**.

'dir > 0'	Real to complex fft. The result is returned in **data**. **data** is organised in real and imaginary pairs after the transform.
'dir = 0'	Complex to real fft, with no scaling. Initially, **data** is organised in real/imaginary pairs. Real result is returned in **data**.
'dir < 0'	Complex to real fft, with scaling. Initially, **data** is organised in real/imaginary pairs. The real result is returned in **data**. The scaling factor is 1/(number of elements in **data**).

In this transform, all data are stored in arrays of type **float** or **double**. If the data actually represent complex numbers, then each complex number pair is stored in the array in the sequence real followed by imaginary.

- **Array<S> rcfft(Array <T>& rdata);**

This member performs a real to complex fft. The real data is provided in the array **rdata**, which is preserved. The result is returned in a complex array. This creates a new array containing the complex output. The original real input image is preserved.

Here is an example using this method.

```
Matrix<float> show_how_mat(512,512); show_how_mat = float(0.0);
```

```
show_how_mat(256, 256) = 10.0;       // assign some data to matrix
show_how_mat(299, 349) = 20.0;       //   show_how_mat
FFTServer show_fft<float,Complex> (show_how_mat);
                                     // initialize an FFTServer object
                                     //   show_fft
Matrix<Complex> show_how_uv_mat = show_fft.rcfft(show_how_mat);
                                     // do real to complex fft
```

show_how_uv_mat is a complex array of size 256 x 512, containing the uv data in a packed format.

Since rcfft works with arrays, the same type of operator will work with either vectors, matrices or cubes.

```
Vector<float> show_how_vec(512); show_how_vec = float(0.0);
show_how_vec(256) = 10.0;            // assign some data to vector
show_how_vec(299) = 20.0;            //   show_how_vec
FFTServer show_fft_vec<float, Complex> (show_how_vec);
                                     // initialize an FFTServer object,
                                     //   show_fft_vec
Vector<float> show_how_uv_vec = show_fft_vec.rcfft(show_how_vec);
                                     // do real to complex fft
```

show_how_uv_vec is a complex vector of size 256, containing the frequency data in a packed format. For a one-dimensional vector, this means that the real part of the first complex point is the real value for frequency 0, and the imaginary part of the first complex point is actually the real value for the Nyquist frequency.

- `Array<T> crfft(Array <S> &cdata, int do_scale = -1);`

This member does a complex to real fft. The complex data is provided in the array cdata, which is preserved. A new array containing the real output is created.

Here is an example of this method.

```
Matrix<Complex> show_how_cmplx(256,512); show_how_cmplx = Complex(0.0,0.0);
show_how_cmplx(128, 256) = Complex(10.0, 0.0);
                           // assign some data to matrix show_how_cmplx
show_how_cmplx(149, 349) = Complex(20.0,10.0);
                           //        ditto
FFTServer show_fft_cmplx<float, Complex> (show_how_cmplx);
                           // initialize an FFTServer object, show_fft
Matrix<float> show_how_image = show_fft_cmplx.crfft(show_how_cmplx);
```

```
                                // do complex to real fft
```

show_how_image is a real matrix of size 512 x 512 containing the image.

Since **crfft** works with arrays, the same type of operator will work with either vectors, matrices or cubes.

```
Vector<Complex> show_how_vec(256); show_how_vec = Complex(0.0);
show_how_vec(128) = Complex(10.0,10.0);
                                // assign some data to vector show_how_vec
show_how_vec(58) = Complex(20.0,5.0);
                                //        ditto
FFTServer show_fft_vec<float, Complex> (show_how_vec);
                                // initialize an FFTServer object,
                                //   show_fft_vec
Vector<float> show_how_vector = show_fft_vec.crfft(show_how_vec);
                                // do complex to real fft
```

show_how_vector is a real vector of size 256.

By default, **crfft** will normalize the data before returning it. The scaling factor is 1/(number of elements in **data**). If you do not wish to have automatic scaling, then you must call **crfft**, with a second integer parameter having a value >= 0, eg

```
Vector<float> show_how_vector = show_fft_vec.crfft(show_how_vec, 0);
```

If you are making maps from a uv grid, you would want to avoid this automatic scaling since it is likely that the sum of the weights associated with the uv grid will not correspond to the automatic scaling factor defined above.

- **void cxfft(Array<S> & data, int dir);**

This method does n dimensional (well, up to 3 dimensional) complex to complex in-place FFTs.

'dir > 0'	complex to complex forward (image to UV domain) fft. The result is returned in **data**.
'dir = 0'	complex to complex backward (UV domain to image) fft, with no scaling.
'dir < 0'	complex to complex backward (UV domain to image) fft, with scaling. The scaling factor is 1/(number of elements in **data**).

See Chapter 6 for examples of doing complex to complex FFTs and the relationship between complex images and real images.

- `void nyfft(Array<T> &data, int dir);`

Real to complex and complex to real fft in n-dimensions. The nyquist data is carried in the last two columns of `data`. So if the image size is X * Y * Z, the size of `data` must be (X + 2) * Y * Z. See the description of method `fft` for an explanation of the `dir` parameter.

Since this method expects that Nyquist data is explicitly stored in the last two columns, no packing or unpacking of Nyquist data is done.

- `float scaleFactor(void);`

Return the scale factor used in the most recently performed complex to real fft.

```
float scale_value = show_fft_cmplx.scaleFactor();
```

- `float wtsum(Array<T> &);`

This method sums up the values in a weights array created by the GridTool, and associated with a Hermetian UV grid. Then proper scaling of an image produced from the FFT of a gridded uv plane can be done.

```
Matrix <float> image(1024,1024), wts(513,1024);
FFTServer fft<float, Complex> (image);
   ...   grid uv data into into image with aid of gridtool, etc
   ...   the gridtool will assign weight values to the
           weight array, wts
fft.fft(image,0);                  // FFT uv plane to image domain
                                   // but don't do default normalization
                                   //   scaling is done by dividing
                                   //   by sum of weights
float sum_of_weight = fft.wtsum(wts);
image = image / sum_of_weight;     // normalize fft
```

- `float cxWtsum(Array<T> &);`

This function sums up the weights associated with a full complex grid for FFT normalization.

- `void uvassign(Array<T> &, Array<float> &)`
- `void uvassign(Array<S> &, Array<float> &)`

This method copies uv grid weights associated with a Hermetian UV grid on to a uv grid in preparation for doing a real to complex FFT for an antenna pattern. Packing is also done.

```
image = 0.0;
fft.uvassign(image, wts);          // copy wts array to image array
fft.fft(image, 0);                 // do the fft
image = image / sum_of_weight;     // normalize by sum of weights
   ... voila, an antenna pattern
```

- `void cxUVassign(Array<S> &, Array<float> &)`

This method copies uv grid weights associated with a full complex UV grid on to a uv grid in preparation for doing a complex to complex backward FFT for an antenna pattern.

5 Special Purpose Methods

The following methods are public members of the FFTServer class, but are unlikely to be used by the casual programmer. If needed, they have normally been called internally by functions such as `crfft`.

- `void flipImage(Array<T> &image, int image_type=fftparms::DEF_IMAGE_TYPE);`
- `void flipImage(Array<S> &image, int image_type=fftparms::DEF_IMAGE_TYPE);`

For an n-dimensional image, divide the image into 2^n equal parts and swap each of these parts with that which diametrically opposes it. So, for a matrix, flip quadrants, for a cube, flip octants, etc. If image_type is not equal to fftparms::DEF_IMAGE_TYPE (0) then the array "image" is assumed to be carrying two additional columns for nyquist data.

- `void exchangeUV(Array<T> &image);`
- `void exchangeUV(Array<S> &image);`

Exchanges column uv locations for FFT so that data fed to FFT are in frequency order 0 ... N/2 -1, Nyquist ... -1.

6 Some Detailed Examples

Here is a sequence of one dimensional ffts that use the methods described above.

```
cout<<"test in-place 1-d fft"<<endl;
Vector<float> oned(16); oned=float(0.0); // declare and initialize
                                         // a real vector of size 16
oned(8) = float(1.0);                    // assign value 1 to central
                                         //   element
cout<<"initial 1 d real array "<<endl;
cout<<oned<<endl;

FFTServer test1d<float, Complex>(oned);  // initialize fft server;
test1d.fft(oned, 1);                     // do real to complex fft
cout<<"fft'd 1 d complex array "<<endl;
cout<<oned<<endl;

test1d.fft(oned, -1);              // do reverse complex to real fft
cout<<"fft'd 1 d final real array "<<endl;
cout<<oned<<endl;

cout<<"test non in-place 1-d fft"<<endl;
Vector<Complex> result1d = test1d.rcfft(oned);
                                         // do real to complex fft
cout<<"fft'd 1 d complex array "<<endl;
cout<<result1d<<endl;
test1d.unpack(result1d);                 // Unpack Nyquist data
cout<<"fft'd 1 d unpacked complex array "<<endl;
cout<<result1d<<endl;
Array<Complex> NyquistC = test1d.extractNYC();
                                         // Extract Nyquist data
cout<<"fft'd 1 d Nyquist data "<<endl;
cout<<NyquistC<<endl;                    // display it

test1d.pack(result1d);                   // repack the Nyquist data
Vector<float> reverse1d = test1d.crfft(result1d);
                                         // do complex to real fft
cout<<"fft'd 1 d real array "<<endl;
cout<<reverse1d<<endl;
```

This code produces the result

```
test in-place 1-d fft
initial 1 d real array
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

fft'd 1 d complex array
```

```
[1, 1, 1, 0, 1, -0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0]

fft'd 1 d final real array
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

test non in-place 1-d fft
fft'd 1 d complex array
[(1, 1), (1, 0), (1, -0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]

fft'd 1 d unpacked complex array
[(1, 0), (1, 0), (1, -0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]

fft'd 1 d Nyquist data
[(1, 0)]

fft'd 1 d real array
[0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

The following code shows some manipulations of a 2 dimensional array

```
#include <iostream.h>
#include <aips/FFTServer.h>
#include <aips/Complex.h>
#include <aips/ArrayIO.h>
main()
//
// test 2-dimensional Fourier transform stuff
//
{
try {
Matrix<float> array(8,8); array=0;        // declare an array in the
                                          //      image domain
array(2,2) = 1.0;                         // stick in some point sources
array(3,3) = 1.0;
array(4,4) = 1.0;
cout <<"initial test array\n";
cout<<array<<endl;cout.flush();           // display initial array
IPosition Shape(2);
Shape = array.shape();
FFTServer<float, Complex> testfft(Shape); // initialize an FFTServer object
testfft.fft(array,1);                     // FFT to UV domain
cout <<"fft'd array in packed form\n";
cout << array <<"\n";                     // display UV array in packed form
testfft.unpack(array);                    // get UV array into unpacked form
Matrix<float> Nyquist = testfft.extractNYF();  // Extract Nyquist data
cout <<"Nyquist array\n";
cout<<Nyquist<<endl;cout.flush();         // display Nyquist data
cout <<"unpacked array\n";
```

```
cout <<array<<endl;cout.flush();          // display unpacked array
testfft.expand(array);                    // add Nyquist data to end of
                                          //     unpacked array
cout <<"expanded array\n";
cout << array <<endl;cout.flush();        // display expanded array
testfft.nyfft(array,-1);                  // use FFT nyfft method to
                                          //     transform expanded array
                                          //     back to image domain
cout <<"output from nyfft (image domain)\n";
cout << array <<endl;cout.flush();        // display result of nyfft
                                          //     note 2 extra rows
testfft.nyfft(array,1);                   // fft back to Fourier domain

testfft.shrink(array);                    // delete Nyquist data from end
                                          //     of unpacked array
cout <<"shrunken array\n";
cout <<array<<endl;cout.flush();          // display shrunken array
testfft.pack(array);                      // pack array for reverse FFT
cout <<"packed array\n";
cout <<array<<endl;cout.flush();          // display packed array
testfft.fft(array,-1);                    // FFT back to image domain with
                                          //     scaling
cout <<"reverse fft'd array\n";
cout << array <<"\n";                     // display final result; this better
                                          //     equal what we started with!
} catch (AipsError x) {
    cout << "Caught exception at line " << x.thrownLine()
     << " from file " << x.thrownFile() << endl;
    cout << "Message is: " << x.getMesg() << endl;
  } end_try;
}
```

Running this program should produce output similar to that which follows. (NOTE: the output has been slightly idealized; numerical roundoff in particular implementations of an FFT will result in actual runs producing small numbers close to but not precisely zero, such as -4.37114e-08.) The X(U) axis is along the columns (down the page) while the Y(V) axis is along the rows (across the page).

```
initial test array
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 1, 0, 0, 0, 0, 0
 0, 0, 0, 1, 0, 0, 0, 0
 0, 0, 0, 0, 1, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0]
```

```
fft'd array in packed form
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[1, 0.292893, 0, 1.70711, 3, 0.292893, 0, 1.70711
 3, 0.292893, -1, -1.70711, 1, 0.292893, -1, -1.70711
 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893, 1
 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893, 0
 0, 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893
 -1, -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893
 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893, 0
 -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893, -1]

Nyquist array
Axis Lengths: [2, 8]  (NB: Matrix in Row/Column order)
[3, 1.70711, 0, 0.292893, 1, 0.292893, 0, 1.70711
 0, 1.70711, 1, -0.292893, 0, 0.292893, -1, -1.70711]

unpacked array
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[1, 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893
 0, 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893
 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893, 1
 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893, 0
 0, 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893
 -1, -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893
 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893, 0
 -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893, -1]

expanded array
Axis Lengths: [10, 8]  (NB: Matrix in Row/Column order)
[1, 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893
 0, 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893
 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893, 1
 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893, 0
 0, 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893
 -1, -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893
 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893, 0
 -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893, -1
 3, 1.70711, 0, 0.292893, 1, 0.292893, 0, 1.70711
 0, 1.70711, 1, -0.292893, 0, 0.292893, -1, -1.70711]

output from nyfft (image domain)
Axis Lengths: [10, 8]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 1, 0, 0, 0, 0, 0
 0, 0, 0, 1, 0, 0, 0, 0
 0, 0, 0, 0, 1, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
```

```
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0]

shrunken array
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[1, 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893
 0, 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893
 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893, 1
 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893, 0
 0, 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893
 -1, -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893
 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893, 0
 -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893, -1]

packed array
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[1, 0.292893, 0, 1.70711, 3, 0.292893, 0, 1.70711
 3, 0.292893, -1, -1.70711, 1, 0.292893, -1, -1.70711
 0.292893, 0, 1.70711, 3, 1.70711, 0, 0.292893, 1
 0.292893, -1, -1.70711, 0, 1.70711, 1, -0.292893, 0
 0, 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893
 -1, -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893
 1.70711, 3, 1.70711, 0, 0.292893, 1, 0.292893, 0
 -1.70711, 0, 1.70711, 1, -0.292893, 0, 0.292893, -1]

reverse fft'd array
Axis Lengths: [8, 8]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 1, 0, 0, 0, 0, 0
 0, 0, 0, 1, 0, 0, 0, 0
 0, 0, 0, 0, 1, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0
 0, 0, 0, 0, 0, 0, 0, 0]
```

Here are some examples of doing complex to complex FFTs. In the two dimensional case, you can compare equivalent real <> complex and complex <> complex transforms as the imaginary data in the complex image are all set to zero.

```
#include <iostream.h>
#include <aips/FFTServer.h>
#include <aips/Complex.h>
#include <aips/ArrayIO.h>

main()
```

```
{
try {
//
// test complex stuff
//
Vector<Complex> vector(8);                          // create 1 d complex vector
vector = Complex(0.0);
vector(4) = Complex(1.0,0.0);                       // put a signal in the middle
cout<<"initial vector \n"<<vector<<endl;            // show initial vector
FFTServer<float,Complex>  onedfft(vector);          // initialize an FFTServer
onedfft.cxfft(vector,1);                            // transform to UV domain
cout<<"FFTd vector \n"<<vector<<endl;               // show complex UV vector
onedfft.cxfft(vector,-1);                           // go back to image domain
                                                    //   with scaling
cout<<"reverse FFTd vector \n"<<vector<<endl;       // show final result is
                                                    //   same as initial vector

Matrix<Complex> narray(4,4);                        // create 4x4 complex matrix
narray=Complex(0.0);
narray(1,1) = Complex(1.0,0.0);                     // put in some point sources
narray(2,2) = Complex(1.0,0.0);
narray(3,3) = Complex(1.0,0.0);
cout <<"initial 2 d complex test array\n";
cout<<narray<<endl;cout.flush();                    // show initial matrix
FFTServer<float, Complex> ntestfft(narray);         // initialize an FFTServer
ntestfft.cxfft(narray,1);                           // transform to UV domain
cout <<"fft'd complex array \n";                    // show complex UV grid
cout << narray <<"\n";                              //   note that Nyquist data
                                                    //   are in first row
ntestfft.cxfft(narray,-1);                          // go back to image domain
                                                    //   with scaling
cout <<"reverse fft'd complex array \n";            // show final result is
                                                    //   same as initial matrix
cout << narray <<"\n";
Matrix<float> farray(4,4); farray=0.0;              // create a 4 x 4 real matrix
farray(1,1) = 1.0;                                  // put in some point sources
farray(2,2) = 1.0;
farray(3,3) = 1.0;
cout <<"initial 2 d real test array\n";             // show initial matrix
cout<<farray<<endl;cout.flush();
FFTServer<float, Complex> ftestfft(farray);         // create an FFTServer
ftestfft.fft(farray,1);                             // do real to complex fft
ftestfft.unpack(farray);                            // get Nyquist data
ftestfft.expand(farray);                            // put Nyquist data in last
                                                    //   two rows
cout <<"fft'd complex Hermetian array after expansion \n";
cout << farray <<"\n";                              // show expanded Hermetian
                                                    //   UV grid - compare with
                                                    //   full complex one above
ftestfft.shrink(farray);                            // put Nyquist data back in
```

```
                                                    //   Nyquist array
ftestfft.pack(farray);                              // pack, since fft method
                                                    //   expects a packed array
ftestfft.fft(farray,-1);                            // do complex to real fft
                                                    //   with scaling
cout <<"reverse fft'd array \n";                    // show final result is same
                                                    //   as initial matrix
cout << farray <<"\n";
} catch (AipsError x) {
    cout << "Caught exception at line " << x.thrownLine()
     << " from file " << x.thrownFile() << endl;
    cout << "Message is: " << x.getMesg() << endl;
  } end_try;
}
```

Running this program produces

```
initial vector
[(0, 0), (0, 0), (0, 0), (0, 0), (1, 0), (0, 0), (0, 0), (0, 0)]
FFTd vector
[(1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0), (1, 0)]
reverse FFTd vector
[(0, 0), (0, 0), (0, 0), (0, 0), (1, 0), (0, 0), (0, 0), (0, 0)]

initial 2 d complex test array
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[(0, 0), (0, 0), (0, 0), (0, 0)
 (0, 0), (1, 0), (0, 0), (0, 0)
 (0, 0), (0, 0), (1, 0), (0, 0)
 (0, 0), (0, 0), (0, 0), (1, 0)]

fft'd complex array
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[(3, 0), (1, 0), (-1, 0), (1, 0)
 (1, 0), (-1, 0), (1, 0), (3, 0)
 (-1, 0), (1, 0), (3, 0), (1, 0)
 (1, 0), (3, 0), (1, 0), (-1, 0)]

reverse fft'd complex array
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[(0, 0), (0, 0), (0, 0), (0, 0)
 (0, 0), (1, 0), (0, 0), (0, 0)
 (0, 0), (0, 0), (1, 0), (0, 0)
 (0, 0), (0, 0), (0, 0), (1, 0)]

initial 2 d real test array
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0
 0, 1, 0, 0
```

```
 0, 0, 1, 0
 0, 0, 0, 1]

fft'd complex Hermetian array after expansion
Axis Lengths: [6, 4]  (NB: Matrix in Row/Column order)
[-1, 1, 3, 1
 0, 0, 0, -0
 1, 3, 1, -1
 0, 0, 0, 0
 3, 1, -1, 1
 0, -0, 0, 0]

reverse fft'd array
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0
 0, 1, 0, 0
 0, 0, 1, 0
 0, 0, 0, 1]
```