Imaging Pipeline Software

Bobbie Ware 17962233

Math705 Research Project

Abstract

Aperture synthesis is the process of taking data from interferometry telescope arrays and producing an image of the sky. This research project covers the gathering of knowledge on the required topics, and then using this knowledge to create software capable of performing these processes. The software will be tested against ideal models to ensure the correct results are being produced, and multiple versions of the software will be created to improve runtime performance.

Acknowledgements

This report acknowledges the assistance of Andrew Ensor who was the supervisor for this research project. He came up with the idea for the research and developing a imaging pipeline and it fit the sort of project desired exactly. Andrew was also extremely helpful when discussing any part of the pipeline and was a great help in solving any issues.

Adam Campbell, also from AUT, deserves a high amount of praise for the assistance in generating the data used, providing help with code at multiple points in the pipeline development, and also providing the code for rendering the images.

Table of Contents

[Abstract 2](#_Toc12009446)

[Acknowledgements 3](#_Toc12009447)

[Introduction 6](#_Toc12009448)

[Literature Review 7](#_Toc12009449)

[Synthesis Imaging 7](#_Toc12009450)

[Fast Fourier Transform (FFT) 9](#_Toc12009451)

[Gridding 10](#_Toc12009452)

[Deconvolution 11](#_Toc12009453)

[Methods 13](#_Toc12009455)

[Design of pipeline 13](#_Toc12009456)

[Implementing 14](#_Toc12009457)

[Testing and Improving 14](#_Toc12009458)

[Software Implementation 16](#_Toc12009459)

[Gridder 16](#_Toc12009460)

[Inverse Fast Fourier Transform 17](#_Toc12009461)

[Deconvolution 17](#_Toc12009462)

[Results 19](#_Toc12009463)

[Gridding 19](#_Toc12009464)

[Fourier Transform 20](#_Toc12009465)

[Deconvolution 22](#_Toc12009466)

[Summary of the versions 23](#_Toc12009467)

[Version 1 23](#_Toc12009468)

[Version 2 23](#_Toc12009469)

[Version 3 23](#_Toc12009470)

[Results of Testing 24](#_Toc12009471)

[Discussion of Results 26](#_Toc12009472)

[Output Comparison 26](#_Toc12009473)

[Deconvolution Output 26](#_Toc12009474)

[Development of Versions 27](#_Toc12009475)

[Concurrent Updates issue 28](#_Toc12009476)

[Analysis of Runtimes 28](#_Toc12009477)

[Gridder Runtime Tests 30](#_Toc12009478)

[Conclusion 31](#_Toc12009479)

[Synthesis Imaging 31](#_Toc12009480)

[Gridder 31](#_Toc12009481)

[Fourier Transform 31](#_Toc12009482)

[Deconvolution 31](#_Toc12009483)

[Performance Improvements 31](#_Toc12009484)

[References 32](#_Toc12009485)

[Appendix 34](#_Toc12009486)

Introduction

An imaging pipeline for Image Synthesis is designed to take data either gathered by radio interferometry telescopes, or generated to simulate those telescopes, and form an image of an area of the sky. This software will be similar to that used for the Square Kilometer Array – a project AUT is involved in.

This project will follow a Design Science methodology where software will be the generated artefact used to experimentally investigate image synthesis. The project will start with the gathering of knowledge on the techniques used in an imaging pipeline. It will also involve gaining knowledge on solutions dealing with complications such as concurrency control and the mapping of visibilities to a grid. The project will involve implementing the techniques in the form of a Java program, and then will be tested using visibility input data available in the High Performance Computing Research Laboratory. The software will be tested against other versions and changes will be made to try to improve its performance.

The output from the developed software, using visibility data as input, will be an image of the sky. The images produced by the pipeline will be analysed to compare it against the known sky images for the data sets to validate whether the techniques are implemented properly and potentially look at its performance.

It is expected that an imaging pipeline will be developed with the capability of image synthesis. Also expected is to gain knowledge in the three main steps involved in the pipeline, namely gridding, (inverse) Fourier transform, and deconvolution. As well as some techniques for algorithm optimization.

Literature Review

## Synthesis Imaging

The resolution of radio telescopes can be increased by using pairs of telescopes (baselines) and taking the product of the received signals. This resolution can be changed by increasing the separation of the baseline, rather then increasing the size of the individual telescopes. This method popularized by the work of (Ryle & Hewish, 1960) states that using these baselines the telescopes produces; “exactly the same result as that obtained by using the complete large aperture”. This technique allowed for cheaper production of much larger apertures and the eventual development of the techniques used now.

These techniques gather Fourier domain data in the form of a visibility. However, the way in which they are sampled is non-uniform, so it must be placed on a rectangular grid. This process is known as gridding and the methods used now are based on the work by (Brouw, 1975). These visibilities V(u, v) fall upon the plane in which the baselines are setup. For a wider coverage of this plane, more baselines can be added and could also be moved around.

With more modern telescopes being developed, moving them around became a substantial task and instead the rotation of the earth can be used to move these points around the plane. An image of these points on the V(u, v) plane can be seen on the left of Figure 1.

As there are gaps within the V(u, v) plane the image is a “dirty image”, this can be seen on the right of Figure 1.

Synthesis imaging is a large research field with many discoveries being made, a recent image produced of a black hole received a lot of public attention. Reading this paper (The Event Horizon Telescope Collaboration et al, 2018), showed insight into the process of developing an imaging pipeline.

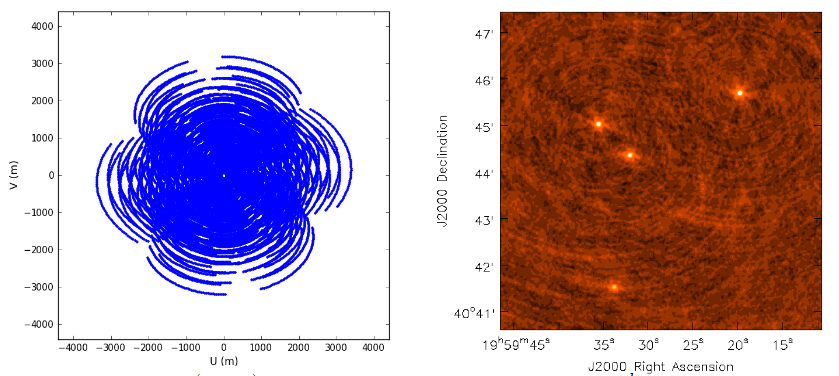


Figure 1. V(u,v) plane showing data points and a dirty image.

**Reference**

Rau, U. (2012, Sept 24). *Imaging and Deconvolution.* Retrieved from Australia Telescope National Facility: https://www.atnf.csiro.au/research/radio-school/2012/lectures/tue/RVU\_ImagingDeconvolution.pdf

## Fast Fourier Transform (FFT)

A Fourier Transform is a process for signal-processing and analysis. (Brigham, 1988) states that the extent of the use the process is as follows; “biomedical engineering, imaging, analysis of stock market data, spectroscopy, metallurgical analysis, nonlinear systems analysis, mechanical analysis, geophysical analysis, simulation, music synthesis”. It is widely regarded as one of the most important algorithms because of its impact in so many areas.

Simply put a Fourier Transform is used to show different parts of a continuous signal. However, for Interferometry an Inverse Fourier Transform is used as we are taking the data from the Fourier Plane and creating an image from it.

While it is possible to perform a basic Fourier Transform using computers, the method has a run time of O(). Instead, we can perform a Fast Fourier Transform with a run time closer to O() as it is a divide and conquer method. Such a method is based upon the work of (Hogg, MacDonald, Conway, & Wade, 1969). The algorithm used was first discovered by Gauss and later rediscovered by (Cooley & Tukey, 1965) which notes that, “Wherever possible the use of N = with r = 2 or 4 offers important advantages”. This algorithm works under the assumption that the data is in an organized array, hence the visibilities must be gridded. Using butterfly operations, the data is combined in pairs using either a decimation in time or in frequency variation.

## Gridding

Gridding is the process of mapping the data collected onto a rectangular grid so that it may be processed by the iFFT, and then displayed as an image. Early techniques for placing the visibility data on a grid involved finding the closest grid point to visibility’s V(u, v) co-ordinate. Then, either adding them all together or averaging them out on that grid location. Early methods were used by (Hogg, MacDonald, Conway, & Wade, 1969). However, their method produced many artifacts in the image. Therefore, there were limited application for the process. An alternative method used by (Brouw, 1975) would take a weighted value based on the distance between local grid point and the point of the visibility. By designating a “support” area around the local grid point, the data can be added to these areas. An ideal gridding method was given by (O'Sullivan, 1985), with his gridding algorithm that used a sinc function and produced images with “arbitrarily small artifact levels”. However, this function would give infinite extent to the support. This is not ideal computationally for the gridder. Instead, convolution kernels with a set support area are used. These functions also have a quick fall off in the grid to help aliasing.

A simplified approach to gridding follows this process; for every visibility, find the closest grid point to the data on the V(u, v) plane. Then, using the convolution kernel, the data point is spread across the support region. By using a convolution kernel generated by a Prolate Spheroidal, the value of the data point is spread based on its distance to the center.

## Deconvolution

Once an image is formed from the Fourier Transform is it called a ‘dirty image’, as seen in Figure 1, this is due to the effects of having limited sampling of the V(u, v) plane. The process of Deconvolution can be used to ‘CLEAN’ the image. This method by (Högbom, 1974) uses the original V(u, v) data to form a “dirty beam”, then by taking away the dirty beam from the points of the dirty image with the greatest brightness you are left with a residual image. By iteratively carrying out this process the effects of the convolution are removed to the best extent possible.

This was expanded upon by (Clark, 1980) to make it more efficient. His method involves using more FFT’s in a major and minor cycle to subtract points away from the dirty image. The minor cycle works by performing a Högbom ‘CLEAN’ on smaller beam patches, then the major cycle applies an FFT on the points found by the minor cycle and is used to subtract from the dirty image. There is also a varient of this process where (Schwab, 1984) uses the major cycle to take away from un-gridded visibilities. This helps to remove noise from and potential errors from the gridding process.

Further examples of the ‘CLEAN’ algorithm and the Maximum Entropy technique by (Skilling & Bryan, 1984) are compared by (Cornwell & Bridle, 1996).

The image in Figure 2 shows the ‘CLEAN’ image based on the images from Figure 1.

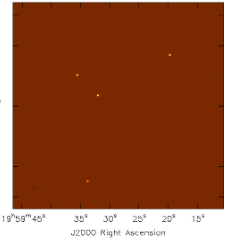
**

Figure 2. A ‘CLEAN’ image.

**Reference**

Rau, U. (2012, Sept 24). *Imaging and Deconvolution.* Retrieved from Australia Telescope National Facility: https://www.atnf.csiro.au/research/radio-school/2012/lectures/tue/RVU\_ImagingDeconvolution.pdf

Methods

The goal of the project is to gather sufficient knowledge on the topics required to produce software capable of taking data from interferometry telescopes and produce an image of the sky. Starting with a literature review, relevant knowledge regarding the topics of Image Synthesis, gridding, Fourier Transforms, and Deconvolution were gathered. Using this knowledge, an artefact will be produced that to perform the required processes. Once the pipeline is created, it will be tested against a perfect image of the sky to ensure that it is correctly carrying out the processes involved. The pipeline will then be adapted and improved upon to increase its performance.

## Design of pipeline

The pipeline will consist of three main sections; the Gridder, the inverse Fourier Transform, and the Deconvolution. The programming language used to develop the pipeline was Java. Java is primarily taught in computer science papers at AUT and is capable of handling the size of the data used. It also supports parallelization, a feature that will be implemented to improve performance.

The grid size to be used will be 1024x1024. Providing a high enough resolution to be able to properly test against. The grid length and height must be a power of 2 due to the (Cooley & Tukey, 1965) radix 2 FFT being used. This algorithm is more performant then a standard Fourier Transform under the condition that the data is regularly spaced, hence why the visibilities are gridded. There are two different ways of implementing an iFFT; decimation in time, or decimation in frequency. Both methods are explained and compared in the software implementation section.

The data contains Visibilities.csv which is a file that includes around 23000 data points. Each visibility consists of its locations along the V(u, v) plane, followed by its value as a complex number. The dataset also includes the Prolate Spheroidal which is used as the convolution kernel for the data with a support size of 7. However, for better precision it has been oversampled by a factor of 4. The last thing required from the data is the configuration for achieving the best accuracy for the gridder.

A double is used as the primitive data type, as in the Java language (Oracle, 1993) a double can store values from 4.94065645841246544e-324 to 1.79769313486231570e+308 including values with high decimal accuracy needed.

For Deconvolution, Högbom’s ‘CLEAN’ method will be implemented to deconvolve the true image out of the dirty image. To achieve this, a dirty beam must be constructed from the V(u, v) plane.

## Implementing

The process of creating the pipeline will undergo many stages. This is due to multiple iterations of the software being produced as improvements are made. When the pipeline is complete, a fully formed Java project with unit testing and multiple output images will be produced.

When implementing the pipeline, constant reference to the literature will be required to ensure the processes are carrying out their operations correctly. The nature of the mathematical operations mean that high precision is needed for a meaningful image to be produced.

For a more in-depth explanation of the methods involved in the implementation, please see the following section.

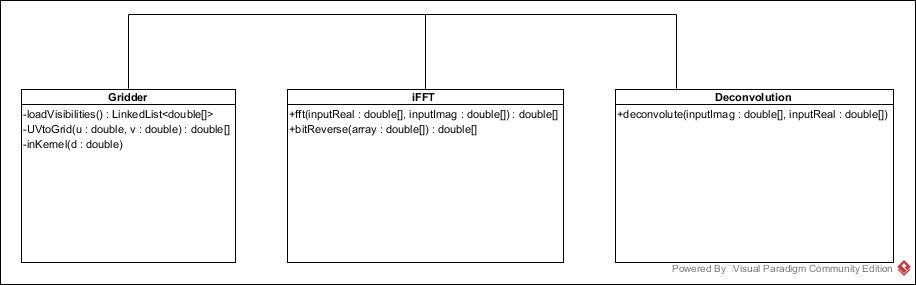
## Testing and Improving

To test the pipeline, a dataset generated by the HPCRL at AUT will be used. By using this data, the three main processes of the pipeline can be checked to see how they are performing as it can be tested against a perfect image made from the real components of the sky. This perfect image is what the pipeline would generate if working under ideal circumstances, so any inconsistences would be evidence that improvements can be made.

As the pipeline is tested, improvements will be made with respect to its runtime. An example of this is using multiple threads to perform the gridding or the Fourier Transform. However, implementing multiple threads brings issues such as concurrent updates from the threads. A concurrency issue occurs when two threads attempt to update the same values simultaneously and only one update is saved. This would arise with overlapping visibilities in the gridder and requires a solution such as a lock on each value to prevent being accessed at the same time. Another method is to have each thread manage its own grid, combining each grid after all the visibilities have been processed.

Software Implementation

The implementation will be broken into three classes. Gridder.java, which is responsible for reading the dataset and placing the points on a grid. iFFT.java, for carrying out an Inverse Fourier Transform on the data. Deconvolution.java, implementing a ‘CLEAN’ algorithm to turn the dirty image into a more representative form. Lastly, a python script will be used to display the image. This script was given by the HPCRL at AUT. Below is a UML diagram of the Java classes.



## Gridder

Implementing the gridder begins with initializing two 2D double arrays. These are used to store the real and imaginary values respectively. As the values are placed on the grid they will be stored in these arrays. Following that, the gridder loads the data from the .cvs file and stores it as a double array. The first two indexes are the u and v coordinates, after that is the real and then imaginary values. These double arrays are placed in an array list data structure to ensure the size of the list is not an issue. It also allows for the data to be spilt up when needed. Then going through the list, each array is transformed as per the formula mentioned before.

In order to correctly place the visibilities, two things must be used; a wavelength to meters ratio, and the UV scale. For the data generated, a frequency of 300000000 was used. To calculate the scale, we use the product of the grid size and the cell size. As discussed earlier, our grid size is 1024, and cell size is 4.848136811095360e-06 given by the dataset.

Once the visibilities are modified correctly, the process of placing the points on the grid begins. Firstly, we take the location modified by the UVScale and round it to the nearest integer to find the closest grid point. Then, using the support value of 7, we take a 7x7 grid around this center grid point as this is where the data point will be placed. Using our convolution kernel, we form a 2D array 27x27, due to our x4 oversampling, and finding the distance of the grid point to the true visibility point.

## Inverse Fast Fourier Transform

The iFFT will be used on a 2D array. This process involves transforming every row in the image, followed by transforming every column. A decimation in time variation will be used so the arrays will be bit reversed first. A bit-reversal is done on the inputs as the in-place operations imply a bit-reversed output. The decision between decimation in time or decimation in frequency is down to preference, it has no impact on the performance of the algorithm. Both carry out the same operations, just in a different order. For decimation in time, a bit reversal occurs first, and then progressively larger distance in the butterfly operations. Decimation in frequency is when the butterfly operations get shorter in distance and then the array is bit reversed. For this implementation, decimation in time is used. There is no advantage in using one technique over another as both carry out the same computations.

The process involves one complex multiplication, addition, and subtraction. A complex multiplication consists of two real additions and four real multiplications. Complex addition and subtractions involve two real additions. For the transform, the real and the imaginary parts are spilt into two separate arrays. This is done to avoid having to use a separate object to store them.

## Deconvolution

For this process, a Högbom ‘CLEAN’ will be implemented. This process will remove any artifacts and leave only point sources in an otherwise empty field. For this process, a dirty beam must be formed. To obtain a dirty image, the process of gridder is carried out using the original visibilty data but instead of the values given, each point is set to have a value of 1. This works by finding points of high brightness within a number of regions in the dirty image. Then, by taking away a set amount of this point using the dirty beam, and adding the point to a set of clean components. This process is done iteratively in each region until no more points that match the required brightness can be found. By using these components, a restored image can be formed. This image is only an estimate of the sky.

Results

The implementation of the pipeline resulted in three different versions of the software, the discussion of the results will first cover the output from the pipeline and then the improvements made.

## Gridding

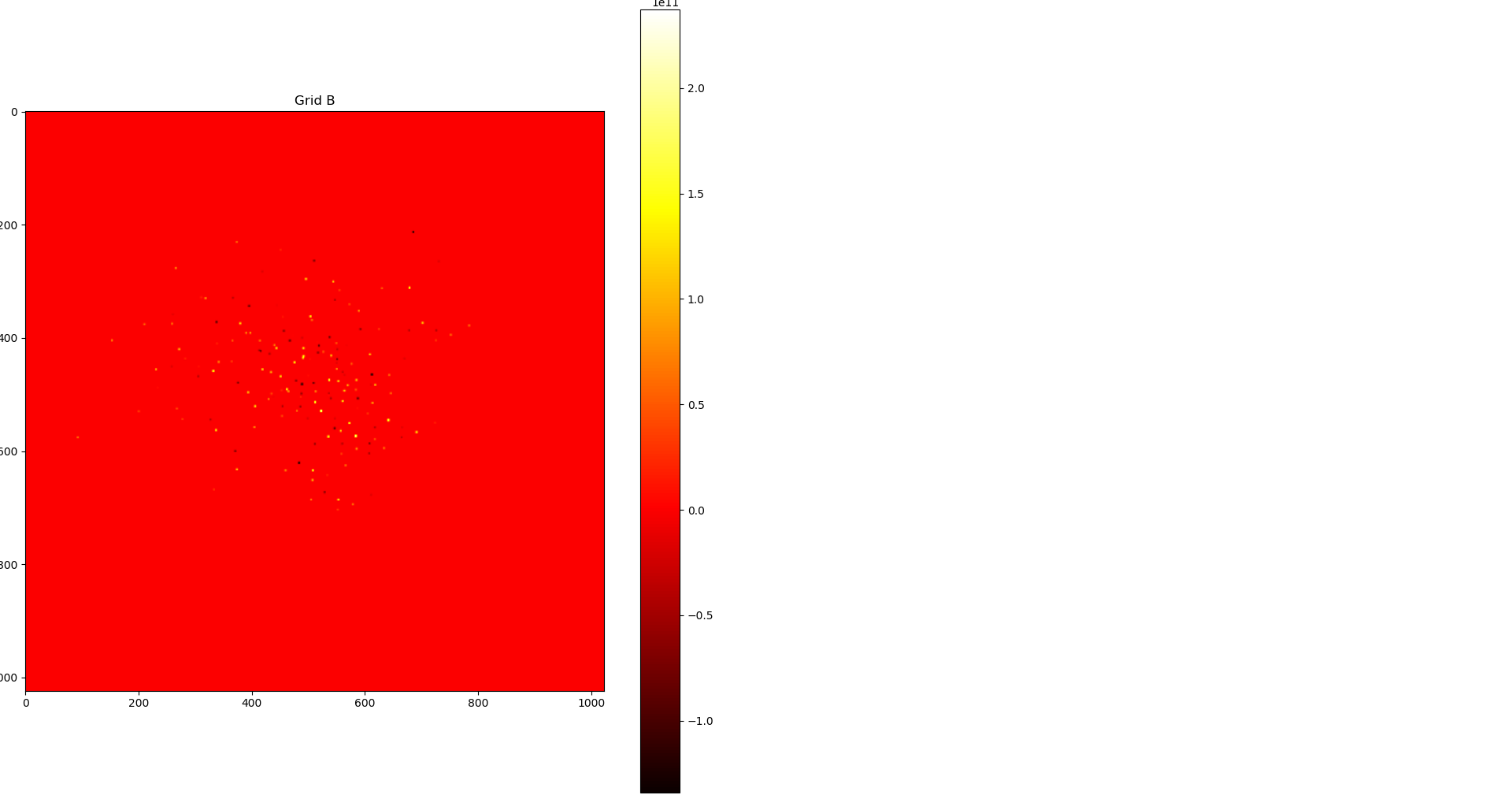
The gridder was able to successfully place all 223000 visibilities onto the grid. The first version of the gridder used a complex data type for each pixel on the grid, this was the simplest way of implementing the correct process for gridding. The image produced from the gridder can be seen in Figure 3, and a closer look can be seen in Figure 4.

Figure 3. The image produced by the gridder

As we can see in Figure 3 small points of positive or negative values are spread across the image. To compare the accuracy of the gridder the inverse Fourier transform must be carried out and if the image is accurate then the gridder is functioning correctly. What we can see is working for the gridder is the shape of the prolate spheroidal, in Figure 4 we can see round shapes of strong values that fall off around the edges. When the gridder is given a position for a visibility, it only points to one pixel, therefore a large amount of shapes with a size of around 7x7 pixels means that it has correctly spread the values.

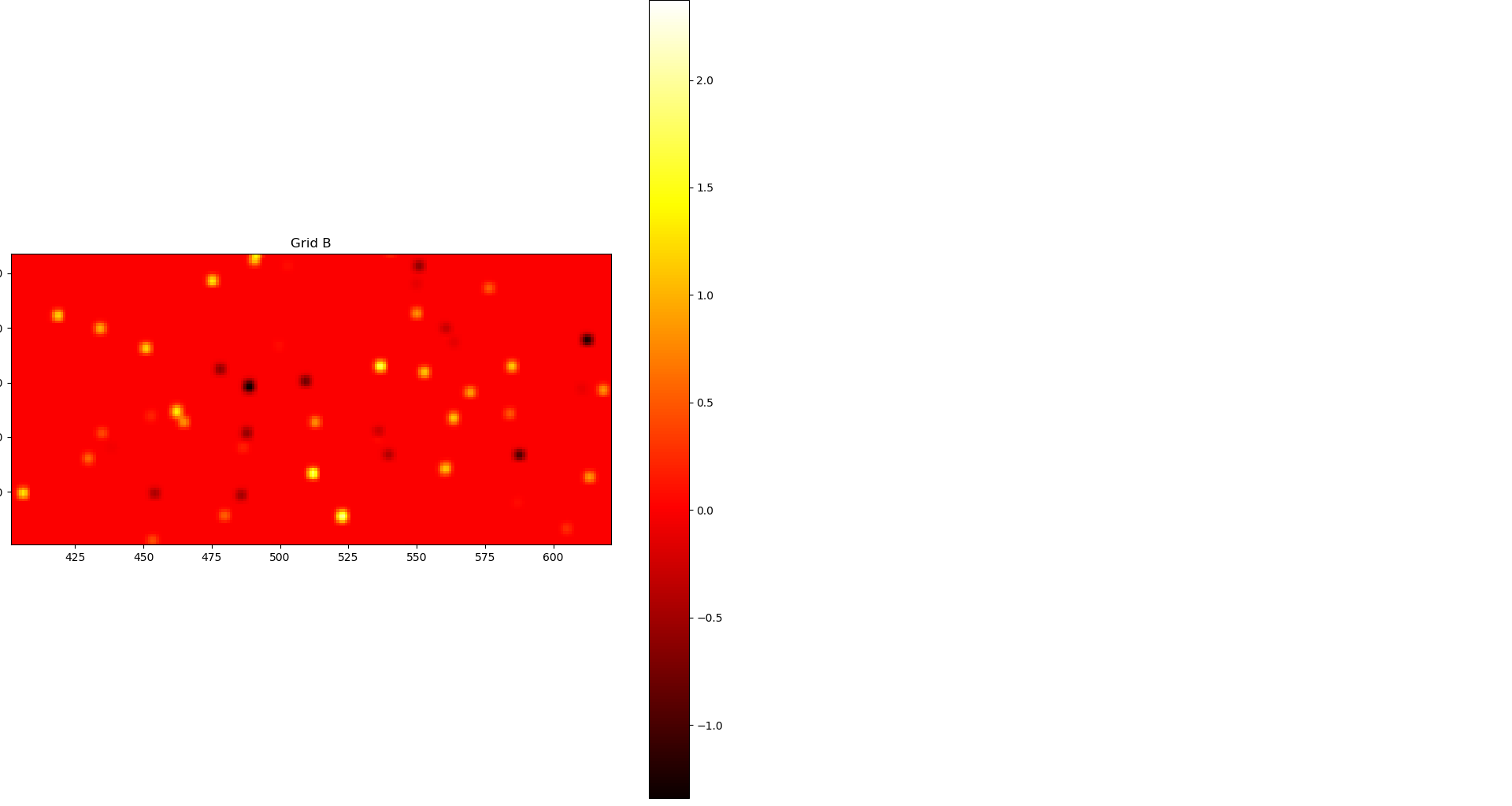


Figure 4. A closer look at part of the image in Figure 3.

## Fourier Transform

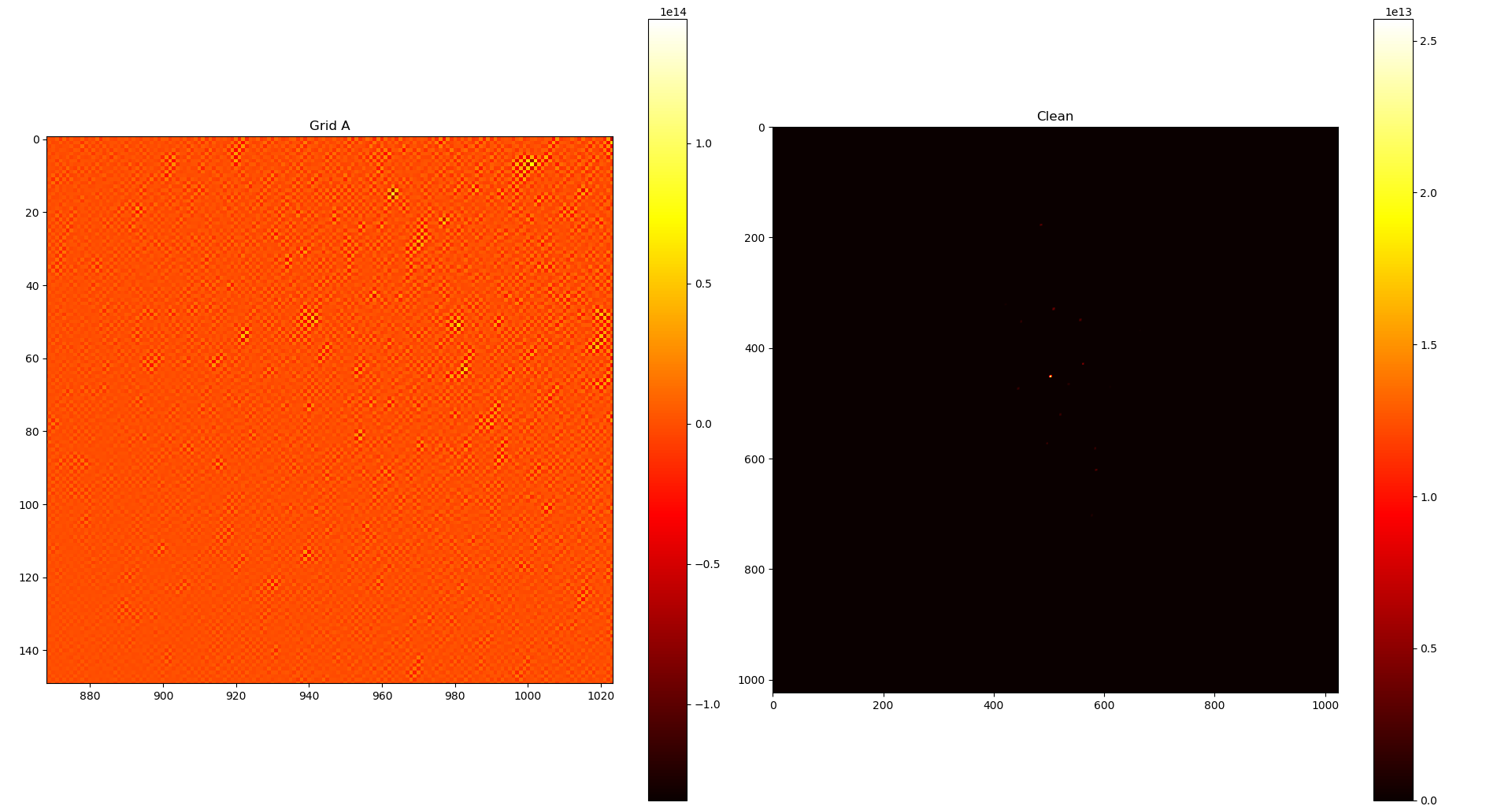
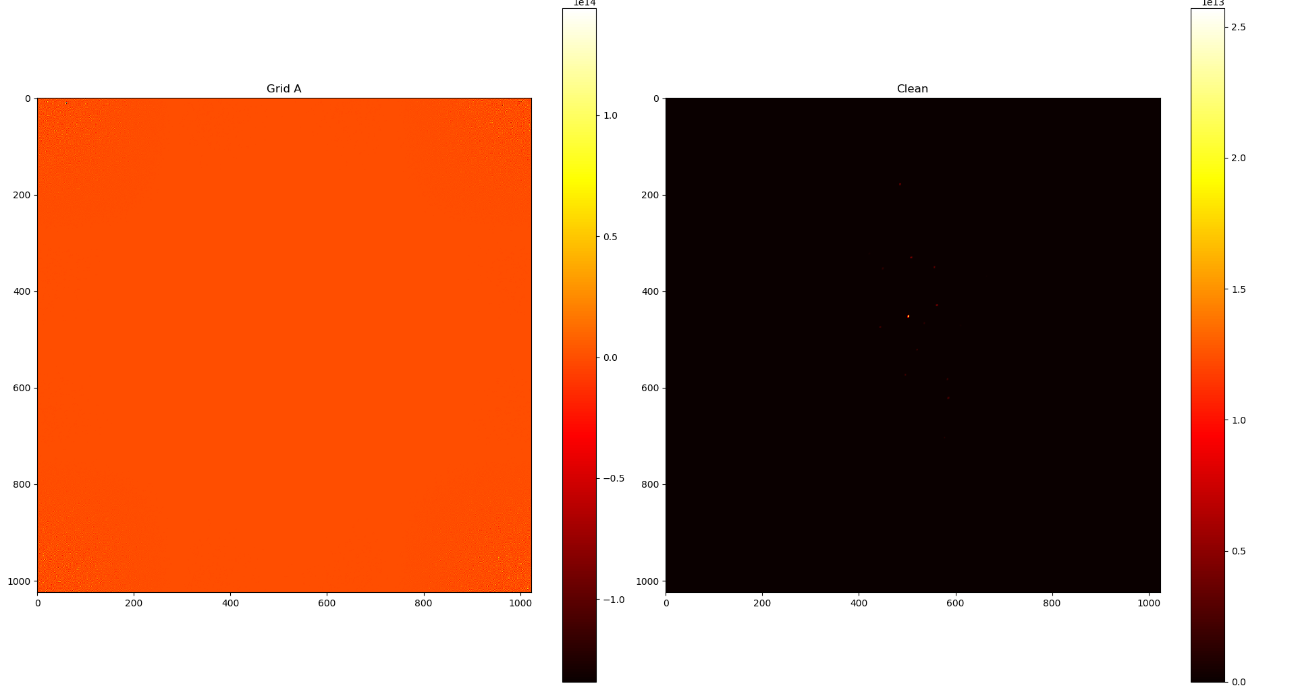
The implementation of the transform took the longest to code, it is not only the most complicated algorithm used, it also requires a lot of manipulation of the grid before and after the transform. For the specific algorithm used, all the data must be shifted so that the outmost corners are placed in the middle of the image. This is done before and after the transform. To illustrate this, the shift hasn’t been done on Figure 5 but has on Figure 6.

Figure 5. Without shifting the image is transformed incorrectly and the bright points are in the corners (Left). A zoomed in look showing the top left corner (Right).

A screenshot of a computer

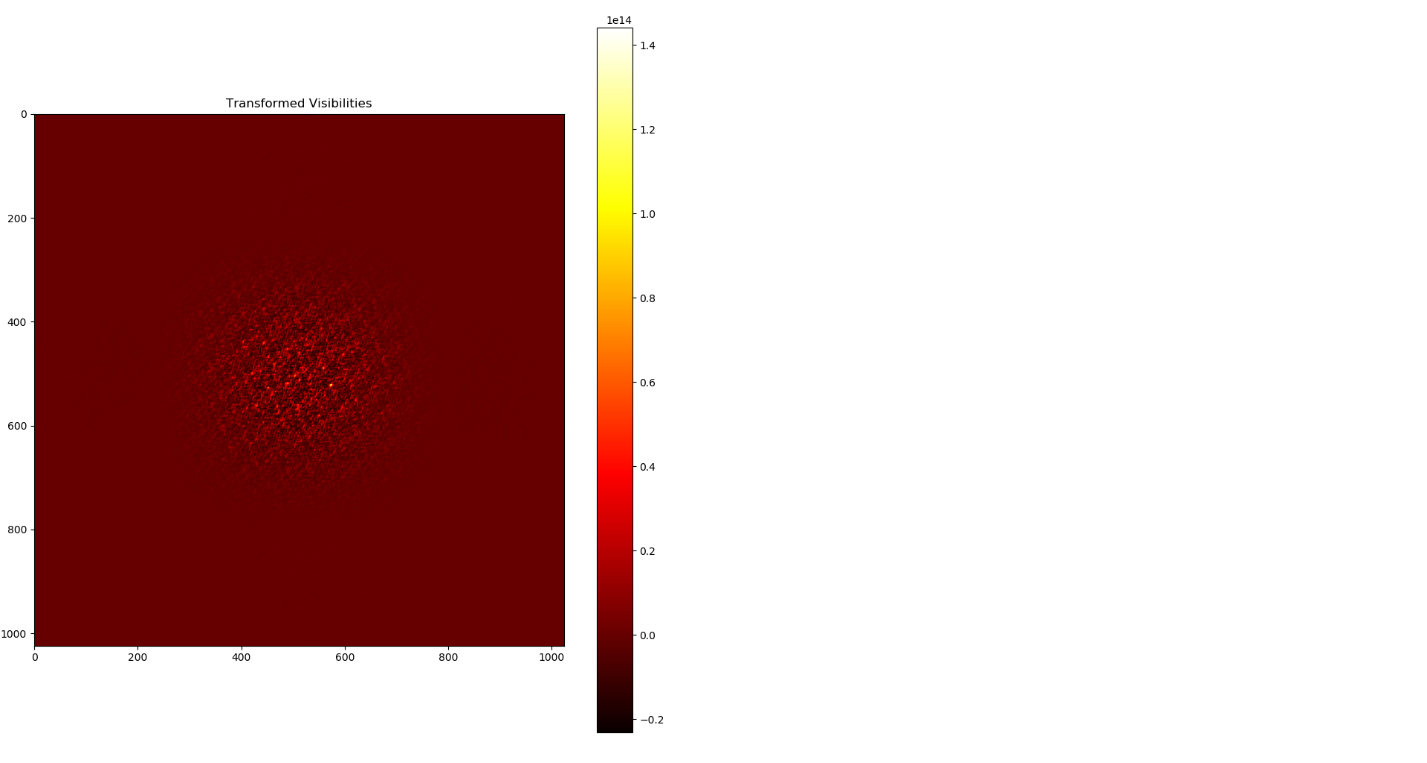
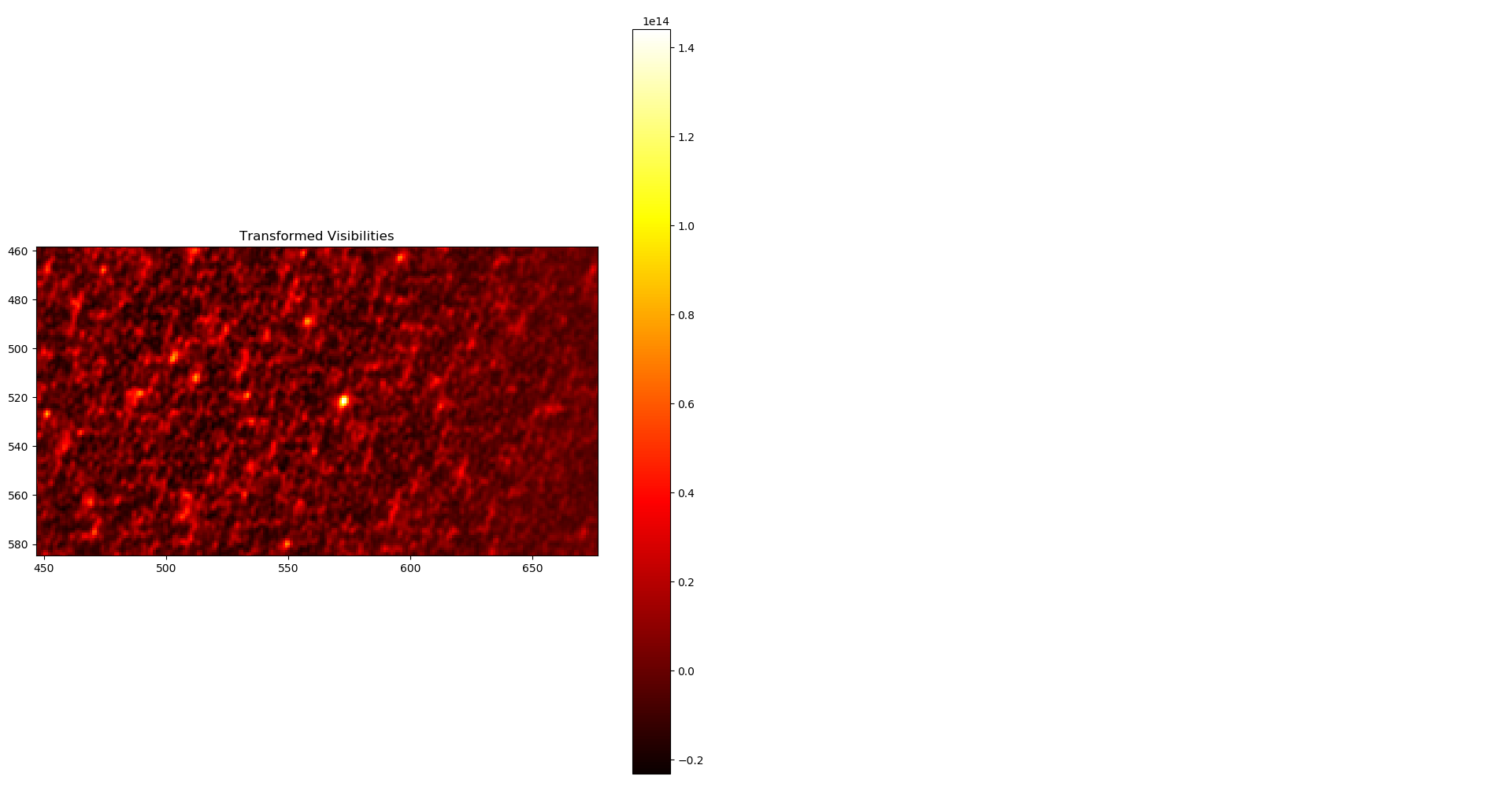
Description generated with high confidenceAs seen in Figure 7, a bright source is found near the center of the center of the image, this is potentially a star within the area of the sky imaged. The result of the transform then has a convolution correction applied so the corners are as bright as the center of the image. The image has the correct appearance for an inverse transform using our data, so we can assume the gridder and the Fourier Transform are implemented correctly.

Figure 7. Image with convolution correction applied.

Figure 6. Grid has been shifted before and after the transform (Left). A zoom in look of the middle of the image (Right).6

## Deconvolution

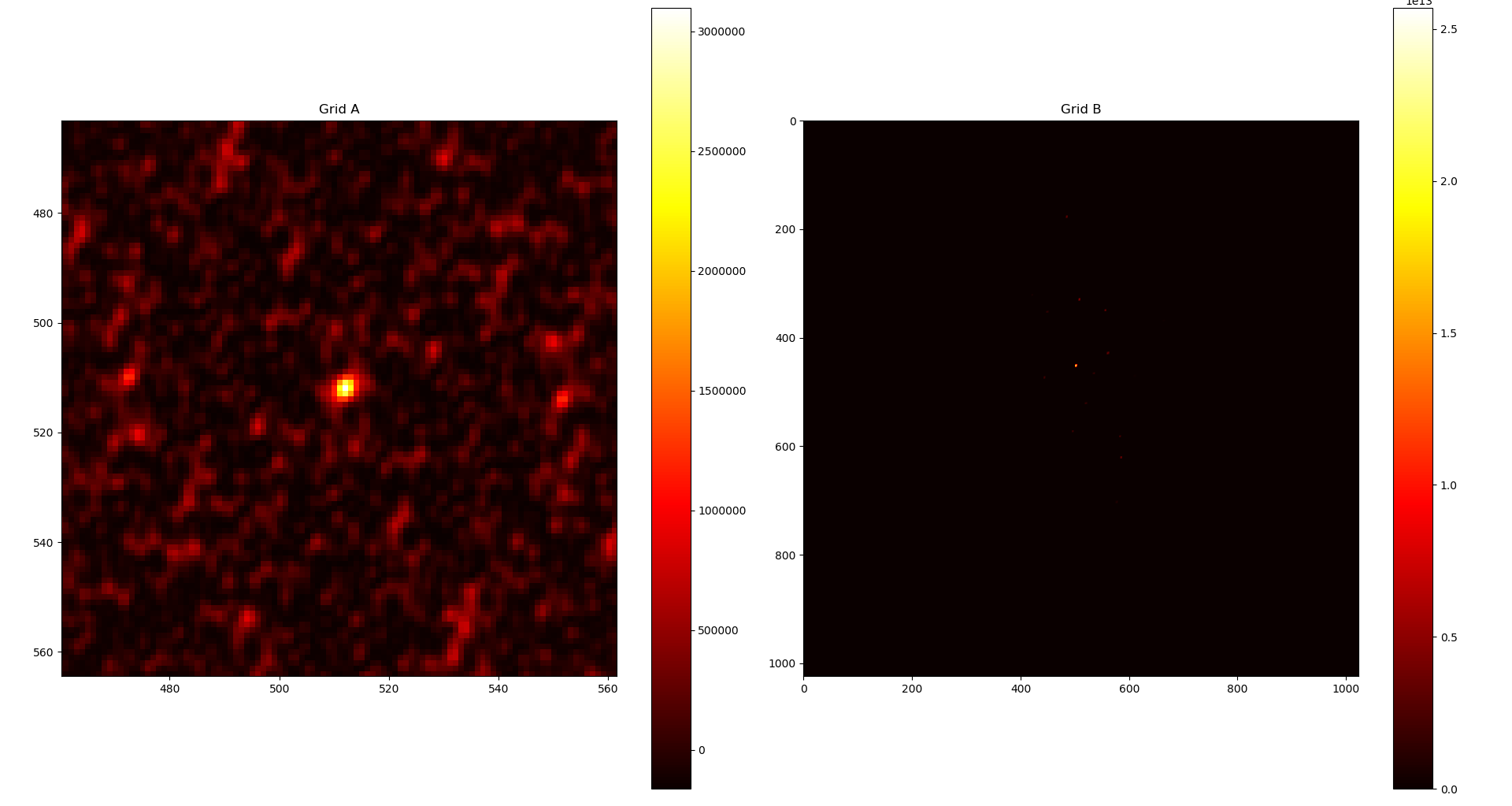
In order to carry out deconvolution of the image from the transform, we must construct a dirty beam. We get this by carry out the gridder and the Fourier transform again but instead of using the large complex numbers given, each point has a value of 1. When this process is done the dirty beam seen in Figure 7 can be seen.

Figure 8. Dirty beam found at center of the produced image

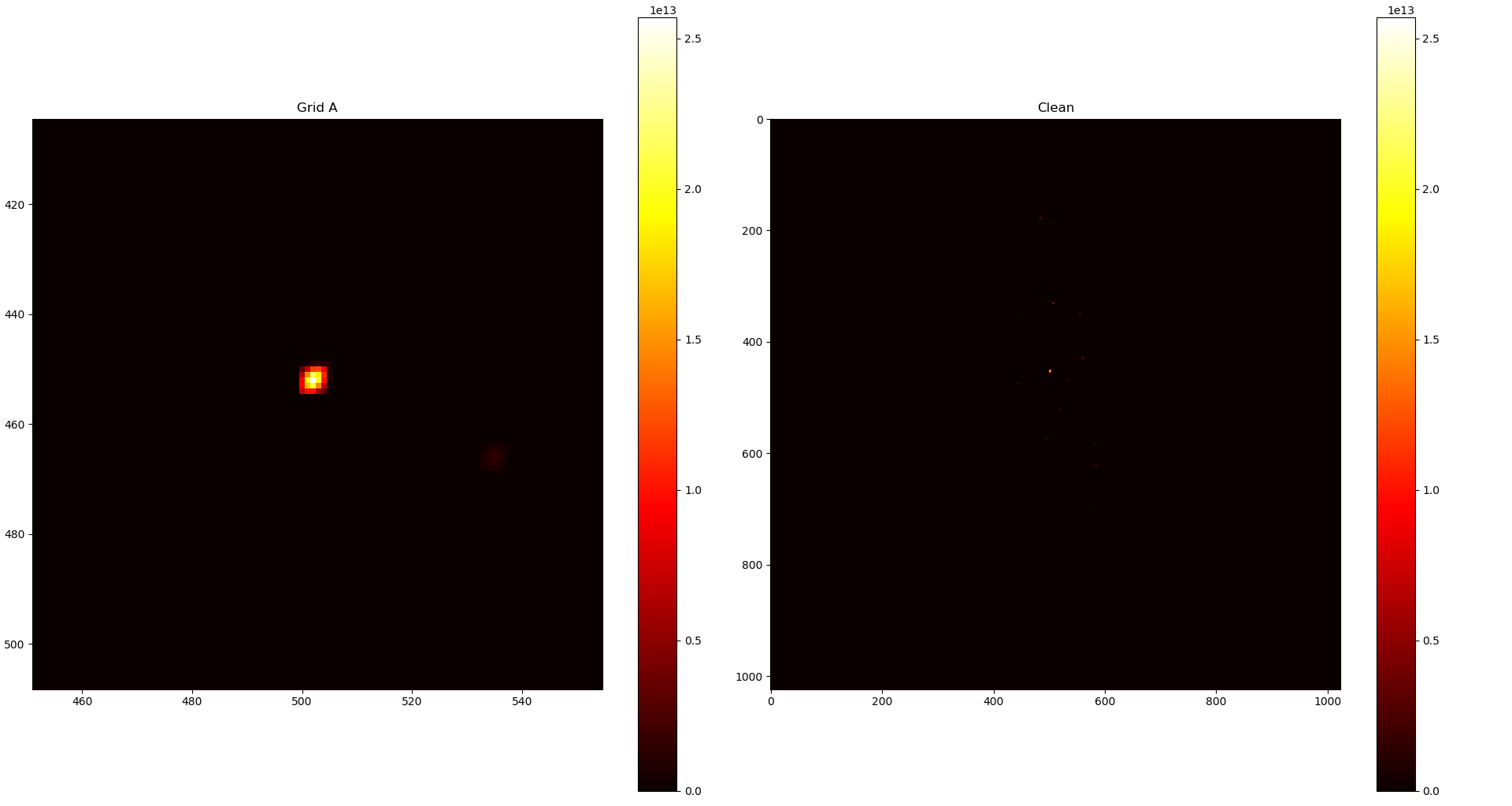
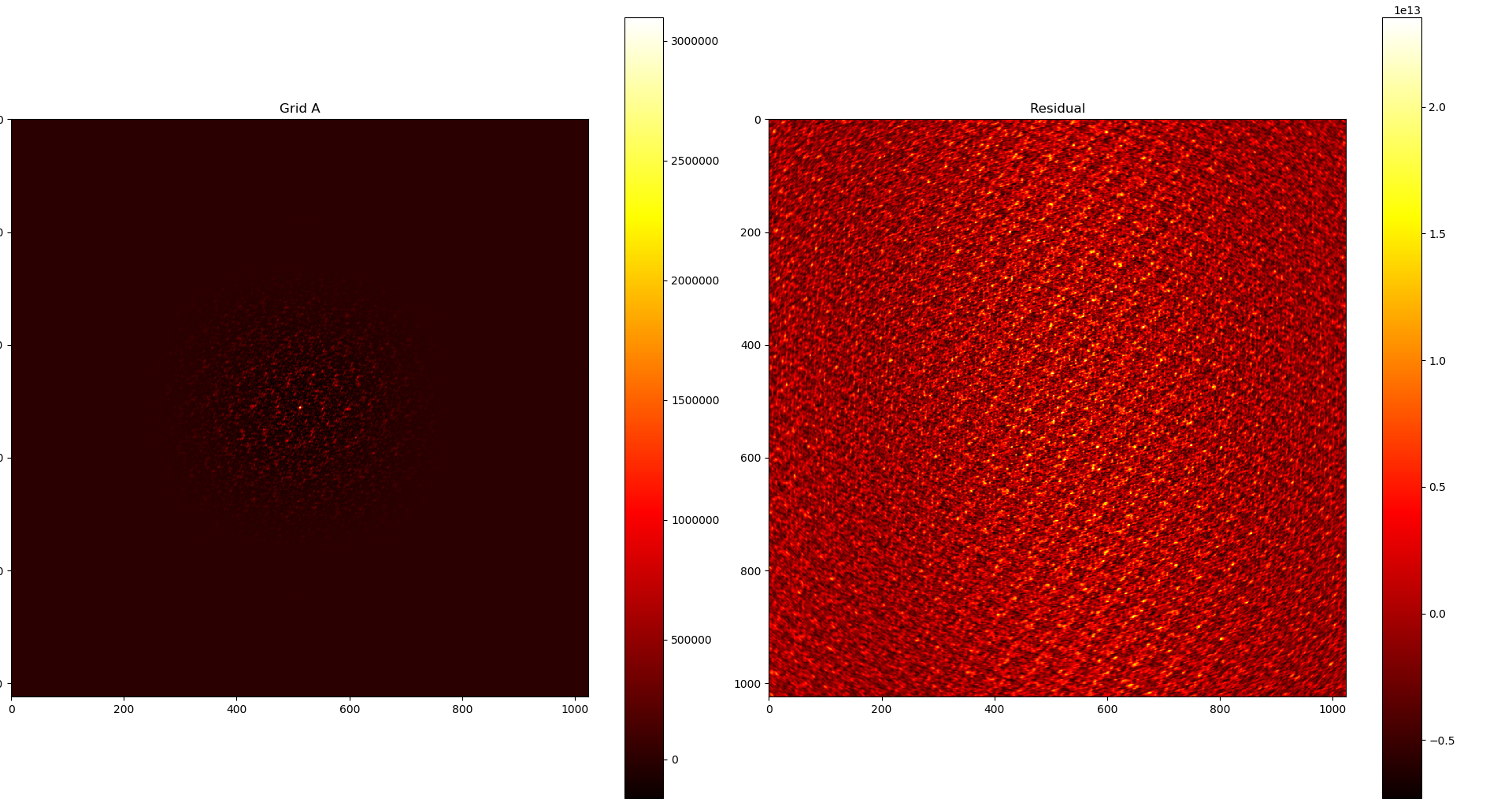
This shape is subtracted from the previous image, leaving a grid with only residual data, this can be seen in Figure 8. The CLEAN image can be seen in Figure 8, this was produced with 1000 iterations of the Högbom clean.

Figure 9. Residuals from CLEAN algorithm

Figure 10. A zoomed in section of the clean image, the whole image is Appendix A.

## Summary of the versions

### Version 1

The first version of the pipeline used a complex data type to store each pixel in the grid. This was done for simplicity when creating the first version. For the functions such as adding, subtracting, and multiplying, the code was contained within the data type. Throughout all versions of the pipeline an assumption to not overwrite data is present, this involves the creation of multiple 2D array of complex objects. This version is referred to as *UsingComplex*.

### Version 2

Instead of using a Complex data type, the second version of the pipeline uses two double data types to represent a complex number. This involves having two separate 2D arrays, one named *real* and the other *imaginary* (imag for short). In order to implement the grid in such a way, all of the complex operations have to occur slightly differently with no functions being stored in a data type. This pipeline was named *Using2Grids*.

### Version 3

For the final version of the pipeline implemented multithreading for the gridder and the Fourier Transform stages. This was to improve performance and compare against previous version. This version required the most development time as it is the most complex. Most of this complexity come from concurrent update issue with the gridder. The transform was easier to implement as each thread deals with a separate array at a time. These arrays are spilt evenly amongst the threads and there is no overlap between the threads. This version is referred to as *UsingMultipleThreads*. It also uses the double 2D array method (used by *Using2Grids*) as it was found to be more performant.

## Results of Testing

All of the versions of the pipeline produced the same output. This was expected as they carry out exactly the same operations just in different ways. Using a perfect image output that was produced when the data was generated we can compare how different the two are. The relative difference between the two can be calculated using the same python script used to render the images. The difference between the produced image and the perfect image is 1.091%.

To compare runtimes, each pipeline ran through the entire process 100 times on the same system and under the same conditions. PC and software specs can be found in Appendix B.

|  |  |  |  |
| --- | --- | --- | --- |
|  | *UsingComplex* | *Using2Grids* | *UsingMultipleThreads* |
| Mean (nanoseconds) | 709,178,183.1 | 568,478,821.6 | 471,953,759.3 |
| Standard Deviation (nanoseconds) | 19,570,930.35 | 24,486,395.06 | 38,638,593.29 |

Figure 11. Table of runtimes

The gridder was also tested independently, with this data set only 223,000 visibilities were used when a typical pipeline would use closer to 3,000,000 visibilities. To test the gridders thoroughly, they were given sets of visibilities of increasing size. The data sets given began with 100,000 and went up to 1,000,000, this will show the differences in how the runtimes scale with size. Figure 10 shows the results of the testing and the raw data can be found in Appendix C.

Figure 12. Graph shows the increasing runtime (nanoseconds) it takes for the three gridders to process data of increasing size.

Discussion of Results

## Output Comparison

The image produced by the pipeline is very close to that of the perfect image. It cannot be exactly same because the perfect image was produced under perfect gridding conditions. The gridders produced for the pipeline use techniques suitable for most situations.

The 1.091% is well within the acceptable parameters for the pipeline, this can be seen visually as well, both images have the same characteristics in the location of the bright point and shape of the pattern made by the Fourier Transform. One difference that can be seen in Figure 7 when compared to Figure 6 is that the image has been rotated after the convolution has been applied, this is to match the configuration of the perfect image and a simple transformation to apply.

When the deconvolution is applied to the perfect image an almost identical output is created. this is due to as much of the convolution applied by the gridder being removed, most of the difference in image comes from the difference in gridders.

## Deconvolution Output

As seen in the CLEAN images in Figure 10, there is a small point of light located just to the side of the very bright point. This is likely an artifact of too many iterations of the Högbom CLEAN taking place, the brightest point has slowly decreased as the iterations take place and lesser point have began to be added to the CLEAN image.

Therefore, iterations of different sizes should be run, and the outputs compared. This was left in this report to show a potential side effect from the deconvolution process.

## Development of Versions

The time to develop the code was by far the longest part of this research project. At each part pseudocode and a large amount of trial and error occurred until the correct processes were being carried out.

For the first version *UsingComplex,* took a large amount of time to figure out the correct implementation of the algorithms. It was made slightly easier by using the complex data type as the code was much easier to read. This allowed for the versions to be developed faster as it was a good base to work off and make more complex.

*Using2Grids* improved upon the previous version by dropping the data type and storing the values in two arrays. This iteration required changes at every point in code as the operations were completely different. This change was effective though as seen in the improvements in runtime. The code itself becomes slightly harder to read as it is difficult to follow where the values are being accessed from.

The final iteration produced, *UsingMultiThreaded,* took much longer to develop then *Using2Grids.* The introduction of using threads made this version much more complex then the other two. Only 4 threads were used for this version as there was found to be diminishing returns when using more threads.

The versions were a large amount of work to produce, but once images were begin produced and the improvements were effective, the development was very successful.

## Concurrent Updates issue

To improve the performance of the gridder, multiple threads were used to deal with an equal amount of visibilities. Four threads were used, each getting a quarter of the 223,000 visibilities. However, this caused an issue when adding values to the grid. If two threads had visibilities located at nearby points, there is an issue where both threads try to update the value at the same time with neither account for the others value. This issue is covered, and some solutions provided in the book (Arpaci-Dusseau & Arpaci-Dusseau, 2018).

A solution considered to solve this issue was assigning a lock variable to each grid point, to ensure only one thread has access at any one time. This approach was not adopted as there will have been significant overhead with threads waiting on each other, instead a different solution was chosen. Each thread contains its own 2D grids, therefore each visibility is added independently from each other. Once each thread has finished, all 4 grids are combined, giving the final grid.

## Analysis of Runtimes

The runtimes shown in Figure 11 show each version having faster performance then the last. To ensure this was the case three statistical tests were carried out. Paired t-tests were used to test the mean runtime as all versions used the same data. The following code was ran using RStudio.

> t.test(UsingComplex, Using2Grids, paired = TRUE)

Paired t-test

data: UsingComplex and Using2Grids

t = 27.165, df = 19, p-value < 2.2e-16

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

129858744 151539979

sample estimates:

mean of the differences

140699361

> t.test(Using2Grids, UsingMultipleThreads, paired = TRUE)

Paired t-test

data: Using2Grids and UsingMultipleThreads

t = 9.6711, df = 19, p-value = 8.995e-09

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

75635012 117415113

sample estimates:

mean of the differences

96525062

Figure 13. Output from t-tests

For the tests we use a null hypothesis that the means are equal, given the p-values from the tests, the results are statistically significant and so we can say that each pipeline has a faster runtime then the previous version.

For the case of *Using2Grids* over *UsingComplex*, the improvements come from not have to create and destroy the complex data type. Each time a new array is created the pipeline has to cerate 10242 objects to store the values whereas using the double primitive this doesn’t have to occur.

*UsingMultipleThreads* has the advantage of splitting the work load for the grinder and the transform up among multiple threads whereas *UsingComplex* and *Using2Grids* deal with each process iteratively one at a time.

## Gridder Runtime Tests

The tests carried out in Figure 12 show how the runtimes of the different versions compare when given increasing amounts of visibilities.

When given smaller amounts, the gridders all perform similarly. *UsingComplex* and *Using2Grids* have similar times as a small amount of complex object are created and destroyed. *UsingMultipleThreads* has a similar performance as the time taken to created threads and start them is relatively large to the time to process the small amounts.

As the amount of visibilities increase, the spread between the lines increase. The order of performancePip of the gridders is the same order for each version when carrying out the entire image synthesis.

*UsingMultipleThreads* is by far the fastest and the difference only increases with more visibilities. With the work load being split between multiple threads, its runtime doesn’t increase at the same rate as the other two gridders. Also, the time taken to instantiate and start the threads becomes negligible compared to the time of processing the data. The time spent combining the threads after they have completed their load is also negligible as it is just an addition of 4 numbers for each grid point and doesn’t change with more visibilities being added in.

Conclusion

## Synthesis Imaging

The literature review led to knowledge being gathered on the processes involved in developing an imaging pipeline. This knowledge was applied with a Design Science methodology to create and improve code. This process was a success as

## Gridder

This project led to the successful development of software capable of placing visibility data on a grid. Through the images output from the pipeline it is known that the gridder carries out the correct processes. The gridder ended up being the complicated code to implement, however, because of the optimisations it ended up being where the most runtime was saved over the older versions.

## Fourier Transform

The transform implemented was capable of performing an FFT (Fast Fourier Transform).

## Deconvolution

## Performance Improvements

References

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating Systems: Three Easy Pieces.* Arpaci-Dusseau Books, LLC.

Brigham, R. O. (1988). *The Fast Fourier Transform and its Applications.* Prentice-Hall.

Brouw, W. N. (1975). Aperture Synthesis. In C. De Jager, & H. Nieuwenhuijzen, *Image Processing Techinques in Astronomy* (pp. 301-307). Dordrecht: Springer.

Clark, B. G. (1980). An efficient implementation of the algorithm 'CLEAN'. *Astronomy and Astrophysics*, 377-378.

Cooley, J., & Tukey, J. (1965). An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 297-301.

Cornwell, T., & Bridle, A. (1996). *Deconvolution Tutorial*. Retrieved from National Radio: https://www.cv.nrao.edu/~abridle/deconvol/deconvol.html

Högbom, J. (1974). *Astronomy and Astrophysics Supplement Series*, 417.

Hogg, D. E., MacDonald, G. H., Conway, R. G., & Wade, C. M. (1969). Synthesis of Brightness Distribution in Radio Sources. *Astronomical Journal*, 1206-1213.

Oracle. (1993). *Class Double*. Retrieved from Javadocs: https://docs.oracle.com/javase/7/docs/api/java/lang/Double.html

O'Sullivan, J. D. (1985). A Fast Sinc Function Gridding Algorithm for Fourier Inversion in Computer Tomography. *IEEE Transactions on Medical Imaging*, 200-207.

Rau, U. (2012, Sept 24). *Imaging and Deconvolution.* Retrieved from Australia Telescope National Facility: https://www.atnf.csiro.au/research/radio-school/2012/lectures/tue/RVU\_ImagingDeconvolution.pdf

Ryle, M., & Hewish, A. (1960). The synthesis of large radio telescopes. *Monthly Notices of the Royal Astronomical Society, Vol. 120*, 220-230.

Schwab, F. R. (1984). Relaxing the isoplanatism assumption in self-calibration; applications to low-frequency radio interferometry. *Astronomical Journal*, 1076-1081.

Skilling, J., & Bryan, R. K. (1984). Maximum Entropy Image Reconstruction. *Monthly Notices of the Royal Astronomical Society*, 111-124.

The Event Horizon Telescope Collaboration et al. (2018). First M87 Event Horizon Telescope Results. I. The Shadow of the Supermassive Black Hole. *The Astrophysical Journal Letters*, L1.

Appendix

## Appendix A

The full ‘CLEAN’ image. It shows the main bright point found near the center of the image.

A close up of a computer

Description generated with high confidence

## Appendix B

The specifications for the computer used are as follows.

* Processor: Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz, 3501 Mhz, 4 Core(s), 4 Logical Processor(s)
* Installed Physical Memory (RAM): 8.00 GB
* OS: Microsoft Windows 10 Pro
* Version: 10.0.17134 Build 17134
* IDE: Eclipse Java 2018-12

## Appendix C

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | UsingComplex (nanoseconds) | Using2Grids (nanoseconds) | UsingMultipleThreads (nanoseconds) |
|  |  |  |  |
| 1 | 721572276 | 595707985 | 471013685 |
| 2 | 707869718 | 598323862 | 549086228 |
| 3 | 737278067 | 564050352 | 529862565 |
| 4 | 702845398 | 576725692 | 473142139 |
| 5 | 695590810 | 553444657 | 492900328 |
| 6 | 736351787 | 552710596 | 481913999 |
| 7 | 709737198 | 553394920 | 472064892 |
| 8 | 685787637 | 553895508 | 477841998 |
| 9 | 687502981 | 558225850 | 466101420 |
| 10 | 707462460 | 602770354 | 444948252 |
| 11 | 741156094 | 644431864 | 437897000 |
| 12 | 738374038 | 570869009 | 576250265 |
| 13 | 733327775 | 569788252 | 439863662 |
| 14 | 704690643 | 557532749 | 444330050 |
| 15 | 695773960 | 550730184 | 439328257 |
| 16 | 693642581 | 558114673 | 453925497 |
| 17 | 717103548 | 556702141 | 451920508 |
| 18 | 699835136 | 545330788 | 445945041 |
| 19 | 686543933 | 549577455 | 442779424 |
| 20 | 681117621 | 557249541 | 447959976 |
| Mean | 709178183.1 | 568478821.6 | 471953759.3 |
| Standard Deviation | 19570930.35 | 24486395.06 | 38638593.29 |

## Appendix D

The following will be the code produced from the development of the pipeline. Included is the gridder and Fourier Transform for each stage, the code that produced the dirty beam, and the deconvolution using the Complex object. Only one version of the deconvolution is included and very little changes were made between the versions.

/\*\*

\* This class is responsible for calling the correct functions for image synthesis.

\* It begins with the gridding of visibility data, then the grid produced is inverse

\* Fourier transformed, then the image can be passed on for deconvolution.

\*

\* This version uses a 2D array of Complex objects to store the image.

\*

\* **@author** Bobbie Ware

\*/

**package** UsingComplex;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.math.BigDecimal;

**import** java.text.DecimalFormat;

**public** **class** PipelineUsingComplex

{

/\*\*

\* Function to carry out the gridder 10 times.

\* Used to get runtimes for the gridder.

\*/

**private** **static** **void** tenIterations()

{

**long** time = System.nanoTime();

**for** (**int** i = 0; i < 10; i++)

{

Complex[][] grid = GridderUsingComplex.grid();

//Complex[][] transformedGrid = iFFTUsingComplex.twoDimensionifft(grid);

}

System.out.println(System.nanoTime() - time);

}

**public** **static** **void** main(String[] args)

{

tenIterations();

// Carries out gridding

Complex[][] grid = GridderUsingComplex.grid();

// Transforms the grid

Complex[][] transformedGrid = iFFTUsingComplex.twoDimensionifft(grid);

// Used to print the grid to a csv file

DecimalFormat df = **new** DecimalFormat("0.000000");

// try

// {

// PrintWriter pw = new PrintWriter("test3.csv");

//

// for (int i = 0; i < grid.length; i++)

// {

// pw.print(df.format(transformedGrid[i][0].getReal()));

// for (int j = 1; j < grid.length; j++)

// {

// pw.print("," + df.format(transformedGrid[i][j].getReal()));

// }

// pw.print("\n");

// }

// pw.close();

// } catch (FileNotFoundException e)

// {

// e.printStackTrace();

// }

}

}

/\*

\* This class is used to place visibility data onto a grid.

\* The grid for this version is a 2D array of Complex objects.

\*

\* @author Bobbie Ware

\*/

**package** UsingComplex;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.LinkedList;

**public** **class** GridderUsingComplex

{

**private** **final** **static** **int** gridSize = 1024;

**private** **final** **static** **int** heightOfSupport = 7;

**private** **final** **static** **int** widthOfSupport = 7;

// Kernel from data

**private** **final** **static** **double**[] prolateSpheroidal = { 0.013051, 0.0308, 0.061057, 0.107442, 0.172657, 0.257676,

0.361065, 0.478594, 0.603286, 0.725965, 0.836271, 0.923991, 0.98049, 1 };

/\*

\* Loads in the visibilities data from a given csv file.

\*/

**private** **static** LinkedList<**double**[]> loadVisibilities()

{

// Where the data will be stored

LinkedList<**double**[]> visibilities = **new** LinkedList<**double**[]>();

// Amount of visibilities.

**int** visibilitiesCount;

// File name

String csvFile = "FILE\_TO\_LOAD.csv";

String line = "";

String cvsSpilt = ",";

**try**

{

// Creates buffered reader by loading file

BufferedReader bufferedReader = **new** BufferedReader(**new** FileReader(csvFile));

// The first line will the count of visibilities

visibilitiesCount = Integer.parseInt(bufferedReader.readLine());

// Reads in each line of the reader

**while** ((line = bufferedReader.readLine()) != **null**)

{

// Splits the line up by each comma

String[] complexNumberAsString = line.split(cvsSpilt);

// Converts the read String into a double

**double**[] complexNumber = **new** **double**[4];

**for** (**int** i = 0; i < complexNumber.length; i++)

{

complexNumber[i] = Double.valueOf(complexNumberAsString[i]);

}

// Adds the complex number to the data

visibilities.add(complexNumber);

}

bufferedReader.close();

} **catch** (FileNotFoundException e)

{

e.printStackTrace();

} **catch** (IOException e)

{

e.printStackTrace();

}

**return** visibilities;

}

/\*

\* Take the u,v location and using the round function returns its grid location.

\*/

**private** **static** **double**[] UVtoGrid(**double** u, **double** v)

{

**double**[] gridPoint = **new** **double**[2];

// From data

**double** cellSize = Double.parseDouble("4.848136811095360e-06");

**double** UVScale = gridSize \* cellSize;

**double** wavelengthsToMeters = 300000000.0 / 299792458.0; // frequencyHZ / speed of light

gridPoint[0] = ((-u \* wavelengthsToMeters) \* UVScale) + gridSize / 2;

gridPoint[1] = ((v \* wavelengthsToMeters) \* UVScale) + gridSize / 2;

**return** gridPoint;

}

/\*

\* Rounds d to find the correct value with the kernel to use

\*/

**private** **static** **int** inKernel(**double** d)

{

**return** (**int**) ((d / 4) \* 14);

}

/\*

\* Function to carry out all the processes involved in forming the grid

\*/

**public** **static** Complex[][] grid()

{

// This represents the grid where we will be placing the visibilities

Complex[][] grid = **new** Complex[gridSize][gridSize];

// Fills the grid with Complex object as we want to add

**for** (**int** row = 0; row < grid.length; row++)

{

**for** (**int** column = 0; column < grid.length; column++)

{

grid[row][column] = **new** Complex();

}

}

// The following lines forms the 2d prolate spherodial.

**double**[][] gridProlateSpheroidal = **new** **double**[14][14];

**for** (**int** i = 0; i < gridProlateSpheroidal[0].length; i++)

{

**for** (**int** j = 0; j < gridProlateSpheroidal[0].length; j++)

{

gridProlateSpheroidal[gridProlateSpheroidal[0].length - 1 - i][gridProlateSpheroidal[0].length - 1

- j] = prolateSpheroidal[i] \* prolateSpheroidal[j];

}

}

// Visibilities

LinkedList<**double**[]> visibilities = loadVisibilities();

// Each visibility is placed on the grid

**for** (**double**[] visibility : visibilities)

{

// The absolute point on the grid

**double**[] trueGridPoint = UVtoGrid(visibility[0], visibility[1]);

// Rounds the true point to a grid point

**int**[] nearestGridPoint = { (**int**) trueGridPoint[0], (**int**) trueGridPoint[1] };

// The area in which each point is spread

**int** xMin = -(heightOfSupport - 1) / 2;

**int** xMax = (heightOfSupport - 1) / 2;

**int** yMin = -(widthOfSupport - 1) / 2;

**int** yMax = (widthOfSupport - 1) / 2;

// For the 2D support area

**for** (**int** i = xMin; i <= xMax; i++)

{

**for** (**int** j = yMin; j <= yMax; j++)

{

// Finds the corresponding point in the kernel

**double** deltaX = (nearestGridPoint[0] + i) - trueGridPoint[0];

**int** kernelX = inKernel(deltaX);

**double** deltaY = (nearestGridPoint[1] + j) - trueGridPoint[1];

**int** kernelY = inKernel(deltaY);

// Loads the support value

**double** kernelValue = gridProlateSpheroidal[Math.abs(kernelX)][Math.abs(kernelY)];

// Stores the value to be add to the Complex grid

Complex newValue = **new** Complex(visibility[2] \* kernelValue, visibility[3] \* kernelValue);

// In-place addition of the Complex number

grid[nearestGridPoint[0] + i][nearestGridPoint[1] + j].addInPlace(newValue);

}

}

}

**return** grid;

}

}

/\*

\* The Fourier Transform is used to show different parts of a continuous signal.

\* However, for Interferometry an Inverse Fourier Transform is used as we are taking

\* the data from the Fourier Plane and creating an image from it.

\*

\* This implementation is a radix 2 iFFT, it also a 2D Complex object array

\*

\* @author Bobbie Ware

\*/

**package** UsingComplex;

**public** **class** iFFTUsingComplex

{

**public** **static** Complex[][] twoDimensionifft(Complex[][] grid)

{

**int** n = grid.length;

// The grid is shifted to correctly center the grid

Complex[][] transformed = shift(grid);

// transforming along each row

**for** (**int** row = 0; row < n; row++)

{

transformed[row] = ifft(transformed[row]);

}

// transforming down each column

**for** (**int** column = 0; column < n; column++)

{

// create temporary array for the column

Complex[] transformedColumn = **new** Complex[n];

**for** (**int** row = 0; row < n; row++)

{

transformedColumn[row] = transformed[row][column];

}

transformedColumn = ifft(transformedColumn);

**for** (**int** row = 0; row < n; row++)

{

transformed[row][column] = transformedColumn[row];

}

}

// Output is shifted to re-center the grid

**return** shift(transformed);

}

/\*\*

\* The transform for an array of complex numbers

\*/

**public** **static** Complex[] ifft(Complex[] input)

{

**int** n = input.length;

// A bit reversal is carried out on the array

Complex[] reversedArray = bitReverse(input);

// Decimation in time butterfly operations

**for** (**int** m = 2; m <= n; m \*= 2)

{

Complex omega = **new** Complex(Math.cos((-2 \* Math.PI) / m), Math.sin((-2 \* Math.PI) / m));

**for** (**int** k = 0; k < n; k += m)

{

Complex x = **new** Complex(1, 0);

**for** (**int** j = 0; j < m / 2; j++)

{

Complex t = x.multiply(reversedArray[k + j + m / 2]);

Complex u = reversedArray[k + j];

reversedArray[k + j] = u.add(t);

reversedArray[k + j + m / 2] = u.subtract(t);

x.multiplyInPlace(omega);

}

}

}

**return** reversedArray;

}

/\*

\* A bit reversal is used to correct for the rearrangement of data in the transform

\*/

**private** **static** Complex[] bitReverse(Complex[] array)

{

Complex[] reversed = **new** Complex[array.length];

**for** (**int** k = 0; k < array.length; k++)

{

// calculate position of k in new array

**int** kNew = k;

**int** r = 0;

**for** (**int** j = 1; j < array.length; j \*= 2)

{

**int** b = kNew & 1;

r = (r << 1) + b;

kNew = (kNew >>> 1);

}

reversed[r] = array[k];

}

**return** reversed;

}

/\*

\* Carries out a shift on the grid rearrange the order of the points.

\* It places the corners at the center of the image and move the other points around

\* accordingly.

\*/

**public** **static** Complex[][] shift(Complex[][] grid)

{

Complex[][] shiftedGrid = **new** Complex[1024][1024];

**for** (**int** row = 0; row < shiftedGrid.length; row++)

{

**for** (**int** column = 0; column < shiftedGrid.length; column++)

{

**int** xDiff, yDiff;

**if** (row < 512)

{

xDiff = 512;

}

**else**

{

xDiff = -512;

}

**if** (column < 512)

{

yDiff = 512;

}

**else**

{

yDiff = -512;

}

shiftedGrid[row][column] = grid[row + xDiff][column + yDiff];

}

}

**return** shiftedGrid;

}

}

/\*\*

\* This class is responsible for containing a complex number

\* It also has all the necessary operations required for

\* the processes

\*

\* **@author** Bobbie Ware

\*/

**package** UsingComplex;

**public** **class** Complex

{

// Complex number consists of a real number and a imaginary number

**private** **double** real;

**private** **double** imag;

**public** Complex()

{

**this**(0, 0);

}

/\*

\* Creates a new object with the given values

\*/

**public** Complex(**double** real, **double** imag)

{

**this**.real = real;

**this**.imag = imag;

}

/\*

\* Returns a new Complex object of the sum of two objects

\*/

**public** Complex add(Complex other)

{

**return** **new** Complex(**this**.real + other.getReal(), **this**.imag + other.getImaginary());

}

/\*

\* Adds the value of another Complex object to this one

\*/

**public** Complex addInPlace(Complex other)

{

**this**.real += other.getReal();

**this**.imag += other.getImaginary();

**return** **this**;

}

/\*

\* Returns a new Complex object of the difference of two objects

\*/

**public** Complex subtract(Complex other)

{

**return** **new** Complex(**this**.real - other.getReal(), **this**.imag - other.getImaginary());

}

/\*

\* Returns a new Complex object of the product of two of objects

\*/

**public** Complex multiply(Complex other)

{

**return** **new** Complex(**this**.real \* other.getReal() - **this**.imag \* other.getImaginary(), **this**.real \* other.getImaginary() + **this**.imag \* other.getReal());

}

/\*

\* Multiplies the value of another Complex object to this one

\*/

**public** Complex multiplyInPlace(Complex other)

{

**double** newReal = **this**.real \* other.getReal() - **this**.imag \* other.getImaginary();

**double** newImag = **this**.real \* other.getImaginary() + **this**.imag \* other.getReal();

**this**.real = newReal;

**this**.imag = newImag;

**return** **this**;

}

/\*

\* Returns the real part of Complex number

\*/

**public** **double** getReal()

{

**return** **this**.real;

}

/\*

\* Returns the imaginary part of Complex number

\*/

**public** **double** getImaginary()

{

**return** **this**.imag;

}

}

/\*\*

\* This class is responsible for deconvolution.

\*

\* It carries out the processes of a Hogbom 'CLEAN'.

\*

\* The process can be easily adjusted to output the residuals

\* by change the grid returned by the Deconvolution class.

\*

\* **@author** Bobbie Ware

\*/

**package** Deconvolution;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.math.BigDecimal;

**import** java.text.DecimalFormat;

**public** **class** Deconvoluter

{

**public** **static** **void** main(String[] args)

{

// Loads the grid to be deconvoluted

Complex[][] grid = LoadGrid.loadGrid();

// Carries out a

Complex[][] cleanedGrid = Deconvolution.deconvolute(grid, 1000);

// Used to print the grid to a csv file

// DecimalFormat df = new DecimalFormat("0.000000");

//

// try

// {

// PrintWriter pw = new PrintWriter("output\_image.csv");

//

// for (int i = 0; i < cleanedGrid.length; i++)

// {

// pw.print(df.format(cleanedGrid[i][0].getReal()));

// for (int j = 1; j < cleanedGrid.length; j++)

// {

// pw.print("," + df.format(cleanedGrid[i][j].getReal()));

// }

// pw.print("\n");

// }

// pw.close();

// } catch (FileNotFoundException e)

// {

// e.printStackTrace();

// }

}

}

/\*

\* This class is used to load the grid that will be deconvoluted

\* It can be configured by changing the csvFile variable to

\* the image that will be processed.

\*

\* @author Bobbie Ware

\*/

**package** Deconvolution;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** LoadGrid

{

**public** **static** Complex[][] loadGrid()

{

Complex[][] grid = **new** Complex[1024][1024];

**for** (**int** row = 0; row < grid.length; row++)

{

**for** (**int** column = 0; column < grid.length; column++)

{

grid[row][column] = **new** Complex();

}

}

// File name

String csvFile = "input-real.csv";

String csvFile2 = "input-imaginary.csv";

String line = "";

String line2 = "";

String cvsSpilt = ",";

**int** lineNum = 0;

**try**

{

// Creates buffered reader by loading file

BufferedReader bufferedReader = **new** BufferedReader(**new** FileReader(csvFile));

BufferedReader bufferedReader2 = **new** BufferedReader(**new** FileReader(csvFile2));

// Reads in each line of the reader

**while** ((line = bufferedReader.readLine()) != **null**)

{

line2 = bufferedReader2.readLine();

// Splits the line up by each comma

String[] complexNumberAsString = line.split(cvsSpilt);

String[] complexNumberAsString2 = line2.split(cvsSpilt);

**for** (**int** i = 0; i < 1024; i++)

{

grid[lineNum][i] = **new** Complex(Double.parseDouble(complexNumberAsString[i]), Double.parseDouble(complexNumberAsString2[i]));

}

lineNum++;

}

bufferedReader.close();

bufferedReader2.close();

} **catch** (FileNotFoundException e)

{

e.printStackTrace();

} **catch** (IOException e)

{

e.printStackTrace();

}

**return** grid;

}

}

/\*

\* This class is used to load the beam that will be used for

\* the deconvolution.

\* It can be configured by changing the csvFile variable to

\* the image that will be processed.

\*

\* @author Bobbie Ware

\*/

**package** Deconvolution;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** LoadBeam

{

**public** **static** Complex[][] loadBeam()

{

Complex[][] grid = **new** Complex[5][5];

**for** (**int** row = 0; row < grid.length; row++)

{

**for** (**int** column = 0; column < grid.length; column++)

{

grid[row][column] = **new** Complex();

}

}

// File name

String csvFile = "DirtyBeam.csv";

String line = "";

String cvsSpilt = ",";

**int** lineNum = 0;

**try**

{

// Creates buffered reader by loading file

BufferedReader bufferedReader = **new** BufferedReader(**new** FileReader(csvFile));

// Reads in each line of the reader

**while** ((line = bufferedReader.readLine()) != **null**)

{

// Splits the line up by each comma

String[] complexNumberAsString = line.split(cvsSpilt);

**for** (**int** i = 0; i < 5; i++)

{

**if** (Double.parseDouble(complexNumberAsString[i]) > 0)

{

grid[lineNum][i] = **new** Complex(Double.parseDouble(complexNumberAsString[i]), 0);

}

}

lineNum++;

}

bufferedReader.close();

} **catch** (FileNotFoundException e)

{

e.printStackTrace();

} **catch** (IOException e)

{

e.printStackTrace();

}

**return** grid;

}

}

/\*\*

\* This class is responsible for carrying out the CLEAN algorithm

\* A factor of 0.1 is used as the fraction to be taken away from

\* the dirty image.

\*

\* **@author** Bobbie Ware

\*/

**package** Deconvolution;

**public** **class** Deconvolution

{

/\*

\* Carries out the process of deconvolution for a given amount of iterations

\*/

**public** **static** Complex[][] deconvolute(Complex[][] grid, **int** iterations)

{

// Location of brightest pixel

**int**[] brightestPixel;

Complex[][] cleaned = **new** Complex[1024][1024];

// Loads in the dirty beam of psf (point spread function)

Complex[][] beam = LoadBeam.loadBeam();

**for** (**int** row = 0; row < grid.length; row++)

{

**for** (**int** column = 0; column < grid.length; column++)

{

cleaned[row][column] = **new** Complex();

}

}

// Carried out i times

**for** (**int** i = 0; i < iterations; i++)

{

brightestPixel = findBrightest(grid);

**int** xMin = -2;

**int** xMax = 2;

**int** yMin = -2;

**int** yMax = 2;

// Using the shape of the dirty beam, 0.1 of the beam is taken

// away from the image and placed onto the 'CLEAN' image

**for** (**int** row = xMin; row <= xMax; row++)

{

**for** (**int** column = yMin; column <= yMax; column++)

{

**int** xCoordinate = (brightestPixel[0] + row);

**int** yCoordinate = (brightestPixel[1] + column);

**double** factor = 0.1;

**double** ammount = beam[2 + row][2 + column].getReal() \* factor;

// Adds the point to the cleaned grid

Complex newValue = **new** Complex(ammount, 0);

cleaned[xCoordinate][yCoordinate].addInPlace(newValue);

grid[xCoordinate][yCoordinate].subtractInPlace(newValue);

}

}

}

// this value can be changed to grid to return the

// residuals from the 'CLEAN'

**return** cleaned;

}

/\*

\* Used to find the strongest signal in the image

\*/

**private** **static** **int**[] findBrightest(Complex[][] grid)

{

**int** xLocation = 0;

**int** yLcoation = 0;

**for** (**int** row = 0; row < grid.length; row++)

{

**for** (**int** column = 0; column < grid.length; column++)

{

**if** (grid[row][column].getReal() > grid[xLocation][yLcoation].getReal())

{

xLocation = row;

yLcoation = column;

}

}

}

**return** **new** **int**[] {xLocation, yLcoation};

}

}

/\*\*

\* This class is responsible for calling the correct functions for image synthesis.

\* It begins with the gridding of visibility data, then the grid produced is inverse

\* Fourier transformed, then the image can be passed on for deconvolution

\*

\* This version uses two 2D double arrays to store the image.

\*

\* **@author** Bobbie Ware

\*/

**package** Using2grids;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.text.DecimalFormat;

**public** **class** Pipeline2Grids

{

/\*\*

\* Function to carry out the gridder 10 times.

\* Used to get runtimes for the gridder.

\*/

**private** **static** **void** tenIterations()

{

**long** time = System.nanoTime();

**for** (**int** i = 0; i < 10; i++)

{

**double**[][][] grid = Gridder2Grids.grid();

}

System.out.println(System.nanoTime() - time);

}

**public** **static** **void** main(String[] args)

{

//tenIterations();

// Carries out gridding

**double**[][][] grid = Gridder2Grids.grid();

// Transforms the grid

**double**[][][] transformedGrid = iFFT2Grids.twoDimensionifft(grid[0], grid[1]);

**double**[][] real = transformedGrid[0];

// Used to print the grid to a csv file

// DecimalFormat df = new DecimalFormat("0.000000");

//

// try

// {

// PrintWriter pw = new PrintWriter("test3.csv");

//

// for (int i = 0; i < real.length; i++)

// {

// pw.print(df.format(real[i][0]));

// for (int j = 1; j < real.length; j++)

// {

// pw.print("," + df.format(real[i][j]));

// }

// pw.print("\n");

// }

// pw.close();

// } catch (FileNotFoundException e)

// {

// e.printStackTrace();

// }

}

}

/\*

\* This class is used to place visibility data onto a grid.

\* The grid for this version is two 2D arrays of doubles.

\*

\* @author Bobbie Ware

\*/

**package** Using2grids;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**import** java.util.LinkedList;

**public** **class** Gridder2Grids

{

**private** **final** **static** **int** gridSize = 1024;

**private** **final** **static** **int** heightOfSupport = 7;

**private** **final** **static** **int** widthOfSupport = 7;

// Kernel from data

**private** **final** **static** **double**[] prolateSpheroidal = { 0.013051, 0.0308, 0.061057, 0.107442, 0.172657, 0.257676,

0.361065, 0.478594, 0.603286, 0.725965, 0.836271, 0.923991, 0.98049, 1 };

/\*

\* Loads in the visibilities data from a given csv file

\*/

**private** **static** LinkedList<**double**[]> loadVisibilities()

{

// Where the data will be stored

LinkedList<**double**[]> visibilities = **new** LinkedList<**double**[]>();

// Amount of visibilities

**int** visibilitiesCount;

// File name

String csvFile = "FILE\_TO\_LOAD.csv";

String line = "";

String cvsSpilt = ",";

**try**

{

// Creates buffered reader by loading file

BufferedReader bufferedReader = **new** BufferedReader(**new** FileReader(csvFile));

// The first line will the count of visibilities

visibilitiesCount = Integer.parseInt(bufferedReader.readLine());

// Reads in each line of the reader

**while** ((line = bufferedReader.readLine()) != **null**)

{

// Splits the line up by each comma

String[] complexNumberAsString = line.split(cvsSpilt);

// Converts the read String into a double

**double**[] complexNumber = **new** **double**[4];

**for** (**int** i = 0; i < complexNumber.length; i++)

{

complexNumber[i] = Double.valueOf(complexNumberAsString[i]);

}

// Adds the complex number to the data

visibilities.add(complexNumber);

}

bufferedReader.close();

} **catch** (FileNotFoundException e)

{

e.printStackTrace();

} **catch** (IOException e)

{

e.printStackTrace();

}

**return** visibilities;

}

/\*

\* Take the u,v location and using the round function returns its grid location.

\*/

**private** **static** **double**[] UVtoGrid(**double** u, **double** v)

{

**double**[] gridPoint = **new** **double**[2];

// From data

**double** cellSize = 0.00000484813681109536;

**double** UVScale = gridSize \* cellSize;

**double** wavelengthsToMeters = 300000000.0 / 299792458.0; // frequencyHZ / speed of light

gridPoint[0] = ((-u \* wavelengthsToMeters) \* UVScale) + gridSize / 2;

gridPoint[1] = ((v \* wavelengthsToMeters) \* UVScale) + gridSize / 2;

**return** gridPoint;

}

/\*

\* Rounds d to find the correct value with the kernel to use

\*/

**private** **static** **int** inKernel(**double** d)

{

**return** (**int**) ((d / 4) \* 14);

}

/\*

\* Function to carry out all the processes involved in forming the grid

\*/

**public** **static** **double**[][][] grid()

{

// This represents the grid where we will be placing the visibilities

**double**[][] realGrid = **new** **double**[gridSize][gridSize];

**double**[][] imaginaryGrid = **new** **double**[gridSize][gridSize];

// The following lines forms the 2d prolate spherodial.

**double**[][] gridProlateSpheroidal = **new** **double**[14][14];

**for** (**int** i = 0; i < gridProlateSpheroidal[0].length; i++)

{

**for** (**int** j = 0; j < gridProlateSpheroidal[0].length; j++)

{

gridProlateSpheroidal[gridProlateSpheroidal[0].length - 1 - i][gridProlateSpheroidal[0].length - 1

- j] = prolateSpheroidal[i] \* prolateSpheroidal[j];

}

}

// Visibilities

LinkedList<**double**[]> visibilities = loadVisibilities();

// Each visibility is placed on the grid

**for** (**double**[] visibility : visibilities)

{

// The absolute point on the grid

**double**[] trueGridPoint = UVtoGrid(visibility[0], visibility[1]);

// Rounds the true point to a grid point

**int**[] nearestGridPoint = { (**int**) trueGridPoint[0], (**int**) trueGridPoint[1] };

// The area in which each point is spread

**int** xMin = -(heightOfSupport - 1) / 2;

**int** xMax = (heightOfSupport - 1) / 2;

**int** yMin = -(widthOfSupport - 1) / 2;

**int** yMax = (widthOfSupport - 1) / 2;

// For the 2D support area

**for** (**int** i = xMin; i <= xMax; i++)

{

**for** (**int** j = yMin; j <= yMax; j++)

{

// Finds the corresponding point in the kernel

**double** deltaX = (nearestGridPoint[0] + i) - trueGridPoint[0];

**int** kernelX = inKernel(deltaX);

**double** deltaY = (nearestGridPoint[1] + j) - trueGridPoint[1];

**int** kernelY = inKernel(deltaY);

// Loads the support value

**double** kernelValue = gridProlateSpheroidal[Math.abs(kernelX)][Math.abs(kernelY)];

// Adds the value in the real grid, the value of the visibility is times by the kernel

// value, the same goes for the imaginary part

realGrid[nearestGridPoint[0] + i][nearestGridPoint[1] + j] += visibility[2] \* kernelValue;

imaginaryGrid[nearestGridPoint[0] + i][nearestGridPoint[1] + j] += visibility[3] \* kernelValue;

}

}

}

**return** **new** **double**[][][] {realGrid, imaginaryGrid};

}

}

/\*

\* The Fourier Transform is used to show different parts of a continuous signal.

\* However, for Interferometry an Inverse Fourier Transform is used as we are taking

\* the data from the Fourier Plane and creating an image from it.

\*

\* This implementation is a radix 2 iFFT, it also uses two 2D double arrays

\*

\* @author Bobbie Ware

\*/

**package** Using2grids;

**public** **class** iFFT2Grids

{

/\*

\* Carries out a 2D transform on a grid

\*/

**public** **static** **double**[][][] twoDimensionifft(**double**[][] inputReal, **double**[][] inputImag)

{

**int** n = inputReal.length;

// The grid is shifted to correctly center the grid

**double**[][][] shiftedGrid = shift(inputReal, inputImag);

**double**[][] transformedReal = **new** **double**[n][n];

**double**[][] transformedImag = **new** **double**[n][n];

// transforming along each row

**for** (**int** row = 0; row < n; row++)

{

**double**[][] transformed = ifft(shiftedGrid[0][row], shiftedGrid[1][row]);

transformedReal[row] = transformed[0];

transformedImag[row] = transformed[1];

}

// transforming down each column

**for** (**int** column = 0; column < n; column++)

{

// create temporary array for the column

**double**[] realColumn = **new** **double**[n];

**double**[] imagColumn = **new** **double**[n];

**for** (**int** row = 0; row < n; row++)

{

realColumn[row] = transformedReal[row][column];

imagColumn[row] = transformedImag[row][column];

}

**double**[][] transformed = ifft(realColumn, imagColumn);

realColumn = transformed[0];

imagColumn = transformed[1];

**for** (**int** row = 0; row < n; row++)

{

transformedReal[row][column] = realColumn[row];

transformedImag[row][column] = imagColumn[row];

}

}

// Output is shifted to re-center the grid

**return** shift(transformedReal, transformedImag);

}

/\*\*

\* The transform for an array of complex numbers

\*/

**public** **static** **double**[][] ifft(**double**[] inputReal, **double**[] inputImag)

{

**int** n = inputReal.length;

// A bit reversal is carried out on the arrays

**double**[] reversedReal = bitReverse(inputReal);

**double**[] reversedImag = bitReverse(inputImag);

// Decimation in time butterfly operations

**for** (**int** m = 2; m <= n; m \*= 2)

{

**double** omegaReal = Math.cos((2 \* Math.PI) / m);

**double** omegaImag = Math.sin((2 \* Math.PI) / m);

**for** (**int** k = 0; k < n; k += m)

{

**double** xReal = 1;

**double** xImag = 0;

**for** (**int** j = 0; j < m / 2; j++)

{

**double** tReal = (xReal \* reversedReal[k + j + m / 2]) - (xImag \* reversedImag[k + j + m / 2]);

**double** tImag = (xReal \* reversedImag[k + j + m / 2]) + (xImag \* reversedReal[k + j + m / 2]);

**double** uReal = reversedReal[k + j];

**double** uImag = reversedImag[k + j];

reversedReal[k + j] = uReal + tReal;

reversedImag[k + j] = uImag + tImag;

reversedReal[k + j + m / 2] = uReal - tReal;

reversedImag[k + j + m / 2] = uImag - tImag;

**double** newXR = (xReal \* omegaReal) - (xImag \* omegaImag);

**double** newXI = (xReal \* omegaImag) + (xImag \* omegaReal);

xReal = newXR;

xImag = newXI;

}

}

}

**return** **new** **double**[][]{ reversedReal, reversedImag };

}

/\*

\* A bit reversal is used to correct for the rearrangement of data in the transform

\*/

**private** **static** **double**[] bitReverse(**double**[] array)

{

**double**[] reversed = **new** **double**[array.length];

**for** (**int** k = 0; k < array.length; k++)

{

// calculate position of k in new array

**int** kNew = k;

**int** r = 0;

**for** (**int** j = 1; j < array.length; j \*= 2)

{

**int** b = kNew & 1;

r = (r << 1) + b;

kNew = (kNew >>> 1);

}

reversed[r] = array[k];

}

**return** reversed;

}

/\*

\* Carries out a shift on the grid rearrange the order of the points.

\* It places the corners at the center of the image and move the other points around

\* accordingly.

\*/

**public** **static** **double**[][][] shift(**double**[][] inputReal, **double**[][] inputImag)

{

**double**[][] realShifted = **new** **double**[1024][1024];

**double**[][] imagShifted = **new** **double**[1024][1024];

**for** (**int** row = 0; row < realShifted.length; row++)

{

**for** (**int** column = 0; column < realShifted.length; column++)

{

**int** xDiff, yDiff;

**if** (row < 512)

{

xDiff = 512;

}

**else**

{

xDiff = -512;

}

**if** (column < 512)

{

yDiff = 512;

}

**else**

{

yDiff = -512;

}

realShifted[row][column] = inputReal[row + xDiff][column + yDiff];

imagShifted[row][column] = inputImag[row + xDiff][column + yDiff];

}

}

**return** **new** **double**[][][]{ realShifted, imagShifted };

}

}

/\*\*

\* This class is responsible for calling the correct functions for image synthesis.

\* It begins with the gridding of visibility data, then the grid produced is inverse

\* Fourier transformed, then the image can be passed on for deconvolution.

\*

\* This version using multi-threading to improve performance of the gridder and

\* the Fourier Transform.

\*

\* **@author** Bobbie Ware

\*/

**package** UsingMultipleThreads;

**import** java.io.FileNotFoundException;

**import** java.io.PrintWriter;

**import** java.text.DecimalFormat;

**public** **class** PipelineMultiThreaded

{

/\*\*

\* Function to carry out the gridder 10 times.

\* Used to get runtimes for the gridder.

\*/

**private** **static** **void** tenIterations()

{

**long** time = System.nanoTime();

**for** (**int** i = 0; i < 10; i++)

{

**double**[][][] grid = GridderMultiThreaded.grid();

//double[][][] transformedGrid = iFFT2GridsMultiThreaded.twoDimensionifft(grid[0], grid[1]);

}

System.out.println(System.nanoTime() - time);

}

**public** **static** **void** main(String[] args)

{

//tenIterations();

// Carries out gridding

**double**[][][] grid = GridderMultiThreaded.grid();

// Transforms the grid

**double**[][][] transformedGrid = iFFTMultiThreaded.twoDimensionifft(grid[0], grid[1]);

// Applies the convolution correction to the image

**double**[][] real = ConvolutionCorrection.correct(transformedGrid[0]);

// Used to print the grid to a csv file

// DecimalFormat df = new DecimalFormat("0.000000");

//

// try

// {

// PrintWriter pw = new PrintWriter("output\_image.csv");

//

// for (int i = 0; i < real.length; i++)

// {

// pw.print(df.format(real[i][0]));

// for (int j = 1; j < real.length; j++)

// {

// pw.print("," + df.format(real[i][j]));

// }

// pw.print("\n");

// }

// pw.close();

// } catch (FileNotFoundException e)

// {

// // **TODO** Auto-generated catch block

// e.printStackTrace();

// }

}

}

/\*

\* This class is used to place visibility data onto a grid.

\* This version uses multi-threading to improve its performance

\*

\* @author Bobbie Ware

\*/

**package** UsingMultipleThreads;

**import** java.io.BufferedReader;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.IOException;

**import** java.util.ArrayList;

**public** **class** GridderMultiThreaded

{

**private** **final** **static** **int** gridSize = 1024;

**private** **static** **int** visibilitiesCount;

**double**[][][] realGrid;

**double**[][][] imaginaryGrid;

// Kernel from data

**private** **final** **static** **double**[] prolateSpheroidal = { 0.013051, 0.0308, 0.061057, 0.107442, 0.172657, 0.257676,

0.361065, 0.478594, 0.603286, 0.725965, 0.836271, 0.923991, 0.98049, 1 };

/\*

\* Loads in the visibilities data from a given csv file.

\*/

**private** **static** ArrayList<**double**[]> loadVisibilities()

{

// Where the data will be stored

ArrayList<**double**[]> visibilities = **new** ArrayList<**double**[]>();

// File name

String csvFile = "Visibilities.csv";

String line = "";

String cvsSpilt = ",";

**try**

{

// Creates buffered reader by loading file

BufferedReader bufferedReader = **new** BufferedReader(**new** FileReader(csvFile));

// The first line will the count of visibilities

visibilitiesCount = Integer.parseInt(bufferedReader.readLine());

// Reads in each line of the reader

**while** ((line = bufferedReader.readLine()) != **null**)

{

// Splits the line up by each comma

String[] complexNumberAsString = line.split(cvsSpilt);

// Converts the read String into a double

**double**[] complexNumber = **new** **double**[4];

**for** (**int** i = 0; i < complexNumber.length; i++)

{

complexNumber[i] = Double.valueOf(complexNumberAsString[i]);

}

// Adds the complex number to the data

visibilities.add(complexNumber);

}

bufferedReader.close();

} **catch** (FileNotFoundException e)

{

e.printStackTrace();

} **catch** (IOException e)

{

e.printStackTrace();

}

**return** visibilities;

}

/\*

\* Function to carry out all the processes involved in forming the grid.

\* Uses 4 threads to each place a quarter of the visibilities.

\*/

**public** **static** **double**[][][] grid()

{

// This represents the grid where we will be placing the visibilities

**double**[][][] realGrid = **new** **double**[4][gridSize][gridSize];

**double**[][][] imaginaryGrid = **new** **double**[4][gridSize][gridSize];

// The following lines forms the 2d prolate spherodial.

**double**[][] gridProlateSpheroidal = **new** **double**[14][14];

**for** (**int** i = 0; i < gridProlateSpheroidal[0].length; i++)

{

**for** (**int** j = 0; j < gridProlateSpheroidal[0].length; j++)

{

gridProlateSpheroidal[gridProlateSpheroidal[0].length - 1 - i][gridProlateSpheroidal[0].length - 1

- j] = prolateSpheroidal[i] \* prolateSpheroidal[j];

}

}

// Visibilities

ArrayList<**double**[]> visibilities = loadVisibilities();

// Creates the 4 threads using the GriddingThread class

Thread[] threads = **new** Thread[4];

GriddingThread[] griddingThreads = **new** GriddingThread[4];

**int** quarterOfCount = visibilitiesCount / 4;

// Instantiates the threads and gives them their start and end indexes, the visibility data and the

// Prolate Spheroidal for the kernel values.

**for** (**int** i = 0; i < threads.length; i++)

{

**int** start = quarterOfCount \* i;

**int** end = quarterOfCount \* (i + 1);

griddingThreads[i] = **new** GriddingThread(start, end, visibilities, gridProlateSpheroidal);

threads[i] = **new** Thread(griddingThreads[i]);

threads[i].start();

}

// Waits for each thread to have finished placing the visibilities

**for** (**int** i = 0; i < threads.length; i++)

{

**try**

{

threads[i].join();

} **catch** (InterruptedException e)

{

e.printStackTrace();

}

}

// Combines the 4 grid from each thread into one grid

**for** (**int** i = 0; i < griddingThreads.length; i++)

{

realGrid[i] = griddingThreads[i].realGrid;

imaginaryGrid[i] = griddingThreads[i].imaginaryGrid;

}

**for** (**int** row = 0; row < gridSize; row++)

{

**for** (**int** column = 0; column < griddingThreads.length; column++)

{

**for** (**int** i = 0; i < 4; i++)

{

realGrid[0][row][column] += realGrid[i][row][column];

imaginaryGrid[0][row][column] += imaginaryGrid[i][row][column];

}

}

}

**return** **new** **double**[][][]{realGrid[0], imaginaryGrid[0]};

}

**public** **void** setRealGrid(**double**[][] realGrid, **int** i)

{

**this**.realGrid[i] = realGrid;

}

**public** **void** setImaginaryGrid(**double**[][] imaginaryGrid, **int** i)

{

**this**.imaginaryGrid[i] = imaginaryGrid;

}

}

/\*

\* The Fourier Transform is used to show different parts of a continuous signal.

\* However, for Interferometry an Inverse Fourier Transform is used as we are taking

\* the data from the Fourier Plane and creating an image from it.

\*

\* This implementation is a radix 2 iFFT, it also uses two 2D double arrays and uses

\* multiple threads to carry out the transform.

\*

\* @author Bobbie Ware

\*/

**package** UsingMultipleThreads;

**public** **class** iFFTMultiThreaded

{

/\*

\* Carries out a 2D transform on a grid

\*/

**public** **static** **double**[][][] twoDimensionifft(**double**[][] inputReal, **double**[][] inputImag)

{

**int** n = inputReal.length;

// The grid is shifted to correctly center the grid

**double**[][][] shiftedGrid = shift(inputReal, inputImag);

**double**[][] transformedReal = **new** **double**[n][n];

**double**[][] transformedImag = **new** **double**[n][n];

// Creates the 4 threads using the GriddingThread class

Thread[] threads = **new** Thread[4];

iFFTThreadRow[] griddingThreads = **new** iFFTThreadRow[4];

**int** quarterOfCount = 1024 / 4;

// Instantiates the threads and gives them their start and end indexes, the grids

**for** (**int** i = 0; i < threads.length; i++)

{

**int** start = quarterOfCount \* i;

**int** end = quarterOfCount \* (i + 1);

griddingThreads[i] = **new** iFFTThreadRow(start, end, shiftedGrid[0], shiftedGrid[1]);

threads[i] = **new** Thread(griddingThreads[i]);

threads[i].start();

}

// Waits for each thread to have finished transforming the grid

**for** (**int** i = 0; i < threads.length; i++)

{

**try**

{

threads[i].join();

} **catch** (InterruptedException e)

{

e.printStackTrace();

}

}

// Combines the 4 parts from each thread into one grid

**for** (**int** i = 0; i < griddingThreads.length; i++)

{

**for** (**int** j = (quarterOfCount \* i); j < quarterOfCount \* (i + 1); j++)

{

transformedReal[j] = griddingThreads[i].transformedReal[j];

transformedImag[j] = griddingThreads[i].transformedImag[j];

}

}

// The same process is applied except using values from each column instead

// of rows.

Thread[] threads2 = **new** Thread[4];

iFFTThreadColumn[] griddingThreads2 = **new** iFFTThreadColumn[4];

**for** (**int** i = 0; i < threads2.length; i++)

{

**int** start = quarterOfCount \* i;

**int** end = quarterOfCount \* (i + 1);

griddingThreads2[i] = **new** iFFTThreadColumn(start, end, transformedReal, transformedImag);

threads2[i] = **new** Thread(griddingThreads2[i]);

threads2[i].start();

}

**for** (**int** i = 0; i < threads2.length; i++)

{

**try**

{

threads2[i].join();

} **catch** (InterruptedException e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for** (**int** i = 0; i < threads2.length; i++)

{

**try**

{

threads2[i].join();

} **catch** (InterruptedException e)

{

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

**for** (**int** i = 0; i < griddingThreads2.length; i++)

{

**for** (**int** j = (quarterOfCount \* i); j < quarterOfCount \* (i + 1); j++)

{

**for** (**int** j2 = 0; j2 < transformedReal.length; j2++)

{

transformedReal[j2][j] = griddingThreads2[i].transformedReal[j2][j];

transformedImag[j2][j] = griddingThreads2[i].transformedImag[j2][j];

}

}

}

// Output is shifted to re-center the grid

**double**[][][] shiftedGrids = shift(transformedReal, transformedImag);

**return** shiftedGrids;

}

/\*

\* Carries out a shift on the grid rearrange the order of the points.

\* It places the corners at the center of the image and move the other points around

\* accordingly.

\*/

**public** **static** **double**[][][] shift(**double**[][] inputReal, **double**[][] inputImag)

{

**double**[][] realShifted = **new** **double**[1024][1024];

**double**[][] imagShifted = **new** **double**[1024][1024];

**for** (**int** row = 0; row < realShifted.length; row++)

{

**for** (**int** column = 0; column < realShifted.length; column++)

{

**int** xDiff, yDiff;

**if** (row < 512)

{

xDiff = 512;

}

**else**

{

xDiff = -512;

}

**if** (column < 512)

{

yDiff = 512;

}

**else**

{

yDiff = -512;

}

realShifted[row][column] = inputReal[row + xDiff][column + yDiff];

imagShifted[row][column] = inputImag[row + xDiff][column + yDiff];

}

}

**return** **new** **double**[][][]{ realShifted, imagShifted };

}

}

/\*

\* This class a Runnable thread for dealing with a given amount of visibilities and

\* place them onto a grid

\*

\* @author Bobbie Ware

\*/

**package** UsingMultipleThreads;

**public** **class** iFFTThreadRow **implements** Runnable

{

**double**[][] transformedReal;

**double**[][] transformedImag;

**double**[][] inputReal;

**double**[][] inputImag;

**private** **int** start;

**private** **int** end;

/\*\*

\* The transform for an array of complex numbers

\*/

**public** **static** **double**[][] ifft(**double**[] inputReal, **double**[] inputImag)

{

**int** n = inputReal.length;

// A bit reversal is carried out on the arrays

**double**[] reversedReal = bitReverse(inputReal);

**double**[] reversedImag = bitReverse(inputImag);

// Decimation in time butterfly operations

**for** (**int** m = 2; m <= n; m \*= 2)

{

**double** omegaReal = Math.cos((-2 \* Math.PI) / m);

**double** omegaImag = Math.sin((-2 \* Math.PI) / m);

**for** (**int** k = 0; k < n; k += m)

{

**double** xReal = 1;

**double** xImag = 0;

**for** (**int** j = 0; j < m / 2; j++)

{

**double** tReal = (xReal \* reversedReal[k + j + m / 2]) - (xImag \* reversedImag[k + j + m / 2]);

**double** tImag = (xReal \* reversedImag[k + j + m / 2]) + (xImag \* reversedReal[k + j + m / 2]);

**double** uReal = reversedReal[k + j];

**double** uImag = reversedImag[k + j];

reversedReal[k + j] = uReal + tReal;

reversedImag[k + j] = uImag + tImag;

reversedReal[k + j + m / 2] = uReal - tReal;

reversedImag[k + j + m / 2] = uImag - tImag;

**double** newXR = (xReal \* omegaReal) - (xImag \* omegaImag);

**double** newXI = (xReal \* omegaImag) + (xImag \* omegaReal);

xReal = newXR;

xImag = newXI;

}

}

}

**return** **new** **double**[][]{ reversedReal, reversedImag };

}

/\*

\* A bit reversal is used to correct for the rearrangement of data in the transform

\*/

**private** **static** **double**[] bitReverse(**double**[] array)

{

**double**[] reversed = **new** **double**[array.length];

**for** (**int** k = 0; k < array.length; k++)

{

// calculate position of k in new array

**int** kNew = k;

**int** r = 0;

**for** (**int** j = 1; j < array.length; j \*= 2)

{

**int** b = kNew & 1;

r = (r << 1) + b;

kNew = (kNew >>> 1);

}

reversed[r] = array[k];

}

**return** reversed;

}

**public** **double**[][] getTransformedReal()

{

**return** inputReal;

}

**public** **double**[][] getTransformedImag()

{

**return** inputImag;

}

/\*

\* Sets the values for the given thread

\*/

**public** iFFTThreadRow(**int** start, **int** end, **double**[][] inputReal, **double**[][] inputImag)

{

**this**.inputReal = inputReal;

**this**.inputImag = inputImag;

**this**.transformedReal = **new** **double**[1024][1024];

**this**.transformedImag = **new** **double**[1024][1024];

**this**.start = start;

**this**.end = end;

}

/\*

\* This function is ran when the thread is started. It carries out the normal tranform operations

\* but only on a given amount of rows.

\*/

@Override

**public** **void** run()

{

**for** (**int** row = start; row < end; row++)

{

**double**[][] transformed = ifft(inputReal[row], inputImag[row]);

transformedReal[row] = transformed[0];

transformedImag[row] = transformed[1];

}

}

}

/\*

\* This class a Runnable thread for dealing with a given amount of visibilities and

\* place them onto a grid

\*

\* @author Bobbie Ware

\*/

**package** UsingMultipleThreads;

**public** **class** iFFTThreadColumn **implements** Runnable

{

**double**[][] transformedReal;

**double**[][] transformedImag;

**double**[][] inputReal;

**double**[][] inputImag;

**private** **int** start;

**private** **int** end;

**private** **final** **static** **int** n = 1024;

/\*\*

\* The transform for an array of complex numbers

\*/

**public** **static** **double**[][] ifft(**double**[] inputReal, **double**[] inputImag)

{

**int** n = inputReal.length;

// A bit reversal is carried out on the arrays

**double**[] reversedReal = bitReverse(inputReal);

**double**[] reversedImag = bitReverse(inputImag);

// Decimation in time butterfly operations

**for** (**int** m = 2; m <= n; m \*= 2)

{

**double** omegaReal = Math.cos((-2 \* Math.PI) / m);

**double** omegaImag = Math.sin((-2 \* Math.PI) / m);

**for** (**int** k = 0; k < n; k += m)

{

**double** xReal = 1;

**double** xImag = 0;

**for** (**int** j = 0; j < m / 2; j++)

{

**double** tReal = (xReal \* reversedReal[k + j + m / 2]) - (xImag \* reversedImag[k + j + m / 2]);

**double** tImag = (xReal \* reversedImag[k + j + m / 2]) + (xImag \* reversedReal[k + j + m / 2]);

**double** uReal = reversedReal[k + j];

**double** uImag = reversedImag[k + j];

reversedReal[k + j] = uReal + tReal;

reversedImag[k + j] = uImag + tImag;

reversedReal[k + j + m / 2] = uReal - tReal;

reversedImag[k + j + m / 2] = uImag - tImag;

**double** newXR = (xReal \* omegaReal) - (xImag \* omegaImag);

**double** newXI = (xReal \* omegaImag) + (xImag \* omegaReal);

xReal = newXR;

xImag = newXI;

}

}

}

**return** **new** **double**[][]{ reversedReal, reversedImag };

}

/\*

\* A bit reversal is used to correct for the rearrangement of data in the transform

\*/

**private** **static** **double**[] bitReverse(**double**[] array)

{

**double**[] reversed = **new** **double**[array.length];

**for** (**int** k = 0; k < array.length; k++)

{

// calculate position of k in new array

**int** kNew = k;

**int** r = 0;

**for** (**int** j = 1; j < array.length; j \*= 2)

{

**int** b = kNew & 1;

r = (r << 1) + b;

kNew = (kNew >>> 1);

}

reversed[r] = array[k];

}

**return** reversed;

}

**public** **double**[][] getTransformedReal()

{

**return** inputReal;

}

**public** **double**[][] getTransformedImag()

{

**return** inputImag;

}

/\*

\* Sets the values for the given thread

\*/

**public** iFFTThreadColumn(**int** start, **int** end, **double**[][] inputReal, **double**[][] inputImag)

{

**this**.inputReal = inputReal;

**this**.inputImag = inputImag;

**this**.transformedReal = **new** **double**[1024][1024];

**this**.transformedImag = **new** **double**[1024][1024];

**this**.start = start;

**this**.end = end;

}

/\*

\* This function is ran when the thread is started. It carries out the normal tranform operations

\* but only on a given amount of columns.

\*/

@Override

**public** **void** run()

{

**for** (**int** column = start; column < end; column++)

{

// create temporary array for the column

**double**[] realColumn = **new** **double**[n];

**double**[] imagColumn = **new** **double**[n];

**for** (**int** row = 0; row < n; row++)

{

realColumn[row] = inputReal[row][column];

imagColumn[row] = inputImag[row][column];

}

**double**[][] transformed = ifft(realColumn, imagColumn);

realColumn = transformed[0];

imagColumn = transformed[1];

**for** (**int** row = 0; row < n; row++)

{

transformedReal[row][column] = realColumn[row];

transformedImag[row][column] = imagColumn[row];

}

}

}

}