Note 158: The AIPS++ GridTool Class

How to use the GridTool class - definitions and tutorial

Anthony G. Willis

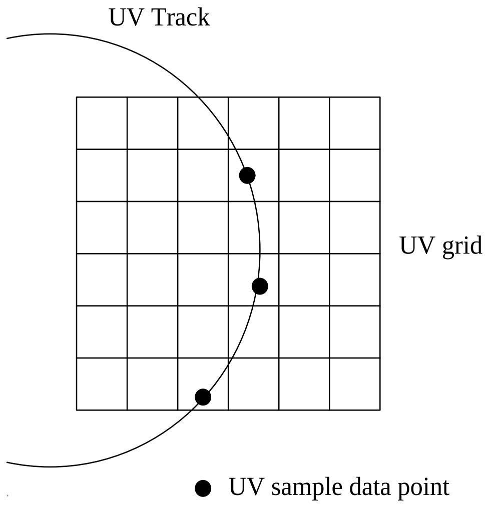
1 Introduction

The AIPS++ `GridTool` class is designed to convolve data on to a grid having an arbitrary number of dimensions. The actual number of dimensions in which gridding is done is specified at the time an object of the class is created. The only limitation to this technique is that the convolution function must be dimensionally separable; any n-dimensional convolution function must be a product of individual one dimensional functions. At present this class has methods for convolution of complex UV domain visibility data on to either a Hermitian grid or a full complex UV grid. There are also methods to convolve floating point data on to a regular cartesian grid.

2 Convolution in the Fourier Domain

While the `GridTool` class can do convolution of floating point data on to a regular Cartesian grid, this class was developed mainly for the purpose of convolving radio interferometer visibility data on to a Fourier domain UV grid. This process is necessary because radio astronomy aperture synthesis telescopes obtain information about the sky brightness distribution by indirect imaging. That is, they do not directly observe the sky brightness distribution, but rather, sample data in the Fourier domain. It is then necessary to Fourier transform the collected data back to the image domain to obtain an image of the sky brightness.

To do the Fourier Transform in a reasonable amount of time we normally use a Fast Fourier Transform (FFT), which requires that the data be on a uniformly spaced grid. However radio interferometers do not sample data on a uniform grid, but rather, sweep out ellipsoidal tracks in the UV domain. In order to use an FFT, it is necessary to convolve data sampled along along a UV track on to a regularly spaced grid (see Figure 1).



The AIPS++ `GridTool` class provides application programmers with a very flexible tool for interpolating Fourier domain visibility data on to a grid of dimension two or higher before doing a FFT into the image domain. The `GridTool` is designed to be able to grid in an arbitrary number of dimensions. The number of actual dimensions in which gridding is done is specified when an object of the class is constructed. (Of course it is unlikely that you would want to grid radio astronomy Fourier domain data of more than 3 dimensions, but you can if you want to!)

This document describes the public methods for this class and how to use them.

3 Data Structure

Since a UV grid created by `GridTool` will be converted into an image by an object of the `FFTServer` class, the UV grids created by `GridTool` will have the same layout as that described in Chapter 2 of the `FFTServer` User's Guide (Note xxx). You should read that document if you wish to further information on the layout of a UV grid.

4 Base Class Methods

As does the `FFTServer` class, the `GridTool` class inherits from a base class called `FourierTool`. `FourierTool` is a class with an array to hold Nyquist data associated with Hermetian UV grids and with 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 Nyquist data. At present the `FFTServer` and `GridTool` classes inherit from this base class.

Here are the methods defined in base class `FourierTool`.

- `int pack(Array<T> & uv_grid);`
- `int 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.

- `int unpack(Array<T> & uv_grid);`
- `int 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 data from the complex conjugate part of the Hermetian UV grid.

- `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 `GridTool` onject after the array has been modified for some reason.

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

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

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.

5 General Purpose Methods

Here are the methods defined in class `GridTool`.

5.1 Constructor

The GridTool constructor is of the form

- `GridTool<T, S> ( MathFunc<T>** mathptr, Matrix<T>& input_parms)`

where `T` can be one of `float` or `double`, and `S` can be one of `Complex` or `DComplex`. Single precision data types `float` and `Complex` should be used together as should types `double` and `DComplex`. Other mixtures are not guaranteed to produce sensible results!

The first parameter required by the constructor is a pointer to an array of MathFunc objects. There will be n of these MathFunc objects, where n will be 2 if we are convolving on to a 2 dimensional UV grid and 3 if we are convolving onto a UVW grid. Each MathFunc object describes a convolution function for a particular dimension; we make the assumption that the net n-dimensional convolution function we derive will be a product of these individual 1-dimensional functions, e.g. f(x,y) = f(x) * f(y)

The second input data item is a matrix of size 3 x n, where n is again the number of dimensions in the UV grid. For each dimension we must specify 3 parameters. The first parameter is the size of the grid in that dimension. The default size is 1024. The second parameter gives a "cellsize" in radians, for the separation between pixels in the final Fourier transformed image of the sky brightness distribution. The `GridTool` object will use this information to compute the proper separation between grid points in the UV domain. Eventually the AIPS++ Measures class will allow you to enter cellsize in units other than radians. (If you just want to interpolate floating point data on to a Cartesian grid with a separation of 1 between grid points, then enter a value of 0 for the cellsize.) The third parameter gives the number of sub elements into which we will divide a grid cell in that dimension in order to compute the lookup table for the convolution function. The default is 100. Here is an example of constructing an object of class `GridTool`.

```
int imsize = 2048;           // define image / UV grid size
                             // could be different in different dimensions
float cellsize = 20.0 *C::arcsec;      // define a cellsize in arcsec
                             //   and convert to radians
int n = 2;                   // number of dimensions (2 in this example)
Matrix<float> input_parms(3,n); input_parms = float(0);
                             // declare and initialize a matrix to hold
                             // input parameters
input_parms(0,0) = imsize;  // specify the size of the image
input_parms(0,1) = imsize;  //      in each dimension
input_parms(1,0) = cellsize;// specify the cell size
input_parms(1,1) = cellsize;//      in each dimension
                             // input_parms(2,0) = input_parms(2,1) = 0
                             // so the default of 100 subdivisions per
                             // UV cell will be used to compute the
                             // convolution function lookup table
MathFunc<float>** Mathptr = new MathFunc<float>*[n];
                             // set up an array of math functions
                             // one math function needed for each dimension
for (int i = 0;i<n;i++) {   // decide which type of math function
   Mathptr[i] = new Sph_Conv<float>(); // to use in each dimension
}                                       // here Spheroidal function used
                                        // in each dimension

GridTool<float,Complex> grid_tool(Mathptr,input_parms);
                             // Construct a GridTool
```

5.2 Gridding

Gridding is done through the methods

- `int gridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<T> &uv_grid, Array<T> & uv_weight);`

- int gridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<S> &uv_grid, Array<T> & uv_weight);
- int cGridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<S> &uv_grid, Array<T> & uv_weight);
- int grid(Array<T> & loc, Array<T> & data, Array<T> &grid, Array<T> & uv_weight);

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

Method gridUV takes arrays of UV data locations (uv_loc) and the visibilities at those locations (uv_data), and interpolates the visibility data on to a Hermetian UV grid (uv_grid) in preparation for a complex to real FFT. At the same time, the weight of each convolved visibility grid point is added to the uv_weight array.

The gridded uv_grid will be in unpacked form, i.e. any UV data that gets gridded at the U Nyquist frequency will get stored in the separate internal Nyquist array. If you intend to FFT the uv_grid data with an object of the FFTServer class, you must first pack the data before doing the FFT.

uv_loc can either be a vector, whose elements are U, V, (and optionally W), or a matrix whose individual columns represent specific UV(W) locations.

uv_data can either be a vector, whose elements are the corresponding real and imaginary values, followed by a weight, of a specific visibility, or a matrix whose individual columns contain the visibility data for the visibility locations defined in matrix uv_loc.

The GridTool has methods specifically developed for gridding and degridding operations. It does not understand anything about tapering of visibility data, uniform weighting of data, etc. If you want to taper visibility data, etc, you must do so outside the GridTool and pass the result of the operation in as a factor in the weight field of uv_data.

uv_grid is used to store the gridded visibilities, and can be a Matrix if we are doing two dimensional gridding, or a Cube if we are doing three dimensional gridding. It can be declared in either real or complex form. If you use the real form then you should define its dimensions to be that of the final output sky image; if you use the complex form then the first dimension should be half that of the final output sky image.

uv_weight is used to store the weight associated with each gridded visibility point. It can be a Matrix if we are doing two dimensional gridding, or a Cube if we are doing three dimensional gridding. For the case of a Hermetian UV grid, the size of this real array in the first dimension is half the final real image size plus 1. In the remaining dimensions it equals the image size.

Here is an example.

We continue with the GridTool object created in the previous example and assume we have a UV data point at U = 20000 wavelengths, V = 30000 wavelengths, with a real visibility value of 7.5 and and imaginary visibility value of 2.2. The weight of this point will be 1.0 .

```
Vector <float> uv(2);
Vector <float> uv_vall(3);
Matrix <float> data(imsize,imsize);
                              // declare 2048 x 2048 array to hold uv grid
data = float(0.0);            // set the array to zero
```

```
Matrix<float> wts(imsize/2+1,imsize);
                            // declare weighting grid; its size is
                            // always imsize/2 + 1  in the first
                            // dimension
wts = float(0.0);           // set the weights array to zero
uv(0) = 20000.0;            // assign UV values to uv
uv(1) = 30000.0;
uv_vall(0) = 7.5;           // assign visibility values to uv_vall
uv_vall(1) = 2.2;
uv_vall(2) = 1.0;
grid_tool.gridUV(uv,uv_vall, data, wts);
                            // grid the visibility
grid_tool.pack(data);       // pack UV data for transport to FFTServer
```

It is important to note the last line of the above example. Gridding on a Hermetian UV grid is always done on an unpacked UV grid, with any data at the U Nyquist frequency being stored in the separate internal Nyquist array. If you intend to FFT this grid to the sky image domain, you must perform a `pack` operation to carry any Nyquist data along to an `FFTServer` object, which will perform the FFT.

Since the GridTool handles general arrays, we can also feed it matrices of UV locations and UV values and it will digest them without a hitch. An example follows:

```
Matrix <float> uv(2,20);      // Matrix uv can hold 20 U, V values
Matrix <float> uv_vall(3,20); // Matrix uv_vall can hold 20
                              //  real / imaginary values and
                              //  their weight
Matrix <float> data(2048,2048);
                              // declare 2048 x 2048 array to hold uv grid
data = float(0.0);            // set the array to zero
Matrix<float> wts(imsize/2+1,imsize);
                              // declare weighting grid
wts = float(0.0);             // set the weights array to zero
for (int i = 0; i < 20;  i++){// assign some silly values
  uv(0,i) = i * 5000.0;       //   to the uv and uv_vall
  uv(1,i) = i * 3500.0;       //   matrices
  uv_vall(0,i) = i * 7.5;
  uv_vall(1,i) = i * 2.2;
  uv_vall(2,i) = 1.0;
}
grid_tool.gridUV(uv,uv_vall, data, wts);  // do the gridding
```

The grid tool can equally well digest cubes of data. See the `proto_3d_image` example in Chapter 7.

Method `cGridUV` takes visibility data and does convolution on to a full complex UV grid. No Hermetian symmetry is assumed in this case. The `uv_weight` array must now have the same dimensions and size as that of the UV grid array. See the detailed example of full complex gridding, `complex proto_map` in the last chapter.

Method `grid` takes real or double floating point data and does convolution on to a Cartesian grid. The `weight` array must have the same dimensions and size as that of the Cartesian grid.

5.3 De gridding

The other main function of the GridTool class is to de-grid visibility data from a UV grid on to interferometer UV tracks. Here are declarations of the methods available to do degridding.

- `int degridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<T> &uv_grid, int do_weight = 0);`
- `int degridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<S> &uv_grid, int do_weight = 0);`
- `int cDegridUV(Array<T> & uv_loc, Array<T> & uv_data, Array<S> &uv_grid, int do_weight = 0);`
- `int degrid(Array<T> & loc, Array<T> & data, Array<T> &grid, int do_weight = 0);`

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

Method `degridUV` de-grids UV domain visibilities from a Hermetian UV grid.

To de-grid UV domain visibilities, you provide the GridTool object with the UV location of the visibility you wish to create (in Array `uv_loc`), the UV grid (in Array `uv_grid`), and the value of a parameter `do_weight`.

A Hermetian UV grid must be in unpacked form, i.e. the U Nyquist data must be stored in the separate internal Nyquist array. If you have obtained this grid from the output of an FFTServer object's FFT from the sky to the UV domain, the first command to the GridTool object must be an `unpack` operation.

The GridTool object will return the real and imaginary components of the visibility in array `uv_data`, along with a weight of 1 if it could successfully degrid the visibility. A weight of -1 will be returned if the visibility could not be properly degridded (location of the UV point is outside the UV grid, or some other cause yet to be determined). The format of this array is the same as was described for the gridding method.

The `do_weight` parameter should be left with its default value of 0 (zero) if you are deconvolving off a grid whose data were put on to it by a convolution operation. If the grid was derived by some type of modelling procedure (eg. find CLEAN components on the sky, then FFT these components into UV space), `do_weight` should be set to a non-zero value.

Here is a simple example of degridding at a single UV location. We assume that the 2048 x 2048 `data` array that we have been working with is now a UV grid that has been obtained by Fourier transfomation from some model of the sky.

```
Vector<float> uv(2);
Vector<float> uv_vall(3);
uv_vall = float(0.0);
uv(0) = 20000.0;          // UV locations in units of wavelength
uv(1) = 30000.0;
grid_tool.unpack(data);   // unpack Nyquist data from UV grid
grid_tool.degridUV(uv,uv_vall, data, 1);  // degrid to uv tracks
```

`uv_vall(0)` and `uv_vall(1)` now contain the real and imaginary components of the visibility at U = 20000.0, V = 30000.0. `uv_vall(2)` is a weight with value 1.0.

Just as we could grid a group of visibilities so can we degrid a group.

```
Matrix <float> uv(2,20);       // Matrix uv can hold 20 U, V values
Matrix <float> uv_vall(3,20); // Matrix uv_vall can hold 20
                               //  real / imaginary values and
                               //  their weight
uv_vall = float(0.0);          // initialize uv_vall to zero
for (int i = 0; i < 20;  i++){// assign some silly values
  uv(0,i) = i * 5000.0;        //   to the uv matrix
  uv(1,i) = i * 3500.0;
}
grid_tool.degridUV(uv,uv_vall, data, 1);  // do the degridding
```

`uv_vall(0,i)` and `uv_vall(1,i)` now contain the real and imaginary components of the `i` th visibility. `uv_vall(2,i)` is a weight with value 1.0.

Method `cDegridUV` de-grids UV domain visibilities from a full complex UV grid laid out in the form described in the `FFTServer` Users' Guide.

To de-grid UV domain visibilities, you provide the GridTool object with the UV location of the visibility you wish to create (in Array `uv_loc`), the UV grid (in Array `uv_grid`), and the value of a parameter `do_weight`.

The GridTool object will return the real and imaginary components of the visibility in array `uv_data`, along with a weight. The formats of these arrays are the same as was described for the gridding method.

The `do_weight` parameter should be left with its default value of 0 (zero) if you are deconvolving off a grid whose data were put on to it by a convolution operation. If the grid was derived by some type of modelling procedure (eg. find CLEAN components on the sky, then FFT these components into UV space), `do_weight` should be set to a non-zero value.

Method `degrid` de-grids single or double precision floating point data from a cartesian grid.

To de-grid floating point numbers, you provide the GridTool object with the coordinate location of the data point you wish to obtain (in Array `loc`), the Cartesian grid of data values (in Array `grid`), and the value of the parameter `do_weight`.

The `GridTool` object will return the computed value of a data point in array `data`, along with a weight. The formats of this array is the same as was described for the gridding method.

The `do_weight` parameter should be left with its default value of 0 (zero) if you are deconvolving off a grid whose data were put on to it by a convolution operation. If the grid was derived by some type of modelling procedure (eg. find CLEAN components on the sky, then FFT these components into UV space), `do_weight` should be set to a non-zero value.

5.4 Auxiliary Methods

- `convCorrect( Array<T> & image)`
- `convCorrect( Array<S> & image)`

If we convolve visibility data on to a UV grid, and then FFT the UV grid into the image domain, the resulting image is not a proper representation of the sky. Instead the image appears to have been multiplied by a function that is the Fourier Transform of the convolution function we used to convolve the original visibility data on to the grid. This is a simple result of the relation

$$A \otimes B \longleftrightarrow a * b$$

where $\otimes$ is the convolution operator and $*$ is the multiplication operator. $\longleftrightarrow$ indicates a Fourier transform. Thus, if A stands for the original visibilities and B for the convolution function, a will stand for the final sky map and b for the Fourier transform of the convolution function.

To obtain the actual sky distribution (a) we must divide out the multiplication function (b). This is done by the `convCorrect` method.

Also, if you are going to convert a model sky into interferometer visibility tracks, you must apply this method, BEFORE FFTing the model sky into the UV domain. This is necessary because the deconvolution operations that take place inside the `degridUV` method are an inverse of the gridding procedure.

The `proto_map` program in Chapter 5 shows examples of the use of `convCorrect`.

- `int countUV( Array<T> & uv_loc, Array<Int> &uv_count)`
- `int cCountUV( Array<T> & uv_loc, Array<Int> &uv_count)`

These methods assist you if you want to do uniform weighting or for any other reason want to count the number of ungridded raw visibility points that will fall into a given UV grid cell. `countUV` is used if you are working with Hermetian UV grids and `cCountUV` if you are working with full complex UV grids.

`uv_loc` is a vector or matrix of UV locations just like that used for the `gridUV` or `degridUV` methods. `uv_count` is an integer array whose dimensions are the same as those of the `uv_weight` array required for methods `gridUV` or `cGridUV`; that is, in the case of `gridUV`, sky image size /2 + 1 in the first dimension and the same sizes as the image in the remaining dimensions, and in the case of `cGridUV` the same dimensions as the complex image.

If the UV location `uv_loc` of an ungridded visibility falls within a particular UV cell, x,y, then the value of `uv_count(x,y)` is incremented by 1.

Here is an example (again we assume an image size of 2048 x 2048):

```
Int imsize = 2048;             // set image size
Matrix <float> uv(2,20);       // Matrix uv can hold 20 U, V values
for (int i = 0; i < 20;  i++){// assign some silly values
  uv(0,i) = i * 5000.0;        //   assign some silly values
  uv(1,i) = i * 3500.0;        //   to the UV matrix
}
Matrix<Int> uv_count(imsize/2+1,imsize);
grid_tool.countUV(uv, uv_count);
```

The locations of matrix `uv_count` now contain the number of telescope based visibilities which will fall into each of the UV grid cells.

- `int numValues( Vector<T> & uv_loc, Array<Int> &uv_count)`
- `int cNumValues( Vector<T> & uv_loc, Array<Int> &uv_count)`

If you have previously run all visibilities that you wish to grid through the `countUV` or `cCountUV` method, the array `uv_count` now contains the count of the number of visibilities which fall into each UV grid cell. Calling method `numValues`, if you are working with a Hermetian UV grid, or or `cNumValues`, if you are working with a full complex UV grid, with the UV location `uv_loc` of an ungridded visibility will cause the method to return the total number of ungridded visibility points that fall within the UV grid cell associated with this visibility.

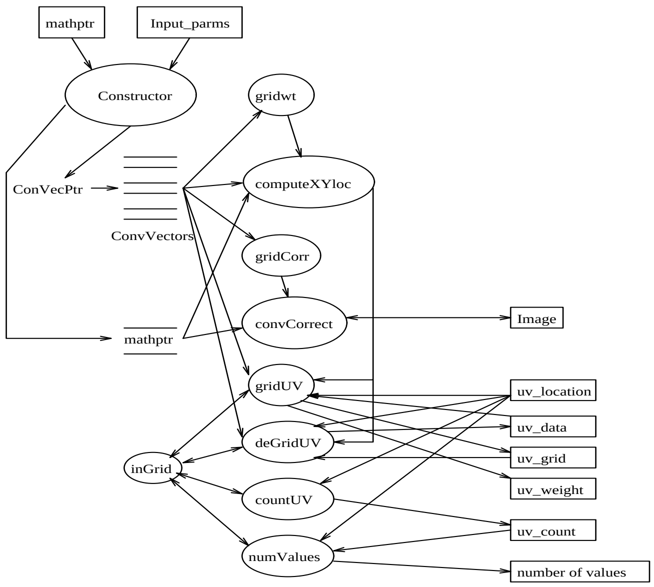
If you assign the visibility at `uv_loc` a weight equal to the inverse of `numValues` or `cNumValues` before gridding, then you should end up with a uv grid having uniform weighting.

Here is an example. It is essentially a continuation of the previous one.

```
      Vector <float> uv(2);          // vector containing uv locations
      Vector <float> uv_vall(3);     // vector containing visibility
                                     //  real / imaginary values and
                                     //  their weight
      uv(0) = 20000.0;               // assign UV values to uv
      uv(1) = 30000.0;
      uv_vall(0) = 7.5;              // assign visibility values to uv_vall
      uv_vall(1) = 2.2;

   // weight the visibility by the inverse of the number of visibilities
   // found in the UV cell
      uv_vall(2) = 1.0 / float(grid_tool.numValues(uv, uv_count) );
```

6 Internal Structure of the GridTool Class

The functional structure of class GridTool is indicated in the following diagram.



GridTool Class

The `Constructor` method takes the input `mathptr` to the array of `MathFunc` objects and stores it as an internal private data member. It takes the image size in each dimension, the number of subdivisions per cell for the convolution function lookup table, and the support width for the convolution function in each dimension, and constructs a `ConvVector` object for each dimension.

These `ConvVector` objects contain methods and data for doing one-dimensional convolutions. Amongst other things they compute the convolution weight of a data point in the given dimension, the weighted value of the visibility in the specified dimension and the inverse direct Fourier transform (DFT) of the one-dimensional convolution function.

`gridwt`, `computeXYloc`, `gridCorr` and `inGrid` are private methods used by the class. `gridwt` combines the convolution weight in each dimension to give the combined weight at a grid point. `gridCorr` combines the one dimensional DFTs to give a correction value at each grid point which is then passed to the public method `convCorrect`. `computeXYloc` is called by the public methods `gridUV` and `degridUV`. It interacts with the `ConvVector` objects to compute the convolved UV grid locations and weights at those locations in each dimension. `inGrid` checks if a computed UV grid coordinate actually lies within the UV grid that will be used for the FFT to the image domain.

The remaining methods are public ones. Their function has been described previously in Chapter 4.

7 Some Working Examples

Perhaps the best way to present the use of `GridTool` and `FFTServer` objects is to give some complete working examples.

7.1 proto_map

`proto_map` is a small program which creates artificial images as seen by a synthetic aperture radio telescope from an array of point sources at positions entered by a user. The image in the sky domain is real, so we can do the gridding and FFTS using methods for Hermetian UV grids.

```
//
// proto_map; a simple program for making maps
//
#include <iostream.h>
#include <aips/Math.h>
#include <aips/Constants.h>
#include <aips/String.h>
#include <aips/aips.h>
#include <aips/ArrayIter.h>
#include <aips/Matrix.h>
#include <aips/Vector.h>
#include <aips/GridTool.h>
#include <aips/FFTServer.h>

//
```

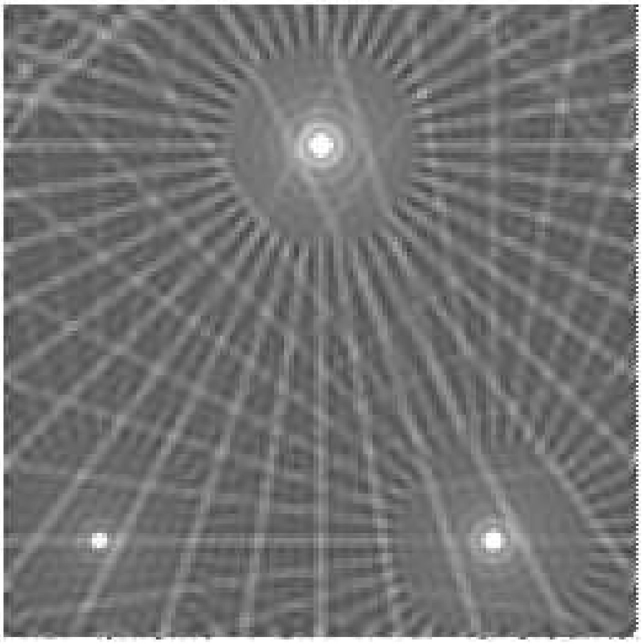
```
// This program tests the combination of GridTool and FFTServer classes
// and makes simple maps
// basic algorithm -
//     get an array of point sources
//     FFT the point source model into UV domain
//     degrid to obtain raw telescope visibilities
//     grid the visibilities
//     FFT UV gridded visibilities to image domain to get artificial sky
//        image  as seen by telescope
//     reset UV grid to zero
//     transfer gridding weights to UV grid
//     FFT UV grid of weights to image domain to get synthetic antenna pattern
//
int main()
{
// obtain image size
   int imsize;
   cout << "Enter image size (must be power of 2) : ";
   cin >> imsize;
// should actually check that image is power of 2 !!

// declare a matrix, could be different sizes in x and y in general
   Matrix<float> image(imsize,imsize); image = 0.0; //uv grid <-> image

// now enter some point sources (positions and flux densities)
// to create a map of point sources
   cout << "Enter point source flux density and x and y locations "<<endl;
   cout << "x and y must lie in range 0 to "<<imsize - 1 <<endl<<endl;
   float fluxdensity;
   int finished = 0;
   int x, y;
   cout <<" "<<endl;
   while (!finished) {
      cout << "Enter point source flux density (-999 to finish) ";
      cin >> fluxdensity;
      if (fluxdensity == -999.0)
         finished = 1;
      if (!finished) {
         cout << "Enter point source x and y location:";
         cin >> x >> y;
         if (x >=0 && x < imsize && y >=0 && y < imsize)
             image(x,y) = fluxdensity;
      }
   }

// set number of dimensions, 2 in this example
   int n = 2;

// create a cellsize (in arcsec) - use 20 arcsec in this test program
```

```
   float cellsize = 20.0 * C::arcsec;

// set up array which will carry set up parameters into GridTool
// constructor
   Matrix<float> input_parms(3,n); input_parms = float(0);

// tell the grid-tool the dimensions of the image and the cellsize
   input_parms(0,0) = imsize;
   input_parms(0,1) = imsize;
   input_parms(1,0) = cellsize;
   input_parms(1,1) = cellsize;

// set up an array of math functions - one for each dimension
// use Spheroidal convolution in  this example
   MathFunc<float>** Mathptr = new MathFunc<float>*[n];
   for (int i = 0;i<n;i++)
     {
     Mathptr[i] = new Sph_Conv<float>();
     }
// Construct an object of class GridTool
   GridTool<float,Complex> grid_tool(Mathptr,input_parms);

// Apply convolution correction to image
// This operation must be done before FFTing to UV domain to degrid
   grid_tool.convCorrect(image);

// Construct an FFTServer
   FFTServer<float,Complex> fft(image);

// FFT array of point sources to UV grid
// The UV grid returned by method fft will be in packed format
   fft.fft(image,1);

// Unpack the UV grid to do uv plane degridding
   grid_tool.unpack(image);

// create some fake uv "tracks" for an east-west interferometer
// we will assume that we can have 100 spacings.
// each spacing will have 20 uv sample points, each sample point
// having a u and v value (uv_coord cube) and a cosine, sine and
// weight value (uv_val cube)
   Cube <float> uv_coord(2,20,100); uv_coord = float(0.0);
   Cube <float> uv_val(3,20,100); uv_val = float(0.0);
   int uvmax = imsize / 3 - 1;
   if (uvmax > 99)
      uvmax = 99;
   float delta_theta = C::pi / 20.0;

// set up some objects of class ArrayIterator so that we will be
```

```
// able to iterate through these data cubes  one layer (matrix) at a time
   ArrayIterator<float> uv_iter(uv_coord,2);
   ArrayIterator<float> uv_iter_dat(uv_val,2);

// The next conversion allows us to take uv location in grid units and
// turn it into uv location in 'wavelengths' so that
// GridTool can reverse the operation!!
   float scale_factor = 1.0 / (C::asec2rad*input_parms(0,0)*cellsize);

// now compute the uv locations for each spacing
   for (i = 1; i <= uvmax; i ++) {
      Matrix <float> muv_coord(2,20); muv_coord = float(0.0);
      Matrix <float> muv_val(3,20); muv_val = float(0.0);
      for (int j = 0; j < 20; j ++) {
         float theta = -C::pi_2 + j * delta_theta;
         float v = i * sin(theta);
         float u = i * cos(theta);
         muv_coord(0,j) = u * scale_factor;  // muv_coord is in wavelengths
         muv_coord(1,j) = v * scale_factor;
      }

// now pass the vector of uv locations into grid_tool to degrid
// and get values for the raw telescope visibilities, muv_val
      grid_tool.degridUV(muv_coord,muv_val, image, 1);
      uv_iter.array() = muv_coord;
      uv_iter_dat.array() = muv_val;
      uv_iter.next();
      uv_iter_dat.next();
   }

// visibilities as seen by the telescope have been created
// now make the map - first do gridding
   uv_iter.origin();             // reset the iterators
   uv_iter_dat.origin();

   image = float(0.0);           // reset the image array to zero
   grid_tool.reset();            // reset internal Nyquist array to zero
   Matrix<float> wts(imsize/2+1,imsize); wts = 0.0;   //weighting grid

// pass though the uv cubes one layer (or baseline) at a time
   for (i = 0; i < uvmax; i ++) {  // do gridding for each baseline
      grid_tool.gridUV(uv_iter.array(),uv_iter_dat.array(), image, wts);
      uv_iter.next();
      uv_iter_dat.next();
   }

   grid_tool.pack(image); // pack uvgrid for transport to FFTServer object, fft
   fft.fft(image,0);      // FFT uv plane to image domain
                          //   -don't do any scaling
```

```
                              //   -scaling is done by dividing
                              //    by sum of weights
      grid_tool.convCorrect(image);   // correct for convolution effects
      float sum_of_weight = fft.wtsum(wts);
      image /= sum_of_weight;      // normalize output image
   // image has been made; in real universe you  would store it to
   // disk at this point

   // now make an antenna pattern
      image = 0.0;                 // again, reset image array to zero

   // copy the weights array to the UV grid
   // weights get copied into real components
   // imaginary components are all zero
      fft.uvassign(image, wts);

      fft.fft(image, 0);           // FFT to image domain
      grid_tool.convCorrect(image);
      image /= sum_of_weight;      // normalize; maximum peak better be 1 !!
   // antenna pattern has been made; in real universe you would store
   // it to disk at this point

   // Whew! finished, I think
   }
```

If we ran the previous program, specifying a map size of 128, and put point sources with intensities 10, 20 and 30 at X,Y locations (19,19), (99,19), and (64,99) respectively, we would get an image like the following:



7.2 proto_3d_image

`proto_3d_image` is a program which takes UVW visibility tracks, grids them and makes a 3 dimensional image.

```
#include <iostream.h>
#include <aips/Math.h>
#include <aips/Constants.h>
#include <aips/aips.h>
#include <aips/ArrayIter.h>
#include <aips/Matrix.h>
#include <aips/Vector.h>
#include <aips/GridTool.h>
#include <aips/FFTServer.h>

//
// This program tests the combination of GridTool and FFTServer objects
// with 3-d grids and makes a simple 3 dimensional image
// basic algorithm -
//     get image parameters:  size, cellsize
//     read in telescope visibilities
//     grid the visibilities
//     FFT UV gridded visibilities to image domain
//        to obtain image of sky as seen by telescope
//     reset UV grid to zero
//     transfer gridding weights to UV grid
//     FFT weights grid to image domain to get synthesized antenna pattern
//
main()
{
// decide on output image size
   int imsize;
   cout << "Enter image size (must be power of 2) : ";
   cin >> imsize;
// should actually check that image is power of 2 !!

// declare a 3-d image, could be different sizes in x,y and z in general
   Cube<float> image(imsize,imsize,imsize); image = 0.0; //uv grid <-> image

// set number of dimensions; here n = 3
   int n = 3;

// create a cellsize (in arcsec) - can be anything in this test program
   cout << "Enter cellsize (arcsec) : ";
   float cellsize;
   cin >> cellsize;

// create matrix to hold input parameters  for GridTool constructor
   Matrix<float> input_parms(3,n); input_parms = float(0);
```

```
// tell the grid-tool the dimensions of the image and the cellsize
//
   input_parms(0,0) = imsize;
   input_parms(0,1) = imsize;
   input_parms(0,2) = imsize;
   input_parms(1,0) = cellsize * C::arcsec;
   input_parms(1,1) = cellsize * C::arcsec;
   input_parms(1,2) = cellsize * C::arcsec;

// set up an array of math functions - one for each dimension
   MathFunc<float>** Mathptr = new MathFunc<float>*[n];
   for (int i = 0;i<n;i++) {
     Mathptr[i] = new Sph_Conv<float>();
     }

// Construct a GridTool object
   GridTool grid_tool<float,Complex> (Mathptr,input_parms);

// find out how many visibilities will be read in
//
   cout << "Enter number of visibilities to be read in : ";
   int no_vis;
   cin >> no_vis;

//
// create arrays to hold visibility locations and visibility values

   Matrix <float> uv_coord(3,no_vis); uv_coord = float(0.0);
   Matrix <float> uv_val(3,no_vis); uv_val = float(0.0);
//
// read in visibility data and store in previously created arrays
//
   int dummy1, dummy2;
   float dummy3;
   for (int m = 0; m < no_vis; m++){
     cin >> uv_coord(0,m);        //  U
     cin >> uv_coord(1,m);        //  V
     cin >> uv_coord(2,m);        //  W
     cin >> uv_val(0,m);          //  real component of visibility
     cin >> uv_val(1,m);          //  imaginary part of visibility
     cin >> uv_val(2,m);          //  weight of visibility
   }

// visibilities have been read in
// now create a cube to hold the uv grid weights
   Cube<float> wts(imsize/2+1,imsize,imsize); wts = 0.0;

// now do the gridding
   grid_tool.gridUV(uv_coord,uv_val, image, wts);
```

```
// pack uvgrid for transport to FFTServer object
   grid_tool.pack(image);

// initialize an FFTServer for "image"
   FFTServer<float,Complex> fft(image);

   fft.fft(image,0);            // FFT uv cube to image domain
                                //   -don't do any scaling
                                //   -scaling is done by dividing
                                //    by sum of weights

// correct for convolution effects
   grid_tool.convCorrect(image);

   float sum_of_weight = fft.wtsum(wts);
   sum_of_weight = 1.0 / sum_of_weight;
   image *= sum_of_weight;   // normalize image
// you would write out the image to disk here

// make an antenna pattern
   image = 0.0;
   fft.uvassign(image, wts);
   fft.fft(image, 0);
   grid_tool.convCorrect(image);
   image *= sum_of_weight;      // central peak better  be 1 !!
// you would write out the antenna pattern to disk here

// Whew! finished, I think

}
```

7.3 complex proto_map

We now take the proto_map example discussed in section 1 and assume that the sky image is complex. So we use gridding and FFT methods appropriate for a complex grid.

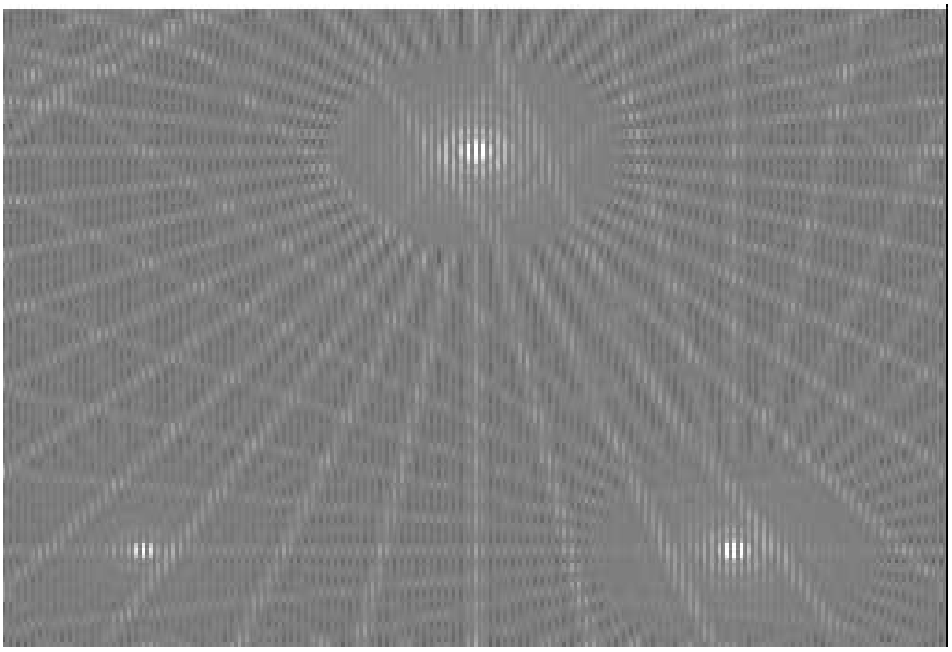
```
#include <iostream.h>
#include <aips/Math.h>
#include <iostream.h>
#include <aips/Constants.h>
#include <aips/String.h>
#include <aips/aips.h>
#include <aips/gridIO.h>
#include <aips/ArrayIter.h>
#include <aips/Matrix.h>
#include <aips/Vector.h>
#include <aips/GridTool.h>
#include <aips/FFTServer.h>
```

```
//
// This program tests the combination of GridTool and FFTServer
// methods for the case of complex images and full complex UV grids
//
int main()
{
   int imsize;
   cout << "Enter image size (must be power of 2) : ";
   cin >> imsize;

// declare a Complex matrix, could be different sizes in x and y in general
// this holds the uv grid <-> image
   Matrix<Complex> image(imsize,imsize);
   image = Complex(0.0);

// now enter some point sources (positions and flux densities)
// to create a map of point sources
   cout << "Enter point source flux density and x and y locations "<<endl;
   cout << "x and y must lie in range 0 to "<<imsize - 1 <<endl<<endl;
   float fluxdensity;
   int finished = 0;
   int x, y;
   cout <<" "<<endl;
   while (!finished) {
      cout << "Enter point source flux density (-999 to finish) ";
      cin >> fluxdensity;
      if (fluxdensity == -999.0)
         finished = 1;
      if (!finished) {
         cout << "Enter point source x and y location:";
         cin >> x >> y;
         if (x >=0 && x < imsize && y >=0 && y < imsize)
             image(x,y) = Complex(fluxdensity);
      }
   }


// create a cellsize (in arcsec) - can be anything in this test program
   float cellsize = 20.0 * C::arcsec;
// set number of dimensions; currently n = 2
   int n = 2;

// create matrix to hold input parameters  for GridTool constructor
   Matrix<float> input_parms(3,n); input_parms = float(0);

// tell the grid-tool the dimensions of the image
// and the cellsize
   input_parms(0,0) = imsize;
   input_parms(0,1) = imsize;
```

```
   input_parms(1,0) = cellsize;
   input_parms(1,1) = cellsize;

// set up an array of math functions - one for each dimension
   MathFunc<float>** Mathptr = new MathFunc<float>*[n];
   for (int i = 0;i<n;i++)
     {
     Mathptr[i] = new Sph_Conv<float>();
     }
// initialize a grid tool
   GridTool<float,Complex> grid_tool(Mathptr,input_parms);
   grid_tool.convCorrect(image);        // correction to image
                                        // required for degridding


// initialize an FFTServer for "image"
   FFTServer<float,Complex> fft(image); // constructor
// do full complex FFT to uv plane
   fft.cxfft(image,1);
// create some fake uv "tracks" for an east-west interferometer
// we will assume that we can have 100 spacings.
// each spacing will have 40 uv sample points, each sample point
// having a u and v value (uv_coord cube) and a cosine, sine and
// weight value (uv_val cube)
// we now need 40 sample points since we do not assume Hermetian
// symmetry
   Cube <float> uv_coord(2,40,100); uv_coord = float(0.0);
   Cube <float> uv_val(3,40,100); uv_val = float(0.0);
   int uvmax = imsize / 3 - 1;
   if (uvmax > 99)
      uvmax = 99;
   float delta_theta = C::pi / 20.0;
//
// create the uv cubes by going through them one layer at a time
//
   ArrayIterator<float> uv_iter(uv_coord,2);
   ArrayIterator<float> uv_iter_dat(uv_val,2);
//
// The next conversion allows us to take uv location in grid units and
// turn it into uv location in 'wavelengths' so that
// GridTool can reverse the operation!!
//
   float scale_factor = 1.0 / (C::asec2rad*input_parms(0,0)*cellsize);
//
// now compute the uv locations for each spacing
//
   for (i = 1; i <= uvmax; i ++) {
      Matrix <float> muv_coord(2,40); muv_coord = float(0.0);
      Matrix <float> muv_val(3,40); muv_val = float(0.0);
```

```
      for (int j = 0; j < 40; j ++) {
         float theta = -C::pi_2 + j * delta_theta;
         float v = i * sin(theta);
         float u = i * cos(theta);
         muv_coord(0,j) = u * scale_factor;
         muv_coord(1,j) = v * scale_factor;
      }
// degrid to get values of visibilities
      grid_tool.cDegridUV(muv_coord,muv_val, image, 1);
      uv_iter.array() = muv_coord;
      uv_iter_dat.array() = muv_val;
      uv_iter.next();
      uv_iter_dat.next();
   }
//
// visibilities have been created
// now make the map - first do gridding
//
   uv_iter.origin();             // reset the iterators
   uv_iter_dat.origin();

   image = float(0.0);           // reset the image array
   Matrix<float> wts(imsize,imsize); wts = 0.0;   //weighting grid
// do the gridding for each baseline
   for (i = 0; i < uvmax; i ++) {
      grid_tool.cGridUV(uv_iter.array(),uv_iter_dat.array(), image, wts);
      uv_iter.next();
   }

   fft.cxfft(image,0);            // FFT uv plane to image domain
// gridding completed, now do FFT
   cout <<"image has been fftd"<ndl; cout.flush();
                              //   -don't do any scaling
                              //   -scaling is done by dividing
                              //    by sum of weights
   grid_tool.convCorrect(image);  // correct for convolution effects
   float sum_of_weight = fft.cxWtsum(wts);
   image /= Complex(sum_of_weight);   // normalize fft
// write out the image to disk here

// make an antenna pattern
   image = Complex(0.0);
   fft.cxUVassign(image, wts);
   fft.cxfft(image, 0);          // FFT to sky domain
// max peak of 'image' better be 1 !!
   grid_tool.convCorrect(image);
   image /= Complex(sum_of_weight);
 // write out the antenna pattern to disk here
}
```

If we run this program using the same parameters as for program `proto_map` the result would look like



Here every second value is imaginary, and all imaginary values are zero, so the map looks like the result for `proto_map` but with vertical stripes of zero in between each real value.