# Note 157: The AIPS++ FFTServer Class

How to use the FFTServer class - definitions and tutorial

**Anthony G. Willis and Philippe McLean**

# 1 Introduction

**Most of the details in this document have been made obsolete by the overhaul of the FFTServer class made in October 1997. It is now recommended you read the class documentation for a description of how to use the FFTServer class.**

The AIPS++ `FFTServer` class provides application programmers with a flexible and efficient tool for doing multi-dimensional Fast Fourier Transforms on AIPS++ Array objects. This document describes the public methods for this class and how to use them. The `FFTServer` class provides methods for in-place and non-destructive FFTs. An in-place FFT modifies its argument in place; a non-destructive FFT returns the transform as a new array, and leaves its argument unchanged. The packing format used to represent Hermetian objects is described, and a description of the implementations of the FFT, packing, and unpacking methods in the `FFTServer` and `FourierTool` classes is provided.

The low-level FFT routines used by the `FFTServer` tool are written in FORTRAN and use the widely available public domain NCAR FFTPACK code available from Netlib. A multi-dimensional FORTRAN FFT routine originally written by R.C. Singleton is also used. Modifications were made to remove static size limits, and a double precision version was written.

This document is intended for readers familiar with discrete and continuous Fourier tranforms, and the use of other FFT implementations.

# 2 Frequency Domain Data Structures

One of the final goals of the radio astronomer is to obtain an image of a piece of the sky. When we apply a Fourier transform to 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 value at the point *(-u, -v)* is the complex conjugate of the data point at *(u, v)*. Consequently we need only sample half the $UV$ grid in order to obtain information about the entire $UV$ domain. In general, the Fourier transform of a real valued function is an Hermetian function. If $f(x_1, x_2, x_3, ..., x_n)$ is a real-valued function of $n$ real variables, then its Fourier transform $F(u_1, u_2, u_3, ..., u_n)$ will be an Hermitian function of $n$ real variables. That is, $F(a, b, c, ..., n) = F^*(-a, -b, -c, ..., -n)$, where $F^*(a, b, c, ..., n)$ is the complex conjugate of $F(a, b, c, ..., n)$. *Bracewell [1].*

The Fourier transform of a complex image (e.g. the Stokes combination *Q + iU)* will in general be complex valued, and will have the same storage requirements as its time domain inverse.

In this document, *grid* refers to an arbitrary n-dimensional `Array` object. The Fourier transform of a time domain sample is referred to as a *frequency domain* grid; a time domain sample is also referred to as an *image.*

## 2.1 Hermetian frequency domain grids

The `FFTServer` class exploits the Hermetian property of frequency domain grids to reduce the required storage by half. For example, in the plane, we form pairs of points *(u,v)* and *(-u,-v)*; the value at one is the complex conjugate of the value at the other, and we can abitrarily choose either to represent both.

We may then represent a complex-valued transform as a sequence of real and imaginary values, in the same amount of space required by time domain inverse. Real and imaginary pairs are stored consecutively; for a multi-dimensional object, real and imaginary pairs are indexed along the first dimension.

Suppose $A$ is an real valued array of size 4, representing two complex numbers. Suppose $U$ is an equivalent complex valued array, of size 2, consisting of four real numbers. Then *A(0)* and *A(1)* are the real and imaginary parts of a single complex number, *U(0)*. Then *A(2)* and *A(3)* represent *U(1)* and *U(-1)*. But what about *U(2)* and *U(-2)*? Well, *U(2)=U(-2)*, since the sample $U$ is periodic with period 4. But *U(4/2)=U(2)* is the Nyquist sample, and there is no room for it in $A$. If $A$ has size 5, then space will be required for the imaginary component of *U(2)*, the last element. Thus another array is required to store this Nyquist component. In general, if $A$ is a multi-dimensional object with dimensions *(a, b, c, ..., n)*, then the Nyquist array will have size *(2, b, c, ..., n)* if *a* is even, and size *(1, b, c, ..., n)* if *a* is odd. To simplify implementation and avoid ambiguity in the representations, the size of the so-called Nyquist array associated with $A$ is *(2, b, c, ..., n)*. Beware that "Nyquist array" is a misnomer; if the first dimension *a* is odd, then there will be no sample at the Nyquist frequency in *a*. Throughout this document, *Nyquist array* will refer to this object.

The association of Hermetian grid and Nyquist array is referred to as the *unpacked* format for an Hermetian grid.

It is possible to reduce this storage size, and pack the Nyquist array into the Hermetian grid, by using the following properties:

1. There will be points whose values are their own conjugates. This property is only true of points where every coordinate is either 0 or *N/2*, where $N$ is the size of the corresponding dimension. Note that *N/2* is the Nyquist frequency. But if a complex number is its own conjugate, then its imaginary part is zero. We may store the real part of another such number in the imaginary part of the first.
2. Consider an array $A$ with dimensions *(a, b, c, ..., n)*. Consider the set $X$ of elements in $A$ whose first coordinate is 0. Then the conjugate $y$ of a point $x$ in $X$ will also be in $X$. If $a$ is even, then the Nyquist array will also have this property, since points in the Nyquist array have an implicit first coordinate of $a/2$, the Nyquist frequency in $a$. Hence we may combine $X$ and the Nyquist array in $X$, and later reconstruct both. If $a$ is odd, then each row in the Nyquist array will contain the real and imaginary components of the last element in the corresponding row in $A$. The real component will be duplicated, for convenience. Thus we need only save the imaginary components of the Nyquist array; these will fit in exactly half of $X$. In either case, we *pack* $X$ and the Nyquist array into $X$, and later *unpack* and reconstruct both.

Thus we may represent an entire complex valued Hermetian frequency domain sample in the same space required by its real valued inverse. This is referred to as a *packed* format.

This section describes the layout of the Hermetian $UV$ grid currently used by the AIPS++ classes `FFTServer`, `FourierTool`, and `GridTool`. We will describe the format using a 2-dimensional $UV$ grid as an example. First we consider the case when $M$ is even. This grid represents the transform of a real image of size *M x N*. The format of the $UV$ grid is shown in Figure 1.

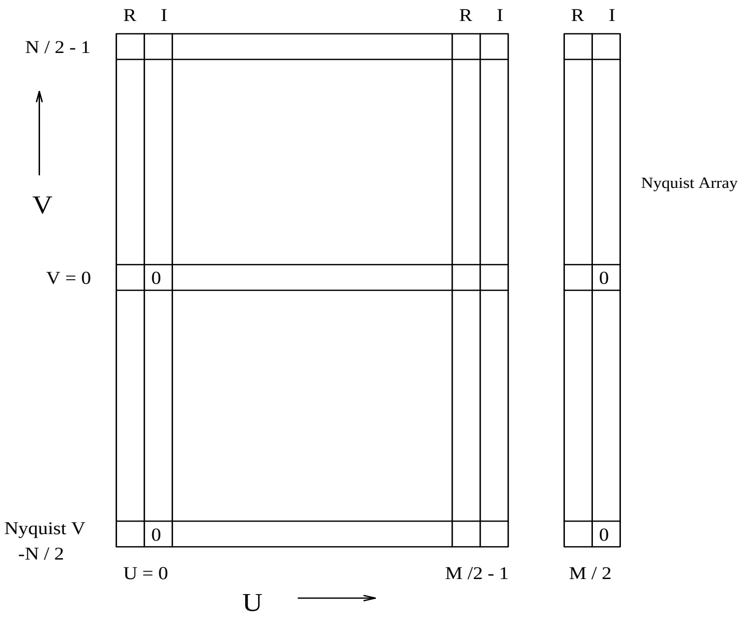



Figure 1: A 2-dimensional UV grid; both M and N are even.

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)* pairs of floating point numbers. We do not need to sample in the range -*(M/2 - 1)* to *-1* because of the Hermetian symmetry of the *UV* frequency domain containing Fourier components of a real image. *R* and *I* denote real and imaginary components.

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.

For example, if we are transforming a 1024 x 1024 real image, then the size of the *UV* grid will actually be 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*. In this way, most of the frequency domain data can be stored in an array the same size as the real-valued time domain image.

This separate array is the *Nyquist array* and is private member object `FourierTool` object.

Now consider the case when both *M* and *N* are both odd.

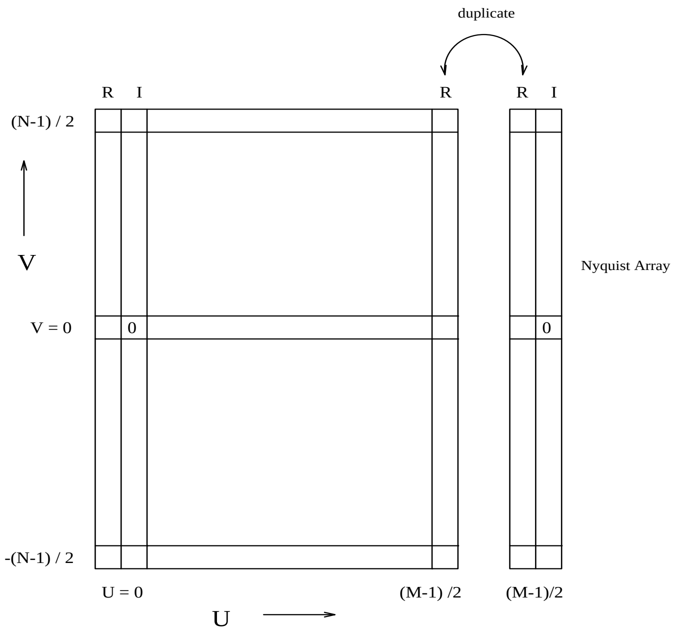



Figure 2: A 2-dimensional UV grid; both M and N are odd.

Since $N$ is odd, there will be no Nyquist component in $V$, and the imaginary component at *(0,-(N-1)/2)* will in general *not* be zero. Since $M$ is odd, there will be no Nyquist component in $U$, and the Nyquist array will be used to store the real and imaginary components of the last elements in each row of the grid. The real element will be duplicated in the first component of each row in the Nyquist array.

Although Figures 1 and 2 describe 2-dimensional grids, the `FFTServer` class can perform FFTs on arbitrary multi-dimensional objects. You should view the grid and Nyquist arrays as a sequence of rows. Each row is a sequence of real/imaginary pairs. If the first dimension is even, then the Nyquist row will contain the real and imaginary components of the Nyquist sample. If the first dimension is odd, then the Nyquist row will contain the real and imaginary components of the last frequency sample. The real component will be duplicated in the last element of the row in the grid.

Normally, any complex arrays that you create from sampled data should not have any values at the Nyquist frequency. A sample at the Nyquist frequency may indicate that your sampling interval is incorrect. `GridTool` will warn you when you place data in a $UV$ grid at the Nyquist frequency.

An `FFTServer` object contains a `private` Array object which is used to store the Nyquist array. We can use the packing technique described above to store the entire Nyquist array in the grid; this allows the user to store the entire frequency domain sample in the same space required by its real valued inverse.

Two different packing techniques are used: one, when the first dimension is even, and the other, when the first dimension is odd. During packing, half of the Nyquist data are copied to the $UV$ grid.

The following figure describes the packing of a 2-dimensional grid when the first dimension $M$ is even:

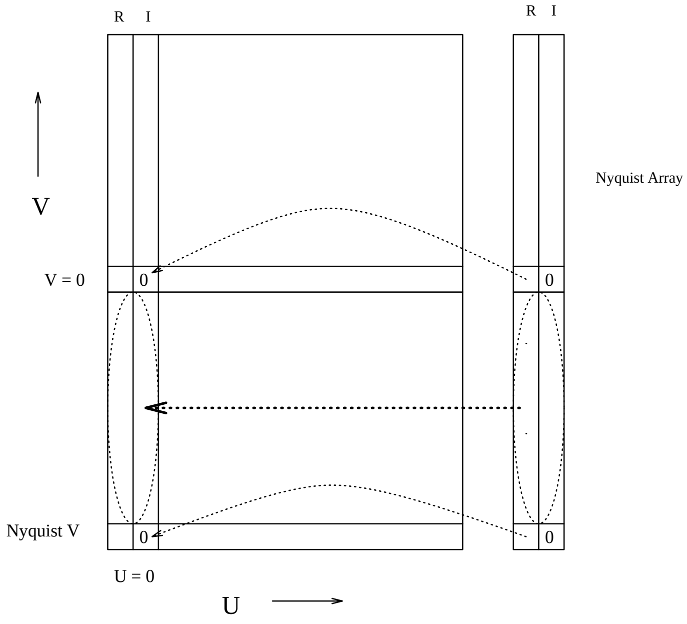



Figure 3: Packing a UV grid when U is even

Both arrays are examined in storage order; AIPS++ `Array` objects are stored in FORTRAN array order. The first index in a FORTRAN multi-dimensional array varies fastest as elements in memory are examined sequentially.

If $M$ is even, then the Nyquist array will have the same symmetry properties as the *U=0* column of the grid, and the imaginary components at the Nyquist and zero frequencies in $V$ will be zero. There we copy the real component from the Nyquist array to the imaginary component of the grid. The other elements are copied from the Nyquist array to the *U=0* column of the grid. The overwritten values can be computed later, by using the fact that the value at *(u, v)* is the complex conjugate of the point at *(-u, -v)*. If $M$ is odd, then the Nyquist array will not have any symmetry properties, but its real component will be duplicated in the last column of the $UV$ grid. Then we need only copy the imaginary components from the Nyquist array to the $UV$ grid.

The following figure describes the packing of a 2-dimensional grid when the first dimension $M$ is odd and the second dimension $N$ is even:

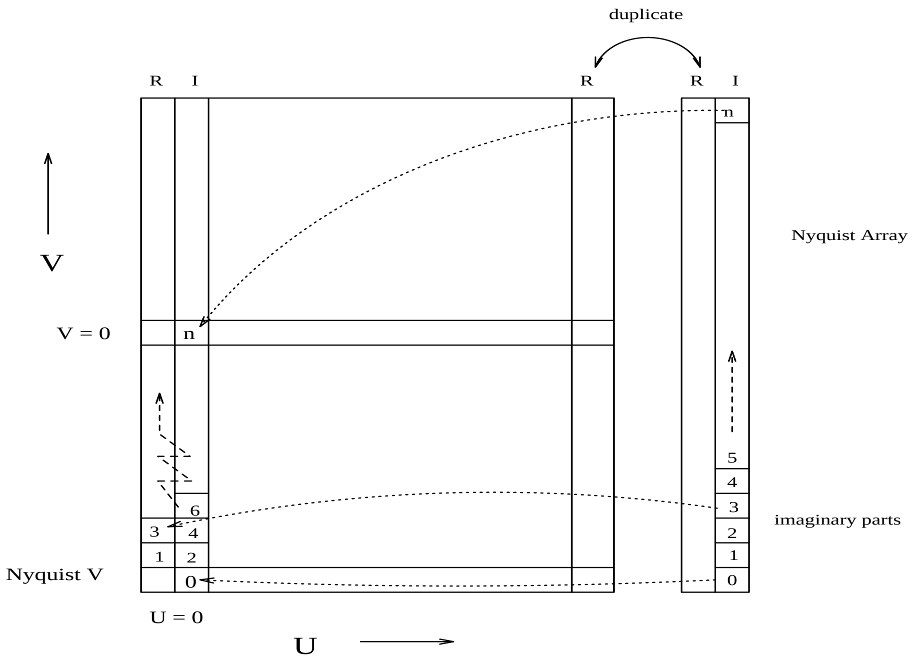



Figure 4: Packing a UV grid when U is odd

The imaginary components of the Nyquist array are copied sequentially, in storage order, to succesive real and imaginary components in the *U=0* column of the *UV* grid. If the element in the *UV* grid is its own conjugate, then we cannot overwrite the real component, so only the imaginary part is used.

The following figure describes the packing of a 2-dimensional grid when both dimensions $M$ and $N$ are odd:

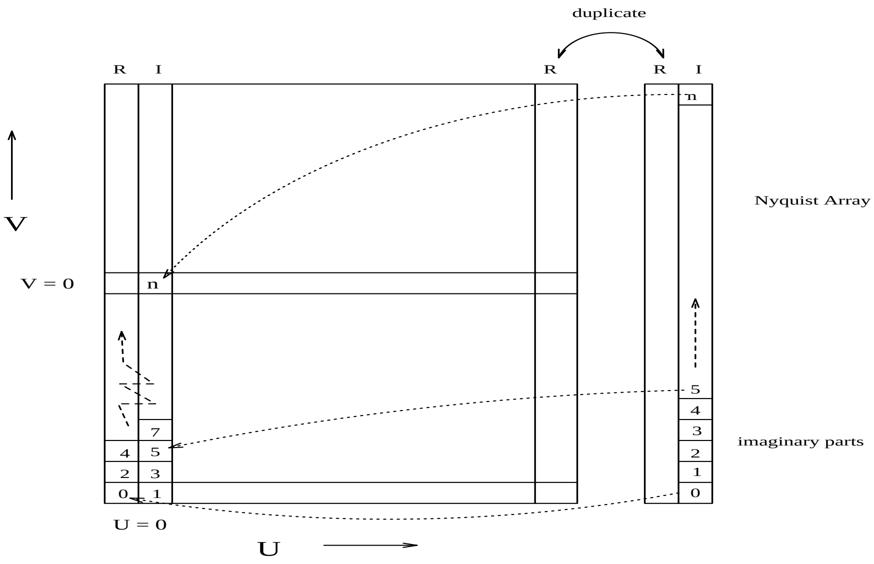



Figure 5: Packing a UV grid when both dimensions are odd

Many of the `FFTServer` methods expect frequency domain data in packed form, and the data are unpacked prior to the transform. The behaviour for each method is specified below. See Chapter 4 [FFTServer methods], page 13, for details.

The first step of unpacking is the reverse of the packing process. Data are copied from the *U=0* column of the *UV* grid to the Nyquist array. The second step recomputes the values that were used to store Nyquist data during packing; the location of an element's conjugate in the grid must be determined. If the first dimension is even, this is performed in both the grid and the Nyquist array. If the first dimension is odd, then the conjugate property is only true for the *U=0* column of the grid. The real components of the Nyquist array are copied from the last element of each row in the *UV* grid.

The computing of complex conjugates during reconstruction is described in the following figure. The figure depicts a *UV* grid with even dimensions.

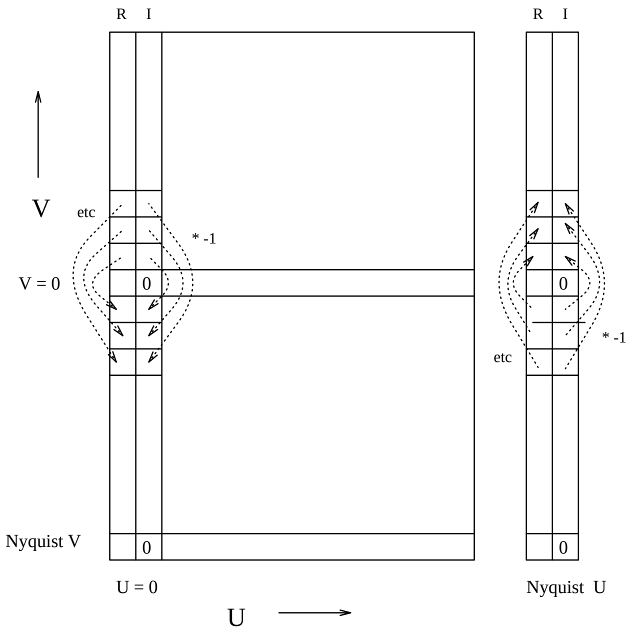



Figure 6: Reconstructing a UV grid during unpacking, when U is even.

Most FFTs expect frequency data to be ordered *(0, 1, ..., Nyquist, ..., -1)*, where the zero frequency sample is at the origin in the array. Our *UV* grid (Figure 1) has its *V* frequencies ordered *(Nyquist, ..., -1, 0, 1, ..., Nyquist - 1)*; the zero frequency is at the mid point. The *phase center* is the zero frequency sample. The *image center* is the zero time sample, or *phase reference*. Note that the phase reference is a time domain sample, not a frequency domain sample, since it is the sample at *t=0*, and its phase offset is zero.

Thus, before performing a transform from the frequency to time domain, it is necessary to reorder the *UV* grid. In the 2-dimensional case, this is equivalent to flipping the top and bottom halves of the array. This operation is done by the `exchangeUV` method described in Chapter 5 and shown in the diagram below. Since a DFT frequency domain sample is periodic in every dimension, this process is equivalent to shifting the sample in frequency by half the period, in all dimensions but the first.

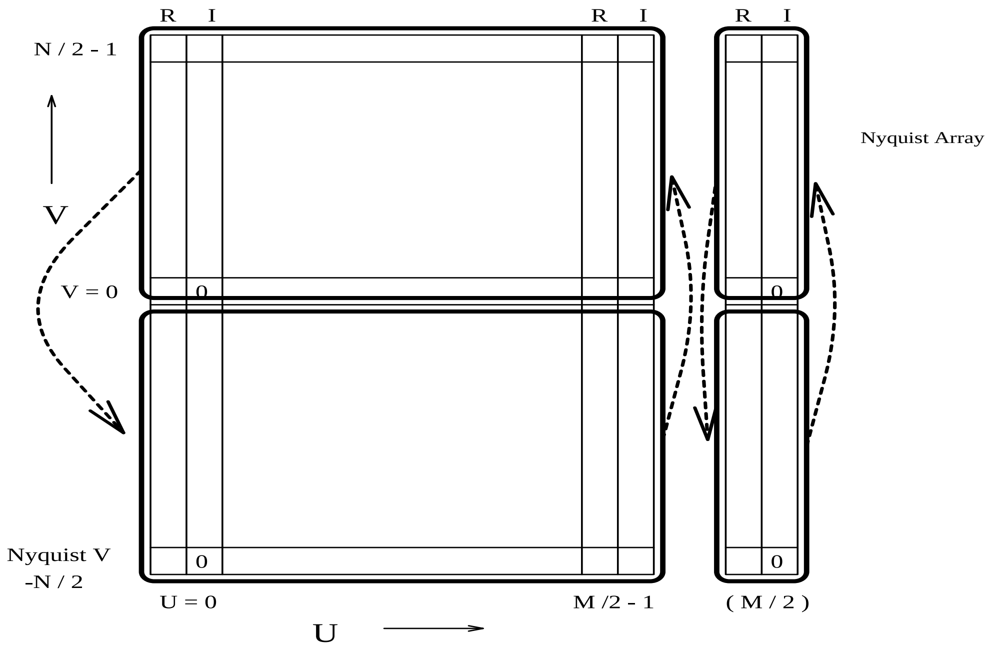



Figure 7: Exchange UV

The phase reference position of a real image that emerges from an FFT will be in the lower left hand corner of the image. However, since radio astronomers normally define the phase reference position to be 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. The `flipImage` method implements this operation. Figure 6 illustrates the process for the 2 dimensional case. Our particular sequence of operations means that the phase reference pixel in the image will be located at point *(M/2, N/2)*, or *(512, 512)* for an image of size 1024 x 1024 points, if indexing starts from 0.

The `flipImage` method is equivalent to moving the object through the periodic sample space. The shift is by half the image size, in each dimension.

If any dimension is odd, then the flipImage operation is not its own inverse. For example, suppose V is the vector *V=(0,1,2,3,4)*. If we choose the middle element 2 as the phase center, and exchange halves, the result is *(2,3,4,0,1)*. If we again choose the middle element as the phase center and exchange halves, the result is *(4,0,1,2,3)*. To ensure invertibility, we must choose the middle point according to the domain in which we are flipping. Thus, the image center and the phase center will in general be two different points. The `FFTServer` and `FourierTool` classes define these points as follows.

1. If a dimension *i* has an even size *N*, the samples will be ordered *-N/2, ..., -1, 0, 1, ..., N/2-1*. If samples are indexed *(0, 1, ..., N-1)*, then both the phase and image centers are defined to be the point with index *N/2*.
2. If a dimension *i* has odd size *N*, the the samples will be ordered *-(N-1)/2, ..., -1, 0, 1, ..., (N-1)/2)*. If samples are indexed *(0, 1, ..., N-1)*, then the phase center is defined to be the point with index *(N-1)/2*. That will be the middle point. The image center is defined to be the point with index *(N+1)/2*.
3. The first dimension is given special treatment, according to whether the object represents an Hermitian grid, a full complex grid, or a real time-domain sample. If the object is an Hermitian frequency-domain sample, then the phase center will always have index zero. Otherwise, the centers will be defined as in the first two rules.

For example, suppose $A$ is a five-dimensional array representing an Hermitian frequency domain sample, with dimensions *(4,5,2,3,4)*. Then the phase center will be the pixel with coordinates *(0,2,1,1,2)*, assuming that indexing starts from 0. The image center will be the pixel with coordinates *(0,3,1,2,2)*. If $A$ is a fully complex grid, the phase center will be the pixel with coordinates *(2,2,1,1,2)*.

**Caution:** The `FFTServer`, `FourierTool`, and `GridTool` classes expect data to be treated in this manner. In particular, it is assumed by the `pack`, `unpack`, `flipImage`, and `exchangeUV` methods. If you expect a different phase or image center, your images will be corrupt!

Methods are also provided for users who wish to perform direct FFTs; both complex to complex and real to real methods are provided. No image flipping, packing, or unpacking is done in those cases. These methods should be used by programmers requiring an optimal DFT.

**Note:** The FFT functions used by the FFTServer class can perform an FFT on an array of arbitrary size. A multi dimensional FFT is reduced to a series of one-dimensional FFTs. Given a one-dimensional array of size $n$, the running time of the FFT algorithm is a function of the prime factorization of $n$. If $n$ is a power of 2, the algorithm has an asymptotic time complexity in $O(n \log n)$. For large primes $n$, the running time will approach $O(n*n)$.

Note that image flipping is equivalent to shifting in either the time or frequency domains; a choice must be made regarding phase-center convention. Frequency or time shifting theorems can also be applied before or after a transform, to achieve the same effect. The `shift` method uses this fact; the transform method `cxfft` uses `shift`. However, for an image with odd size, the multipliers will in general be complex valued, and so this technique cannot be used for Hermetian grids.

This figure describes the image flipping of a fully complex image.

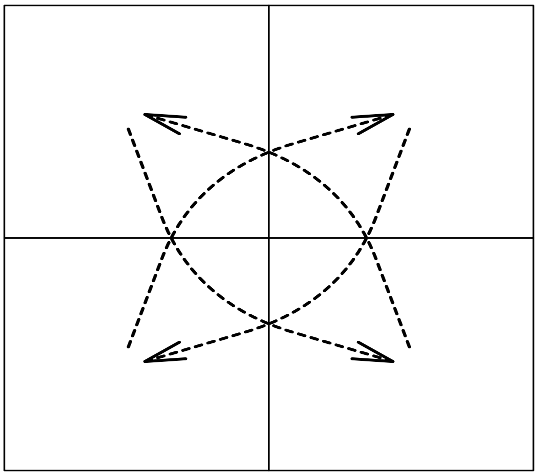


Figure 8: Flip Image

The steps that occur inside the FFT method for transforming from the frequency domain to the time domain are

1. unpack the $UV$ frequency data
2. exchange the $UV$ data to move the phase center to the `Array`'s origin

3. perform an inverse FFT from the frequency to the time domain
4. flip the resulting image

During an FFT from the image domain to the frequency domain, we perform these steps in reverse order.

1. flip image to move the phase reference point to the `Array`'s origin
2. do a forward FFT from the image domain to the frequency domain
3. exchange the *UV* data to move the phase center to the center of the grid
4. pack the *UV* frequency data

## 2.2 Full Complex *UV* grids

In general, the transform of a complex image will be a complex image without Hermetian symmetry. In this case, we must sample and store the entire domain.

For example, consider a *UV* grid of dimensions *M x N*, where both *M* and *N* are even. Then the grid will be structured as follows:

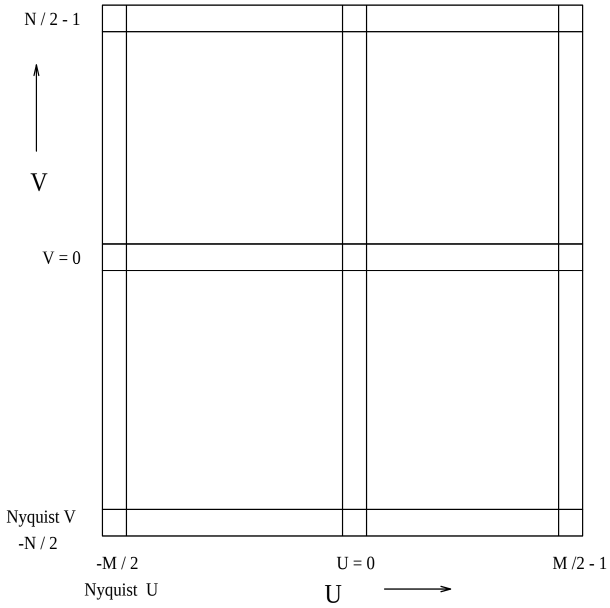



Figure 9: A complex UV grid with both dimensions even.

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 discussed in the previous section, this is the sequence expected by most FFT algorithms. Then we may perform a forward complex to complex FFT. The transformed result will have its phase reference position in the lower left hand corner of the image, 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. The steps that occur inside the full complex FFT method for going from the frequency domain to the complex image domain are

1. flip $UV$ grid to move zero frequency sample to origin
2. do FFT from frequency to time domain
3. flip complex image that comes out of FFT

During a fully complex FFT from the time domain to the frequency domain, we perform these steps in reverse order.

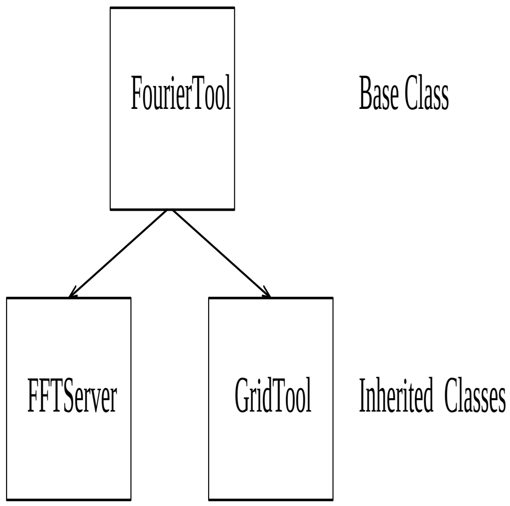
1. flip complex image to move phase reference point to origin
2. do FFT from complex image domain to $UV$ frequency domain
3. flip the quadrants of the $UV$ data to arrange frequencies with zero frequencies in centre of the $UV$ grid

# 3 Base Class Methods

## 3.1 Class `FourierTool`

The `FFTServer` class is derived from the `FourierTool` base class. A `FourierTool` object contains the Nyquist data array used during the manipulation of Hermetian grids, and the class provides methods to manage this array. Currently the `FFTServer` and `GridTool` classes are derived from this base class.

This inheritance hierarchy is shown below.




The class `FourierTool<T, S>` is a template class with two parameters. The only legal instantiations of this class or those derived from it are

- `FourierTool<float,Complex>`
- `FourierTool<double,DComplex>`

This is to ensure that the class methods may always map between real and complex valued floating point types of a given precision.

## 3.2 FourierTool Methods

This section describes the use of the `public FourierTool` methods. Examples of using these methods are shown in Chapter 6.

Note that the only legal instantiations of `FourierTool` are

- `FourierTool<float,Complex>`
- `FourierTool<double,DComplex>`

hence the parameter `T` will refer to a real floating point type and the parameter `S` will refer to a complex floating point type.

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

These methods pack the `FourierTool` object's Nyquist array into the grid `uv_grid`. All sampled frequency domain data are stored in `uv_grid`.

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

These methods extract the Nyquist data from the packed `uv_grid` object, and store them in the `FourierTool` object's Nyquist array. They then restore all data in the Hermetian grid `uv_grid`, overwriting those values that were used to store the packed data.

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

- `void reset();`

This method initializes the Nyquist array to zero.

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

These methods return a copy of the Nyquist array. `extractNYF` returns a real valued array with first dimension of size 2; complex values are represented as consecutive real and imaginary pairs. `extractNYC` returns a complex valued array with first dimension of size 1.

- `void insertNYF(const Array<T> &arr);`
- `void insertNYC(const Array<S> &arr);`

These methods copy the array `arr` into the `FourierTool` object's internal Nyquist array. The first accepts a real valued array as a parameter; the second accepts a complex valued array. The dimensions of `arr` must be the same as the existing Nyquist array; since the internal Nyquist array is stored as a real array, its first dimension will be of size 2. An equivalent complex valued array will have a first dimension of size 1.

- `void expand(Array<T> &arr, int by1ifodd=0);`
- `void expand(Array<S> &arr, int by1ifodd=0);`

These methods increase the size of the first dimension of `arr` by 2. Each row in the internal Nyquist array will be copied to the two new elements at the end of each row in `arr`. If `by1ifodd != 0`, and the first dimension in `arr` is odd, then `expandby1(arr)` will be invoked instead.

*Note:* Currently these methods copy the entire array `arr`.

- `void expandby1(Array<T> &arr);`
- `void expandby1(Array<S> &arr);`

These methods increase the size of the first dimension of `arr` by 1. The imaginary component of each row in the internal Nyquist array will be copied to the new element at the end of each row in `arr`.

*Note:* Currently these methods copy the entire array `arr`.

- `void shrink(Array<T> &arr, int by1ifodd=0);`
- `void shrink(Array<S> &arr, int by1ifodd=0);`

The last two elements of each row in `arr` are copied to each row in the Nyquist array. Then the size of the first dimension of `arr` is reduced by two.q If the first dimension of `arr` is odd and `by1ifodd != 0`, then `shrinkby1(arr)` will be invoked instead.

*Note:* Currently these methods copy the entire array `arr`.

- `void shrinkby1(Array<T> &arr);`
- `void shrinkby1(Array<S> &arr);`

These methods reduce the size of the first dimension of `arr` by 1. The imaginary component in each row of the internal Nyquist array will be copied to the new element at the end of each row in `arr`.

*Note:* Currently these methods copy the entire array `arr`.

# 4 General Purpose Methods

This chapter describes the `public FFTServer` methods which will be most frequently used. This set includes constructors and the transform methods.

Note that the only legal instantiations of `FFTServer` are

- `FFTServer<float,Complex>`
- `FFTServer<double,DComplex>`

hence the parameter `T` will refer to a real floating point type and the parameter `S` will refer to a complex floating point type.

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

These are the constructors for the class. Since the size of the internal Nyquist array is determined by the size of your working array, an `FFTServer` object must be constructed or `Set` with an `Array` or `IPosition` object before use.

The default constructor associates the new `FFTServer` object with a 1 dimensional `Array` of 0 size. If you use this constructor explicitly, you must later use the `Set` method to resize the object before doing any transforms.

Once the size of an `FFTServer` object is determined, you may safely use it to perform FFTs on different `Array` objects. A transform on a different `Array` objects will be safe if the contents of the internal Nyquist array are not relevant to, or will not be used during, the transform. See the transform method descriptions for more details.

The `Array` or `IPosition` object may be of arbitrary size and dimension. You use also any objects of classes derived from `Array`, such as `Vectors`, `Matrixes`, or `Cubes`.

Some of the FORTRAN routines used by the `FFTServer` class separate the factorization of integers and the computation of trigonometric functions into initialization subroutines. This saves computation time when performing repeated transforms on `Vectors` of the same size. The constructors perform this initialization, affording you this optimization.

- `void Set(const Array<T> &arr);`
- `void Set(const Array<S> &arr, int has_ny=0);`
- `void Set(const IPosition&);`

These methods resize the `FFTServer` object to conform with the size `arr`. They should be used when an existing `FFTServer` object is to perform transforms on a new array of a different size. The array can be specified, or an `IPosition` object can be passed specifying an `Array`'s shape.

The `has_ny=0` parameter for the complex case specifies that the `arr` parameter has an extra column of Nyquist data; this would be the case if `arr` had been `expand`ed.

Here is an example of creating an `FFTServer` object.

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

An `FFTServer` object called `fft` has been created and can be used to perform FFTs on the `Matrix abc` of size *128 x 128*, or other `Array` objects of the same size. (And different sizes, see the discussion above.)

An equivalent initialization could be done by

```
Matrix<float> abc(128,128);
IPosition Shape(abc.shape()); // note that the
                              // IPosition(const IPosition&) constructor
                              // uses copy, not reference semantics
FFTServer<float,Complex> fft(Shape);
```

The next example constructs an `FFTServer` object with a `Complex` array.

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

The object `abc` can be used to perform FFTs on any two dimensional Complex array of size 128 x 128, since the Nyquist array is *not* used during full complex to complex transforms.

- ~FFTServer();

The class destructor. It is currently trivial.

- FFTServer<T,S> &operator=(const FFTServer<T,S>&);

The copy constructor. It has copy semantics.

For all transforms, the dir parameter specifies the direction of the transform:

- If dir > 0, then a forward FFT will be performed.
- If dir <= 0, then a backward FFT will be performed. In the current version of the FFTServer class, dir == 0 always means perform a backward transform.
- The case dir == 0 has additional method specific meaning; for example, fft will not normalize the array after the backward transform.

Usually, the result of an inverse transformation will be normalized; every element will be multiplied by the factor $1/N$, where $N$ is the number of elements in the array.

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

This method performs an in-place forward or backward FFT on arr. The direction is specified by dir. The array arr is real-valued. After a forward transform, arr will be a packed Hermetian grid with complex values arranged as consecutive real, imaginary pairs. The inverse transform expects a packed Hermetian grid with complex values arranged as consecutive real, imaginary pairs; after the transform, arr will be a normalized, real valued time domain sample.

This method flips arr before and after the transform.

Normalization of arr is performed after a backward transform unless dir == 0.

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

This method performs an in-place forward or backward FFT on arr; the behaviour is identical to the method fft, except that the internal Nyquist array is not read. Rather, implied Nyquist data are stored in the last two elements of each row in arr. If the size of that dimension is odd, then the real component of the last element will be duplicated after a forward transform. During a forward transform, the last two elements will be set, and the internal Nyquist array will also be set, for convenience. During a backward transform, the last two elements will be read from; after the transform, they should be zero. The internal Nyquist array is neither modified nor read during a backward transform.

The size of arr should be *(x+2, y, z, ..., n)*, where *x* is the implied true size of the real valued time domain sample.

The method flips the image before and after the transform.

Normalization of arr is performed after a backward transform unless dir == 0.

- void cxfft(Array<S> &arr, int dir, int center=1);

This method performs an in-place complex to complex transform on the array arr. The internal Nyquist array is neither read nor modified. arr is flipped before and after the transform. Normalization of arr is performed after a backward transform unless dir==0. If center==0, no flipping is done.

This method can be used with any `FFTServer` object, regardless of size, since the internal Nyquist array is not referenced.

*Note:* The following methods `rcfft`, `crfft`, `rcnyfft`, and `crnyfft` copy the entire array several times. Avoid using them if speed is your first concern; they are provided for convenience.

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

This member performs a forward transform on `rdata`, and returns the normalized Hermetian result as a packed complex array. If the first dimension of `rdata` is odd, then `expandby1` will be invoked on the result; thus `crfft` will not in general be its inverse. See below.

Flipping is done before and after the transform. `rdata` is not modified.

Here are some examples using this method.

```
Matrix<float> m1(512,512); m1 = float(0.0);
m1(256, 256) = 10.0;      // assign some data to matrix
m1(299, 349) = 20.0;      //   m1
FFTServer ffts<float,Complex>(m1);
        // initialize an FFTServer object
Matrix<Complex> cm1 = ffts.rcfft(m1);
        // do real to complex fft

Matrix<double> m2(179,159); m2 = float(0.0);
        // odd sized double array
m2(25, 25) = 100.4;
        // assign some data to m2
m2(120, 105) = 110;

FFTServer ffts2<double,DComplex>(m2);
Matrix<DComplex> cm2 = ffts.rcfft(m2);
                                // real to complex fft
                                // note that expandby1 has been called.
                                // since the first dimension of m2 is odd.
                                // the size of the first dimension
                                // of cm2 will be 90

// do the reverse transform. We must set shrinkodd to 1.
Matrix<double> m2back = ffts.crfft(m2, 1);

// Now m2back will be the same as m2.
```

`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<Complex> 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 the real value for the Nyquist frequency.

- `Array<T> rcnyfft(Array<T> &arr);`

Performs a forward transform on arr, and returns the result as a normalized complex array. The treatment of arr is as in nyfft; except if the first dimension is odd, then the method expandby1 will be invoked on the complex result. Thus crnyfft is not always the inverse of this operation. See below.

- `Array<T> crfft(Array<S> &arr, int do_scale=1, int shrinkodd=0);`

Performs an inverse transform on the packed Hermetian complex array arr, and returns the result as a real array. arr will be normalized after a backward transform, unless do_scale == 0. If arr was the result of rcfft on an array with a first dimension of odd size, then shrinkodd should be set to 1. The method shrinkby1 will be invoked on the object first.

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);

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);

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
```

```
// odd example, showing use of shrinkodd
Vector<double> v1(17);   // vector of size 17
v1=0;                   // assign some data
v1(1)=15;
FFTServer fft2<double, DComplex> (v1);

Vector<DComplex> cv1=rcfft(v1); // cv1 is a complex vector of length 9
                                // it has been expanded by 1

                                // must set shrinkodd to 1, because
                                // v1 is odd size
Vector<double> v1same=crfft(cv1, 1, 1)
                                // v1same should now be equal to v1
```

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 do_scale==0, the result will not be normalized after a backward transform.

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.

- Array<T> crnyfft(Array<S> &arr, int do_scale=1, int shrinkodd=0);

Performs an inverse transform on the Hermetian complex array arr, and returns the result as a real array. Each row in arr should have an additional complex element at the end containing the Nyquist component, as in nyfft. The result will be normalized unless do_scale == 0. If arr was the result of rcnyfft on an array with a first dimension of odd size, then shrinkodd should be set to 1. The method shrinkby1 will be invoked on the object first.

Here is an example demonstrating the use of crnyfft and rcnyfft:

```
Cube<float> c(3, 4, 4);
        // cube with an odd first dimension
FFTServer<float,Complex> ffts1(c);
Cube<Complex> cx = ffts1.rcnyfft(c);
        // cx is a complex cube with dimensions (2, 4, 4)

        // must set shrinkodd to 1 if we want the size to be interpreted
        // correctly
Cube<float> csame = crnyfft(cx, 1, 1)

        // csame and c should be the same

IPosition size(5);
        // let's make a 5 dimensional array
size = 4;
size(3) = 7;
size(2) = 9;
```

```
        // size now looks like (4, 4, 9, 7, 4);
Array<double> arr(size);
        // define a 5 dimensional array of doubles
        // ... initialize arr ...

FFTServer<double,DComplex> ffts2(size);
        // create our FFTServer object

Array<DComplex> carr = ffts2.rcnyfft(arr);
        // ok, carr is a complex array with dimensions (3, 4, 9, 7, 4);
        // that's  3 = (4/2) + 1 for the Nyquist part

        // Do the inverse. Don't have to set shrinkodd, because the first
        // dimension is even
Array<double> darr = ffts2.crnyfft(carr);
        // darr should now be the same as arr
```

- `void rrfft(Array <T> &arr, int dir, int do_scale=1);`

Performs a forward or backward FFT on `arr`. If the transform is forward, the result will be an unpacked Hermetian grid. If the transform is backward, the input must be an unpacked Hermetian grid and the result will be a real valued time domain sample. Identical to `fft`, except that the data are assumed to be unpacked. No packing or unpacking is done. Image flipping is performed.

- `void rndfft(Array <T> &arr, int dir, int do_scale=1);`

Performs a forward or backward FFT on `arr`; the direction is specified by `dir`. Identical to `rrfft`, except that no image flipping is performed. This invokes the FORTRAN FFT functions; the method `rrfft` invokes this after performing image flipping. In turn, the method FFT performs packing and unpacking, and invokes `rrfft`.

- `void rndnyfft(Array<T> &arr, int dir, int do_scale=1);`

Performs a forward or backward FFT on `arr`; the direction is specified by `dir`. Identical in behaviour to `nyfft`, except that no image flipping, packing, or unpacking, are performed. Result will be normalized after an inverse transform unless `do_scale==0`. The internal Nyquist array will be assigned to, for later convenience. `nyfft` invokes this routine after (and before) doing image flipping.

The following methods `cndfft` and `rcndfft` provide a direct interface to multi-dimensional FORTRAN DFT routines. They should be used for programmers who want optimal complex to complex transforms, and do not require image flipping or internal Nyquist array manipulation.

Since the `cndfft` and `rcndfft` methods do not modify or read the internal Nyquist array, any `FFTServer` object can be used to transform any `Array` object with these methods.

- `void cndfft(Array<S> &arr, int dir, int do_scale=1);`

Perform a complex to complex transform on the array `arr`. Result will be normalized after a backward transform unless `do_scale==0`

- `void rcndfft(Array<T> &rdata, Array<T> &cdata, int dir, int do_scale=1);`

Perform a complex to complex transform, with the real and imaginary components provided as separate real arrays `rdata` and `cdata`, which must conform. Result will be normalized after a backward transform unless `do_scale==0`.

- `float scaleFactor(void);`

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

Here is an example of using `cndfft`, `rcndfft`, and `scaleFactor`:

```
FFTServer<float, Complex> ffts;
        // generic FFTServer object

Vector<Complex> cxx(1024);
        // complex Vector of size 1024

        // ... initialize it

ffts.cndfft(cxx, -1, 0);

float s = scaleFactor();
        // get the scale factor from the reverse transform

cxx = cxx * Complex(s, 0);
        // normalize cxx

        // direct inverse transform, don't scale

Matrix<float> real(512, 512), imag(512, 512);
        // real and imaginary parts

ffts.rcndfft(real, imag, 1);
        // direct forward transform
```

- `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<T>&)`

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 Methods

The following methods are public members of the FFTServer class, but are unlikely to be used often. They are normally used by methods such as such as `crfft`.

- ```
  void flipImage(Array<T> &image,
   int image_type=fftparms::DEF_IMAGE_TYPE, int parity=1);
  ```
- ```
  void flipImage(Array<S> &image,
     int image_type=fftparms::DEF_IMAGE_TYPE, int parity=1);
  ```

If `parity==1`, moves `image`'s phase center to the origin of the array, in the periodic n-dimensional sample space whose period is determined by the dimensions of `image`. If `parity==0`, moves `image`'s image center to the origin.

You can think of it as dividing an n-dimensional image into 2^n equal parts and swapping each of these parts with that which diametrically opposes it. So, for a matrix, it flips quadrants, for a cube, it flips octants, and so on.

If `image_type` is not equal to `fftparms::DEF_IMAGE_TYPE==0`, then `image` is assumed to have dimensions *(a+2, b, c, ..., n)*, and the last two elements in each row will not be moved.

You might be asking yourself, "what about the method `nyfft`, and other methods that have an extra two elements in each row? Don't those values need to be moved too?"

The answers are they don't need it, and no, because this method is meant to be invoked on time domain samples, or non-Hermetian complex-valued frequency domain samples. A time domain sample does not have Nyquist data; the extra space is only used in an Hermetian frequency-domain sample. The method `exchangeUV` is used to flip an Hermetian grid, because the phase center for the first dimension is already at the origin. Hence no flipping in the first dimension needs to be done for an Hermetian grid.

- `void shift(Array<S> &arr, S factor, int timeshift, const Vector<T> &shift);`

Performs the equivalent of shifting the complex valued array `arr` in either the time or frequency domains. The shifting theorems of discrete transforms are applied. For example, let $h(k)$ be a function of time, and let $H(n)$ be its Fourier Transform. Then for the real number $i$, $h(k-i)$ has as its transform $H(n)e^{-j2\pi ni/N}$, where $j = \sqrt{-1}$, and $N$ is the number of samples for $h$. Similarly, $H(n-i)$ has as its inverse transform $h(k)e^{j2\pi ik/N}$. This property can be generalized to an $n$ dimensional transform. *Brigham [2].*

If `timeshift` is nonzero, the sign of the exponent in the complex exponential function will be negative. If `timeshift==0`, the sign of the exponent will be positive. Every element in `arr` will be multiplied by `factor`. The amount to shift in dimension `i` is specified by `shift(i)`; the number of elements in `shift` must be equal to the number of dimensions in `arr`.

Thus invoking this method on an `Array` before a transform is equivalent to applying the transform, then shifting the elements after; similarly, this method can be applied after a transform to perform the equivalent of shifting before.

Example:

```
Vector<Complex> v1(4) = 0;
v1(0) = Complex(1,0);
        // v1 looks like ((1,0), (0,0), (0,0) (0,0))
Vector<float> v2(4) = 0;
v2(1) = Complex(1,0);
        // v2 looks like ((0,0), (1,0), (0,0) (0,0))

Vector<float> s(1) = 1;
        // s looks like (1)

FFTServer<float,Complex> ffts(v1);

ffts.shift(v1, 1, 1, s);
        // shift in time by 1

ffts.cxfft(v1, 1);
ffts.cxfft(v2, 1);
        // can use same FFTServer object to transform both
        // for full complex transform.

        // now v1 and v2 will be the same

v1 = 0;
v2 = 0;
v1(0) = v2(1) = Complex(1,0);
        // Reset v1 and v2; they now look like they did above
```

```
ffts.shift(v1, 1.0/4, 0, s);
        // shift in frequency domain; use factor to normalize

ffts.cxfft(v1, 0);
        // backward transform, don't normalize
ffts.cxfft(v2, -1);

        // now v1 and v2 should be the same.
```

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

Move `image`'s phase center to the origin of the array, in the periodic n-dimensional sample space whose period is determined by the dimensions of `image`.

This method should only be used for Hermetian frequency domain samples. You can think of it as moving the sample through periodic space, in each dimension but the first, through a distance of half the period. Thus it does nothing for a `Vector`, and exchanges top and bottom halves for a `Matrix`.

# 6 Examples

For more examples of using the `FFTServer` and `FourierTool` classes, please see the source code for the test program `FFTServerTest.cc`. The correct output can be found in `FFTServerTest.out`. Note that roundoff errors are introduced by the transform methods, and you will see small numbers where you expect zero.

## 6.1 Structure of `FFTServerTest.cc`

The `FFTServerTest.cc` program demonstrates the validity of `FFTServer` and `FourierTool` methods by visually demonstrating their behaviour. For the transform methods, a forward transform is always followed by an inverse transform; the difference between the original array and the identity transform is computed and reported. This should usually be close to zero; small errors may be introduced due to round off error. Substantial errors are a reason for concern and should be investigated.

Three `template` functions are provided for testing the class methods. Each testing function tests both complex and real forms of the methods, using the complex and real valued `Array` parameters. The results of the operations are displayed on the standard output, for visual inspection. The same initial arrays are used for each test.

- `template<class T, class S> void testBaseMethods(Array<T> &re, Array<S> &im)`

  Demonstrates the `FourierTool` methods `pack`, `unpack`, `extractNYF`, `extractNYC`, `insertNYF`, `insertNYC`, `shrink`, `expand`, `shrinkby1`, and `expandby1`.

- `template<class T, class S> void testTransformMethods(Array<T> &re, Array<S> &im)`

Demonstrates the `FFTServer` constructors, all of the transform methods, `extract*` and `insert*`. Note that the `FFTServer` constructors simply invoke the `Set` methods directly. The `FourierTool` base methods must work correctly for these results to be meaningful. The direct transform tests `rndfft` and `cndfft` directly call FORTRAN; compare other transform methods against these. Be aware of the effects of flipping and packing.

- `template<class T, class S> void testFlipMethods(Array<T> &re, Array<S> &im)`

  Demonstrates the `FFTServer` methods `flipImage`, `exchangeUV`.

At the end of `main` in `FFTServerTest.cc`, there is a demonstration of the validity of the `FFTServer` method `shift`. The equivalence of invoking `shift` before a transform and transforming a shifted array is demonstrated.

## 6.2 Implementations of the transform, packing, and unpacking methods

This section describes the implementations of some of the transform methods in the `FFTServer` class, and the packing and unpacking methods in the `FourierTool` class.

### 6.2.1 Transform Methods

The `FFTServer` class uses FORTRAN FFT routines. Functions from the Netlib library and a function originally written by R.C. Singleton are used. The Netlib functions are used for one-dimensional transforms on Hermetian arrays. The Singleton function is used for full complex to complex multidimensional transforms, and for transforms that require arbitrary indexing.

There are four `FFTServer` methods that call the FORTRAN FFT routines: `rndfft`, `rndnyfft`, `cndfft`, and `rcndfft`. The last routines do little more than invoke the multidimensional R.C. Singleton FFT. All other transform methods invoke one of these four methods. A general description of `rndfft` and `rndnyfft` follows; please refer to the source code for more details.

A multi-dimensional FFT may be performed by applying FFTs along 1-dimensional vectors. For example, transforming a *8 x 10* `Matrix` *M* is equivalent to performing *10* FFTs along the rows of *M*, then performing *8* FFTs along the columns of *N*. In general, a n-dimensional FFT is performed by applying FFTs along all vectors in each of the n-dimensions. If the FFT is to be performed in place without copying of elements, an indexing skip must be specified. The file `extern_fft.h` provides the following interface to the R.C. Singleton multidimensional FFT:

```
void mfft(float *a, float *b, int *ntot, int *n, int *nspan, int *isn);
```

The sign of `isn` specifies the direction of the transform; negative for a forward transform, and positive for a backward transform. The magnitude of `isn` speicifies the indexing increment between successive elements in the current dimension. This allows the user to apply a transform to any sub vector of a multidimensional array, by specifying its starting point in the array, the vector's length, and the increment between successive elements. For an array in FORTRAN order with dimensions *(a, b, c, ..., n)*, the indexing increment between successive elements along dimension *i* will be *a * b * ... * j* where *j* is the size of dimension *i-1*. (1)

Since the methods `rndfft` and `rndnyfft` perform transforms on Hermetian arrays, the general multi-dimensional complex to complex Singleton routine can not be used directly; the Nyquist elements must be considered separately.

Both methods proceed as follows, given an `Array arr`. Details of indexing calculations are not given here; please refer to the source code.

1. If the transform is forward, the Netlib FFT routine is used to perform a forward transform on each row in `arr`. Elements in each row will be arranged *re(1), re(2), im(2), ..., re(n-2), re(n-1)* for the even case, and *re(1), re(2), im(2), ..., re(n-1), im(n-1)* for the odd case. Elements are shuffled to arrange each row as an array of real and imaginary pairs.
2. For each dimension, the Singleton FFT routine is used to perform a forward or backward transform on each vector in that dimension. The first real and imaginary components will be consecutive elements in `arr`; the indexing skip between successive complex pairs will be twice the product (1) computed above. If the location of the next vector is beyond the end of `arr`, then the location of the next vector can be computed by subtracting the length of `arr` plus one normally, or `arr` plus two if the last vector computed was a Nyquist component.
3. For a backward transform, the Netlib FFT routine is used to transform each row in `arr`. Elements are shuffled and copied to the Nyquist array.

### 6.2.2 Packing and Unpacking Methods

The `FourierTool` methods access the underlying storage of an `Array arr` during the `pack` and `unpack` operations. The real valued Nyquist array is considered a sequence of rows; the first plane of `arr` is isomorphic to the Nyquist array. Each row in the Nyquist array is a single complex number.

The `pack` and `unpack` operations proceed similarly at first. An array `used` of integers, the same shape and size as the Nyquist array, is used to record elements and their conjugates as the array is traversed.

The pack operation proceeds as follows.

For each row *i* in the Nyquist array,

1. The location of the complex conjugate of *i* in the Nyquist array is computed, under the assumption that the first dimension of `arr` is even. The corresponding location in the first plane of `arr` is the complex conjugate in `arr`, whether the first dimension of `arr` is even or odd.
2. If the element in `arr` corresponding to *i* is marked as used, then this step is skipped and loop continues with the next row. Otherwise, the element and its conjugate are marked as used.
3. If the element in `arr` is its own conjugate, then only the imaginary part can be overwritten. If the first dimension of `arr` is even, then the real part of the Nyquist element is copied to the imaginary part of the element of `arr`. If the first dimension of `arr` is

odd, then the imaginary part of the Nyquist element is copied to the imaginary part the element of `arr`.

Otherwise, if the element in `arr` is not its own conjugate, both the real and imaginary parts can be overwritten. If the size of the first dimension of `arr` is even, then the real and imaginary parts from the Nyquist row and copied to `arr`. Since `used` marks this element and its conjugate, we are guaranteed to not overwrite the conjugate later in the traversal.

If the size of the first dimension of `arr` is even, then the imaginary parts from two consecutive elements in `arr` are copied to the real and imaginary parts of the element in `arr`. A separate pointer into the Nyquist array must be advanced by two rows.

The unpack operation proceeds similary, and differs in the third step:

3. If the element in `arr` is its own conjugate, then the real part is the real part in `arr`, and the imaginary part is the real or imaginary part in the Nyquist array. If the first dimension of `arr` is odd, then the imaginary part of `arr` is copied to the imaginary part in the Nyquist array, and the real component at the end of the row in `arr` is copied to the real part of the element in the Nyquist array. Otherwise, the imaginary part of `arr` is copied to the real part of the Nyquist array, and zero is written to the imaginary part in the Nyquist array.

   If the element in `arr` is not its own conjugate, then its values are copied to the Nyquist array. If the first dimension of `arr` is odd, then the real and imaginary parts from `arr` are copied to the imaginary parts of two consecutive elements in the Nyquist array, and the real part of the last element of the row in `arr` is copied to the real part of the Nyquist element.

   If the first dimension of `arr` is even, then the real and imaginary parts of the element in `arr` is copied to the Nyquist element. The conjugate of the element in the Nyquist array is computed, and stored in the conjugate's location.

   In any case, the value of the conjugate in `arr` is used to reconstruct the value in `arr`.

For details of the process, please consult the code.

# References

1. BRACEWELL, RONALD N. *The Fourier Transform and Its Applications.* McGraw Hill, 1986.
2. BRIGHAM, E. ORAN. *The Fast Fourier Transform.* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.
3. Personal communications with Dr. Tony Willis, Dominion Radio Astrophysical Observatory, Penticton, B.C., Canada, January to April 1995.