

Scientific Computing & Simulation Report

This report will be divided into two sections. The first section will comprise of the lab reports that have been completed each week. The second section will be a development report of the mini-project I will be undertaking.

Table of Contents

1	Lab Report	3
1.1	Lab 1 – Fourier Transforms for Image Filtering	3
1.1.1	Doing a Fourier Transform	3
1.1.2	Inverse Fourier Transform	4
1.1.3	Filtering Images with Fourier Transforms	5
1.2	Lab 2 – The Fast Fourier Transform	9
1.2.1	Using the FFT	9
1.2.2	Exercises	12
1.3	Lab 3 – Inverting the Radon Transform	14
1.3.1	The Given Program	14
1.3.2	Filtered Back Projection	14
1.3.3	Exercise	16
1.4	Lab 4 – An Attempt at Sky Imaging	18
1.4.1	A Simple Imaging Program	18
1.5	Lab 5 – Lattice Gas Models	19
1.5.1	A Lattice Gas on a Square Grid – The HPP Model	19
1.5.2	An Approach to Coding the FHP Model	20
1.6	Lab 6 – Cellular Automata, Excitable Media & Cardiac Tissue	21
1.6.1	Excitable Media	21
1.6.2	Gerhardt Schuster Tyson (GST) Model	23
1.7	Lab 7 – A Lattice Boltzmann Model	25
1.7.1	Running The Code – 5000 Iteration	25
1.7.2	Code Optimisation – Loop Unrolling	26
1.8	Lab 8 – An Attempted Application to Aerodynamics	27
1.8.1	Running the Code – Angle of Attack: 0 Degrees	27
1.8.2	Exercise	27
1.9	Lab 9 – Solution of Differential Equations	29
1.9.1	Newton’s Equation of Motion	29
1.9.2	Using the Velocity Verlet Algorithm	29

2	Mini Project	30
2.1	Simultaneous Algebraic Reconstruction Technique (SART) for Reconstruction of CT Scan Images	30
2.1.1	Original Code Execution.....	30
2.1.2	Benchmarking	33
2.1.3	More Iterations of SART Reconstruction	33
3	Conclusion	35
4	References.....	36

1 Lab Report

1.1 Lab 1 – Fourier Transforms for Image Filtering

The goal for this lab session is to better understand Fourier Transforms, along with its inverse and its use for image filtering. This section will show the code for a naïve two-dimensional Discrete Fourier Transform (DFT) which will be based on the formulae shown in the lecture slide which will be applied to a simple filtering of a simple image.

1.1.1 Doing a Fourier Transform

Forward 2D DFT:

$$C_{kl} = 1/N^2 \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} X_{mn} \cdot e^{-2\pi i(km+nl)/N}$$

Figure 1.1: Forward 2D DFT Formula

With the help of the formula in Figure 1.1, the main body code inside of the nested for loop can be shown; 'arg' represents the exponential argument for 'e'. When doing DFT, half of the frequencies are sine wave (sumIm), and the other half are cosine waves (sumRe).

```
// Nested for loops performing sum over X elements
for(int m = 0; m < N; m++) {
    for(int n = 0; n < N; n++) {
        double arg = -2 * Math.PI * (m * k + n * l) / N;
        double cos = Math.cos(arg) ;
        double sin = Math.sin(arg) ;
        sumRe += cos * X [m] [n] ;
        sumIm += sin * X [m] [n] ;
    }
}
```

Figure 1.2: Code for Nested Loops

The image below shows output for the Fourier transform graph. The image on the left is the original image used and the right is the Fourier transform graph. The FT graph shows the different frequencies of the original image.

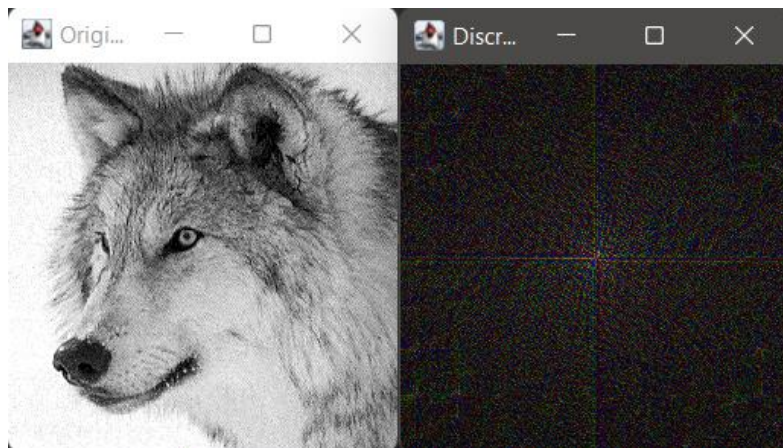


Figure 1.3: Original Image (Left) & Discrete Fourier Transform Output (Right)

1.1.2 Inverse Fourier Transform

Inverse 2D DFT:

$$X_{mn} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} C_{kl} \cdot e^{2\pi i(km+nl)/N}$$

Figure 1.4: Inverse 2D DFT

Using the formula above, the nested for loop constructed for the inverse 2D DFT is similar to Forward 2D DFT with the difference being that nested for loop uses $N - 1$ rather than N to represent the inverse 2D DFT formula and that reconstructed array takes in sum which is the calculation of $ac - bd$.

```
// ---- INVERSE 2D FT ---- //
double [][] reconstructed = new double [N] [N] ;

for(int m = 0 ; m < N ; m++) {
    for(int n = 0 ; n < N ; n++) {
        double sum = 0 ;
        // ... nested for loops performing sum over C elements
        for (int k = 0; k < N - 1 ; k++) {
            for (int l = 0; l < N - 1; l++) {
                double arg = 2 * Math.PI * (m * k + n * l) / N;
                double cos = Math.cos(arg) ;
                double sin = Math.sin(arg) ;
                sum += (cos * CRe [k] [l]) - (sin * CIm [k] [l]);
            }
        }
        reconstructed [m] [n] = sum ;
    }
    System.out.println("Completed inverse FT line " + m + " out of " + N) ;
}
DisplayDensity display3 = new DisplayDensity(reconstructed, N, title: "Reconstructed Image") ;
```

Figure 1.5: Inverse FT Code

Comparing the original image and the reconstructed image, it is seen that the image is comparatively lighter than the original image. This is due to the loss of some floating-point number in the process.

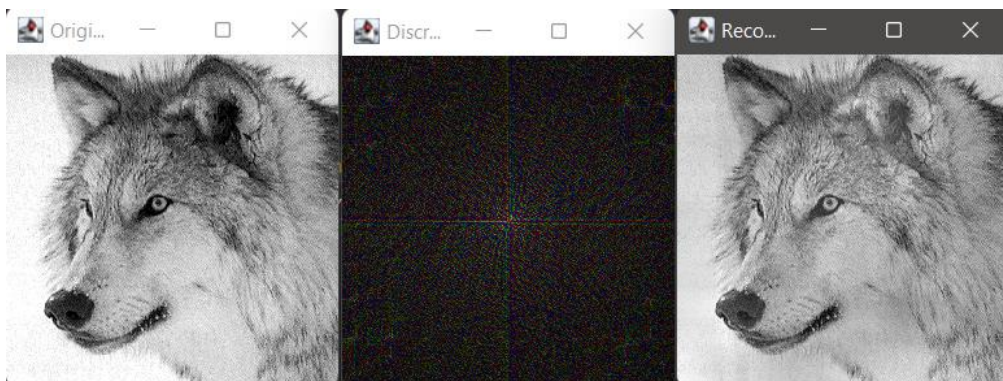


Figure 1.6: Original Image (Left), DFT Output (Centre) & Reconstructed Image (Right)

1.1.3 Filtering Images with Fourier Transforms

1.1.3.1 Low Pass Filtering

The image filtering code, below, was provided in this week's lab sheets. The *cut-off* value is used to dictate how much the DFT would be filtered. The image filtering code below represents a simple low pass filter by omitting Fourier components with large wave numbers before reconstructing the image.

```
// ---- IMAGE FILTERING ---- //
int cutoff = N/8 ; // for example
for(int k = 0 ; k < N ; k++) {
    int kSigned = k <= N/2 ? k : k - N ;
    for(int l = 0 ; l < N ; l++) {
        int lSigned = l <= N/2 ? l : l - N ;
        if(Math.abs(kSigned) > cutoff || Math.abs(lSigned) > cutoff) {
            CRe [k] [l] = 0 ;
            CIm [k] [l] = 0 ;
        }
    }
}

Display2dFT display2a = new Display2dFT(CRe, CIm, N, "Truncated FT") ;
```

Figure 1.7: Image Filtering Code

The reconstructed image below is shown to be much blurrier than the original image, this is due to the low pass filter omission that were mentioned earlier. Figure 1.9 represents how the image is reconstructed with different cut-off values. The main takeaway from the table is that by increasing the denominator of the calculation, the blurrier it gets due to that fact that range of what counts as large wave number increase causing more to be omitted in reconstruction.

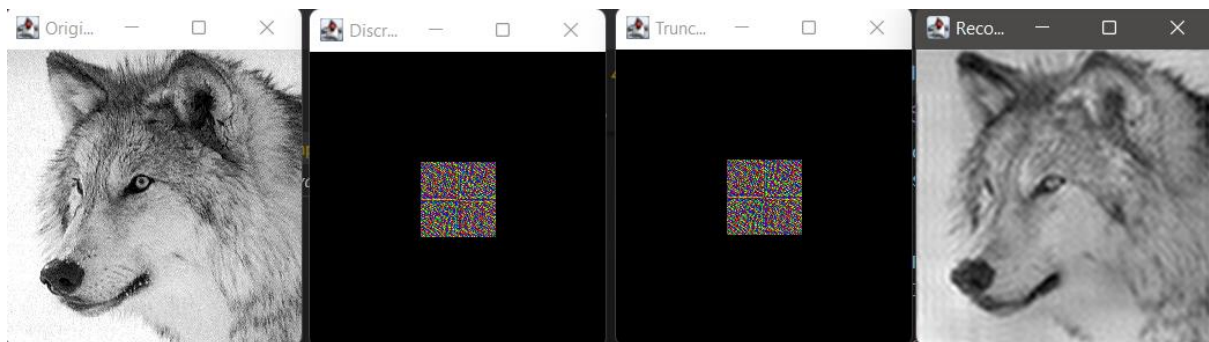
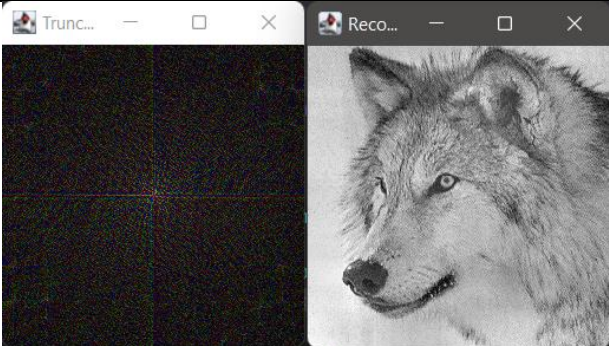


Figure 1.8: Original Image (Left), Truncated FT (Centre) & Reconstructed Image (Right)

Cut-off Value	Image
N/2	

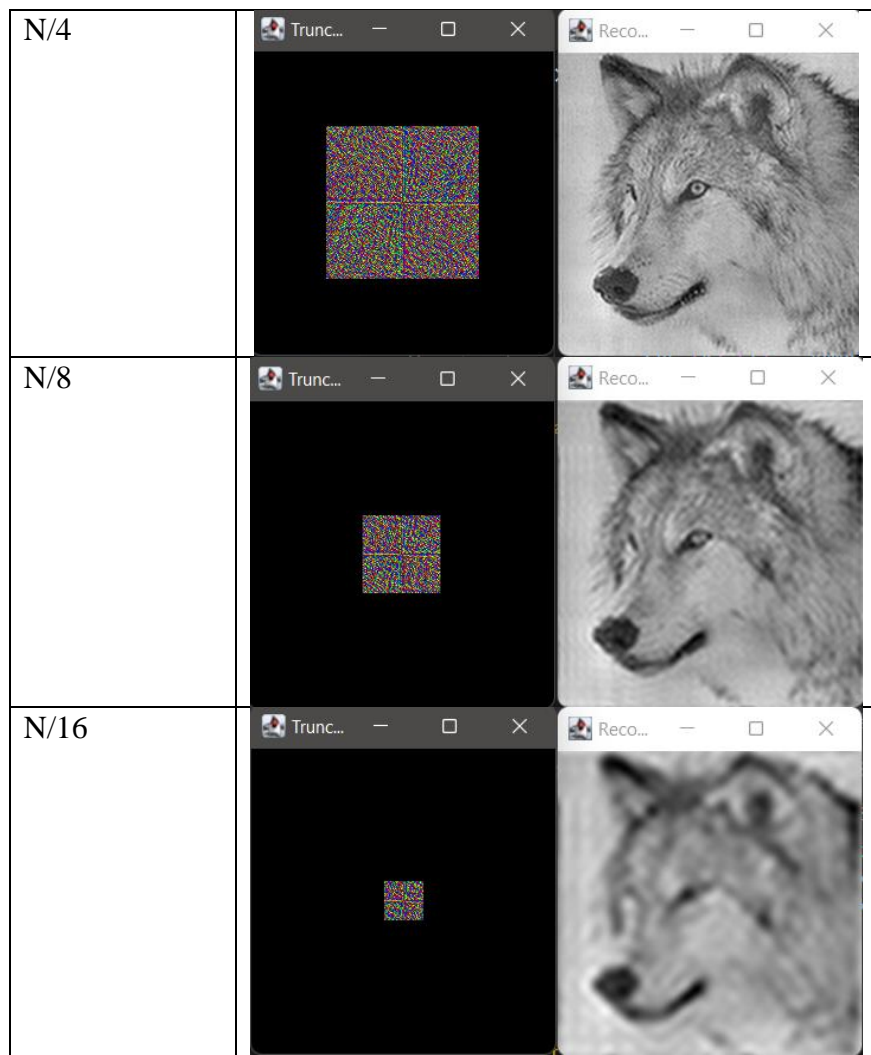


Figure 1.9: Cut-off Value Comparison

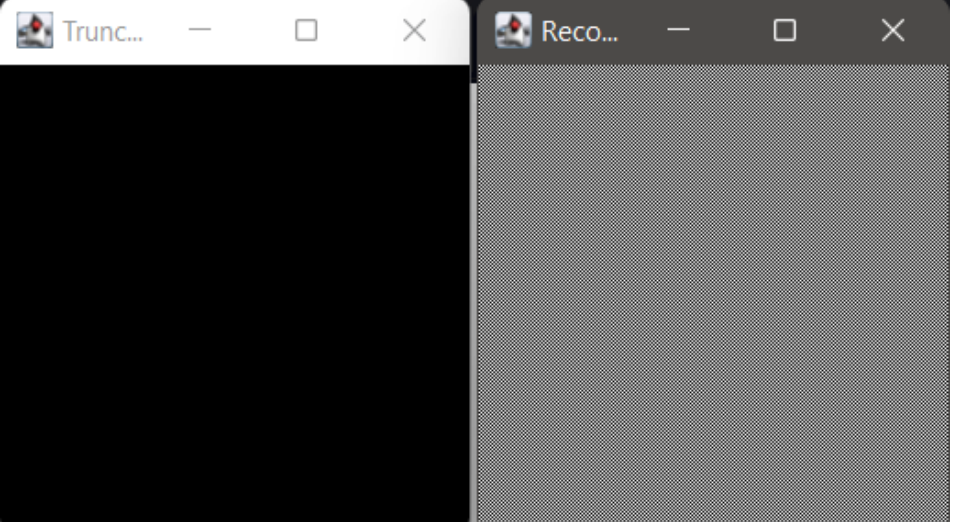
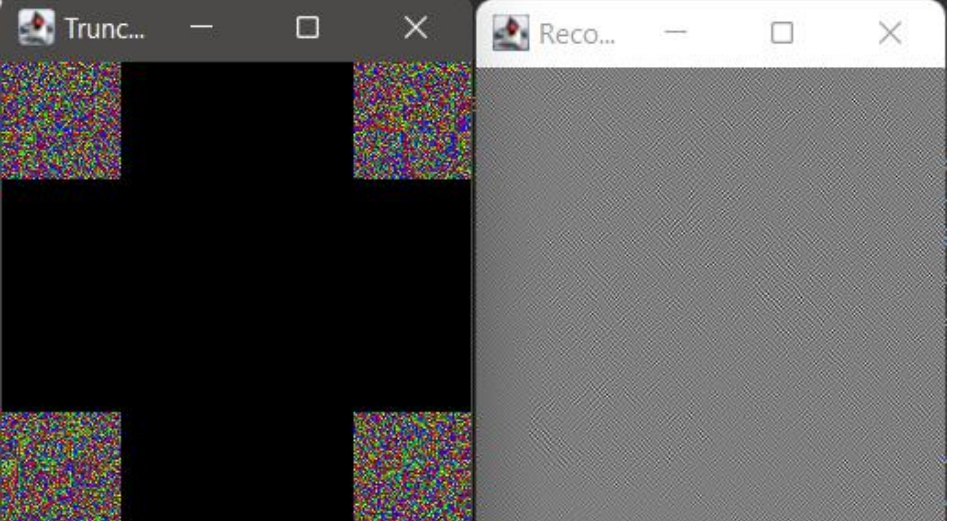
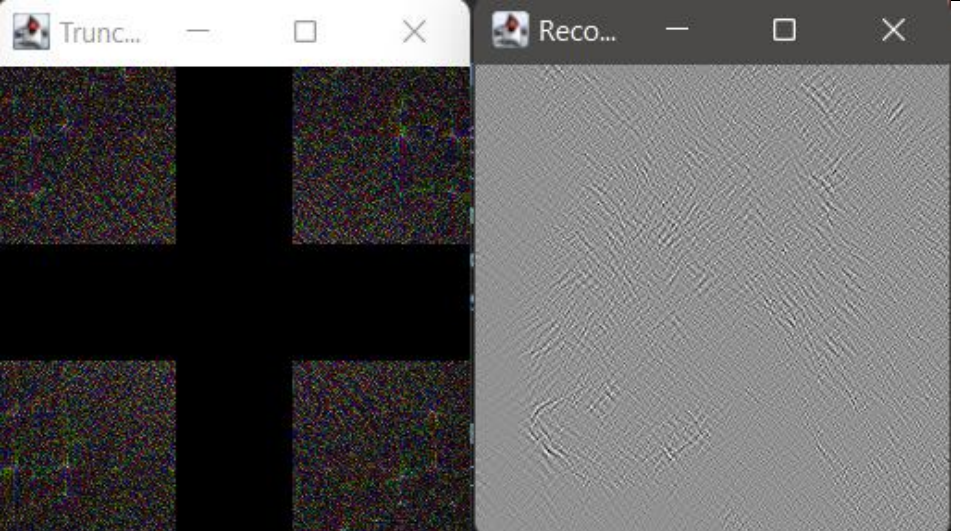
1.1.3.2 High Pass Filtering

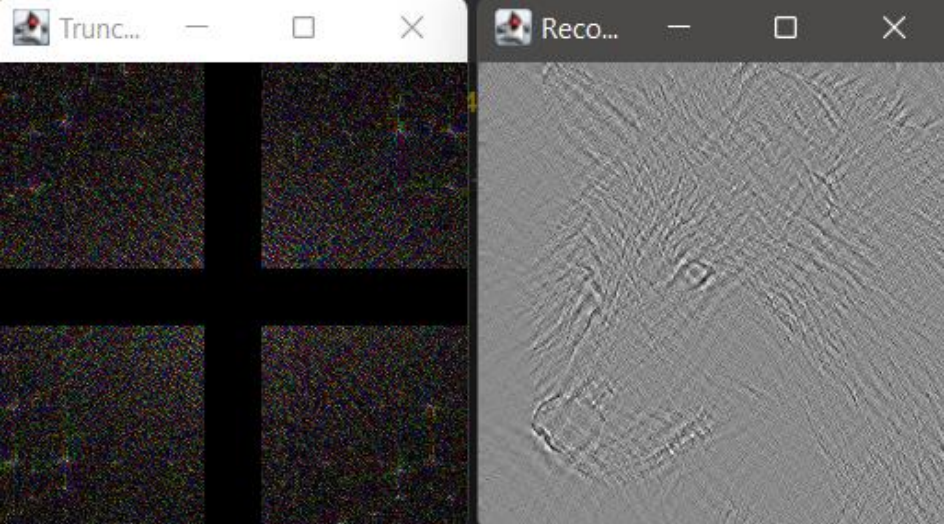
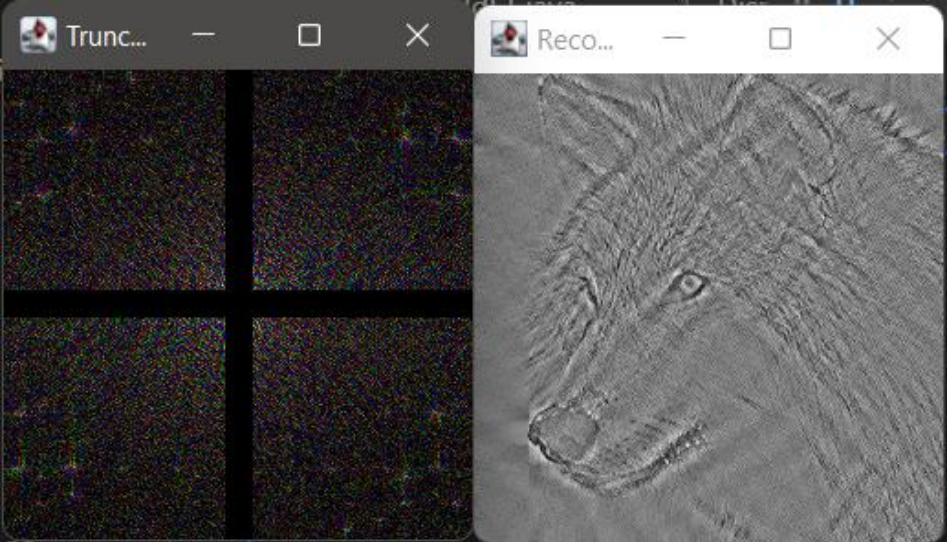
For high pass filtering, the same code from Figure 1.7 was used but with a single change which was to invert the if statement as shown below.

```
if(Math.abs(kSigned) < cutoff || Math.abs(lSigned) < cutoff)
```

Figure 1.10: Image Filtering Code Change [High Pass]

As shown in the figure below, as the value of the denominator of the cut-off value increase, the image slowly comes into view.

Cut-off Value	Image
N/2	
N/4	
N/8	

N/16	
N/32	

1.2 Lab 2 – The Fast Fourier Transform

This week will look at the Fast Fourier Transform (FFT) and applying the FFT to the image filtering task from Week 1.

1.2.1 Using the FFT

The transpose function, shown in Figure 1.11, is used for the purpose of image filtering during the program and below that is to iterate through the given parameter, in this case ‘double [][] a’. It converts the rows of the input data into columns so that the 1D FFT can be applied to each column because FFT is more efficient when applied to columns rather than rows.

```
//... implementation of fft2d ...
static void transpose(double [][] a) {
    // Image Filtering
    int cutoff = N/4;
    for(int k = 0 ; k < N ; k++) {
        int kSigned = k <= N/2 ? k : k - N ;
        for(int l = 0 ; l < N ; l++) {
            int lSigned = l <= N/2 ? l : l - N ;
            if(Math.abs(kSigned) > cutoff || Math.abs(lSigned) > cutoff) {
                a [k] [l] = 0 ;
            }
        }
    }

    for(int i = 0 ; i < cutoff ; i++) {
        for(int j = 0 ; j < i ; j++) {
            // ... swap values in a [i] [j] and a [j] [i] elements ...
            a [i] [j] = a [j] [i];
        }
    }
}
```

Figure 1.11: Transpose Function

The fft2d function, shown in figure 1.12, is used to do an FFT on the image with the help of the fft1d class that was provided to us in this week’s lab sheet. The function applies 1D FFT to all the rows of the 2D array and then transposes it and then reapplies 1D FFT to the rows (now columns) and then that transposed again to revert the array back to its original orientation. 1D FFT is applied twice to compute 2D DFT efficiently using the FFT algorithm.

```

static void fft2d(double [] [] re, double [] [] im, int isgn) {
    FFT fft1D = new FFT();

    // For simplicity, assume square arrays
    // ... fft1d on all rows of re, im ...

    for(int i = 0 ; i < N ; i++) {
        fft1D.fft1d(re[i], im[i], isgn);
    }

    transpose(re) ;
    transpose(im) ;

    // ... fft1d on all rows of re, im ...

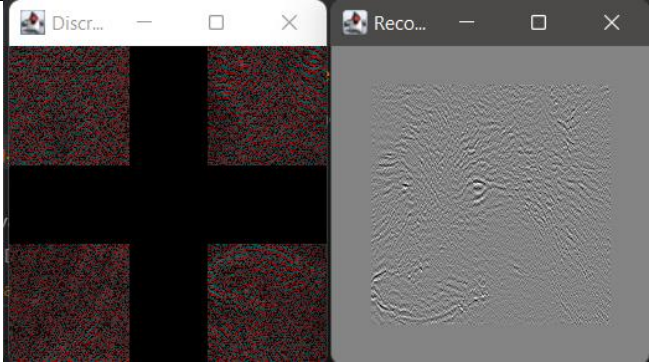
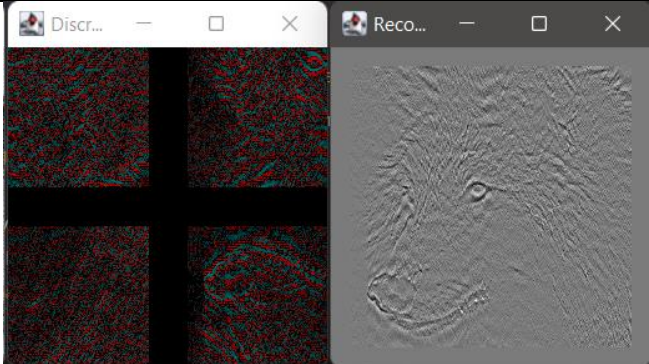
    for(int j = 0 ; j < N ; j++) {
        fft1D.fft1d(re[j], im[j], isgn);
    }

    transpose(re) ;
    transpose(im) ;
}

```

Figure 1.12: 'fft2d' Function

The figure below shows both the FT and Inverse FT outputs for each of the cut-off values ranging from $N/8$ to $N/256$. When looking at the cut-off value of $N/8$, it can be seen that the reconstructed image is much smoother compared to the original wolf image because the high-frequency components of the image have been removed. As the denominator value of the cut-off variable increase, fewer high-frequency components are removed resulting in a sharper image eventually leading to a near identical image compared to the original.

Cut-off Value	Image
N/8	
N/16	

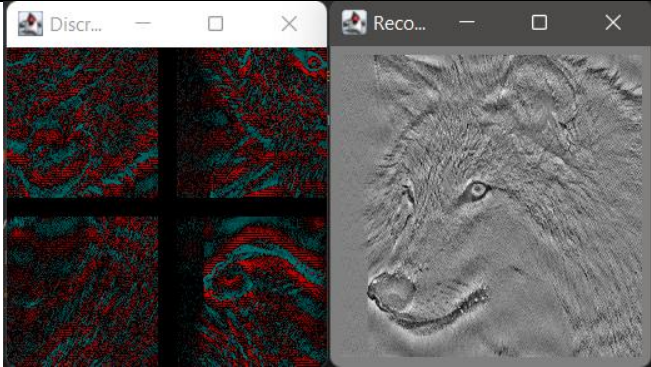
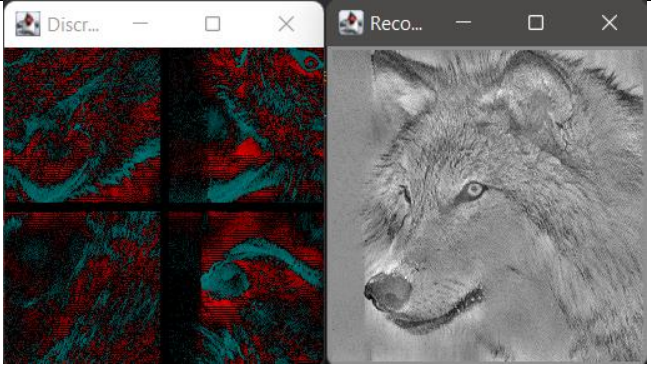
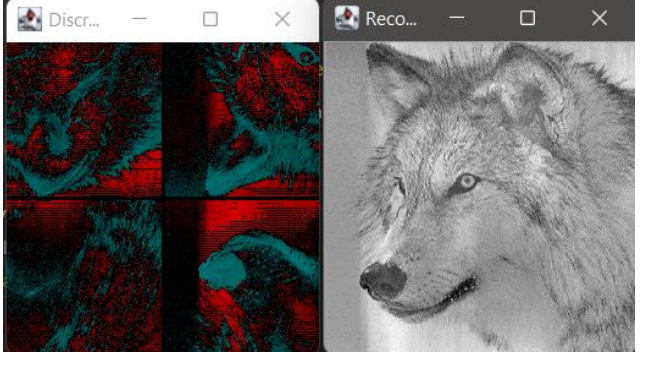
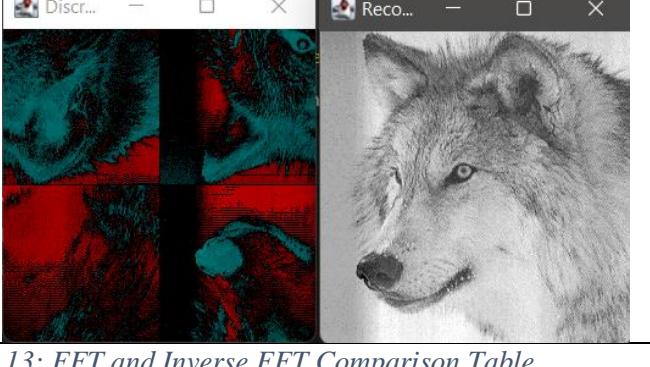
N/32	
N/64	
N/128	
N/256	

Figure 1.13: FFT and Inverse FFT Comparison Table

1.2.2 Exercises

1.2.2.1 Benchmark Comparisons 2D FFT and Naïve FT Code from Week 1

The table below shows the time taken for both Forward FT and Inverse FT using a high pass filter. The results show that program from week 2 is significantly faster when compared to the program used for week 1.

Cut-off Value	Forward FT Time Taken (ms)	Inverse FT Time Taken (ms)
Lab 2: 2D FFT Code		
N/8	46	56
N/16	46	55
N/32	47	59
N/64	52	60
N/128	47	56
N/256	56	54
Lab 1: Naïve FT Code		
N/8	336910	323533
N/16	337358	319754
N/32	336345	319899
N/64	341705	319164
N/128	344481	335996
N/256	336673	324193

Figure 1.14: Week 1 & 2 - FT & Inverse FT Benchmark Results [High Pass Filtering]

The figure below shows the placement of the code used for the benchmarking results. For the above results, the output of the display of the FT and Inverse FT are included in the times.

```

long FTstartTime = System.currentTimeMillis(); // <-- new code
// create array for in-place FFT, and copy original data to it
double [] [] CRe = new double [N] [N], CIm = new double [N] [N] ;
for(int k = 0 ; k < N ; k++) {
    for(int l = 0 ; l < N ; l++) {
        CRe [k] [l] = X [k] [l] ;
    }
}

fft2d(CRe, CIm, isgn:1) ; // Fourier transform

Display2dFT display2 = new Display2dFT(CRe, CIm, N, title: "Discrete FT") ;
long FTendTime = System.currentTimeMillis(); // <-- new code

long IFTstartTime = System.currentTimeMillis(); // <-- new code
// create array for in-place inverse FFT, and copy FT to it
double [] [] reconRe = new double [N] [N],
            reconIm = new double [N] [N] ;
for(int k = 0 ; k < N ; k++) {
    for(int l = 0 ; l < N ; l++) {
        reconRe [k] [l] = CRe [k] [l] ;
        reconIm [k] [l] = CIm [k] [l] ;
    }
}

fft2d(reconRe, reconIm, -1) ; // Inverse Fourier transform

DisplayDensity display3 = new DisplayDensity(reconRe, N, title: "Reconstructed Image") ;
long IFTendTime = System.currentTimeMillis(); // <-- new code

System.out.println("FT Time = " + (FTendTime - FTstartTime) + "ms"); // <-- new code
System.out.println("IFT Time = " + (IFTendTime - IFTstartTime) + "ms"); // <-- new code

```

Figure 1.15: Placement of Benchmark Code

1.2.2.2 Experimentation using Low Pass & High Pass Filtering with Week 1 Naïve FT Code
 For this exercise refer to Section 1.1.3 which shows the results of both low pass filtering and high pass filtering at different cut-off values.

1.3 Lab 3 – Inverting the Radon Transform

This week's task is to reverse the process to recover the original model from the sinogram (Radon Transform)

1.3.1 The Given Program

Using the program that was given in the lab sheet and without modifying the code, the resulted outputs are displayed below. The source model (on the left) represents the density values that will be used to generate the sinogram output (central image)

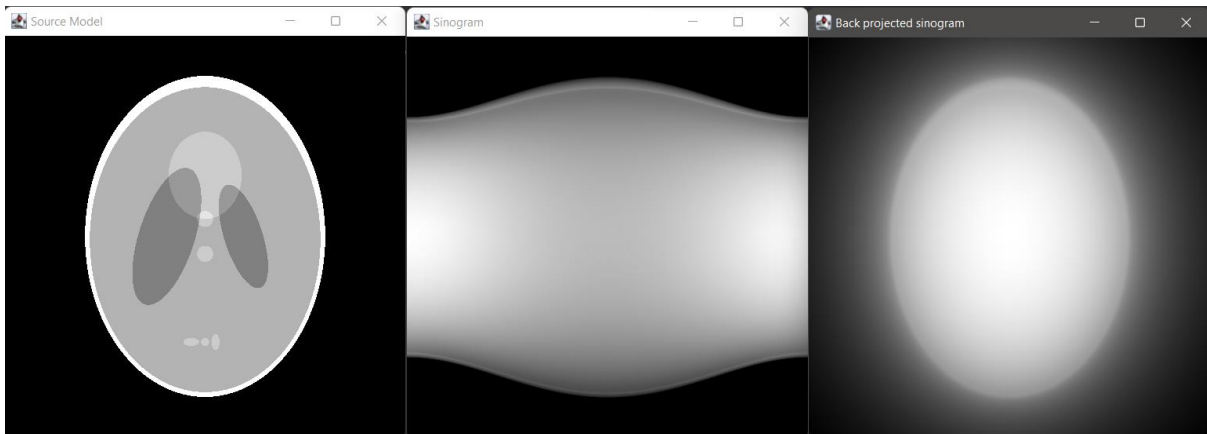


Figure 1.16: Given Program Code Outputs

1.3.2 Filtered Back Projection

The code below, Figure 1.17, has the purpose of performing a 1D FFT of a sinogram and to display the FT of the sinogram using the display tool, in this case being the DisplaySinogramFT() function. The main function of the code is done within the for loops of the code. The first for loop initialises the sinogramFTRe and sinogramFTIm2 using the values from the sinogram array. Whilst the second loop performs a 1D FFT on each row the two arrays which are then stored in sinogramFTIm and sinogramFTIm2.

```
// ---- SINOGRAM FILTERING ---- //
double [] [] sinogramFTRe = new double [N] [N],
            sinogramFTIm = new double [N] [N] ;

double [] [] sinogramFTRe2 = new double [N] [N],
            sinogramFTIm2 = new double [N] [N] ;
for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    for(int iR = 0 ; iR < N ; iR++) {
        sinogramFTRe [iTheta] [iR] = sinogram [iTheta] [iR] ;
        sinogramFTRe2 [iTheta] [iR] = sinogram [iTheta] [iR] ;
    }
}

for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    FFT.fft1d(sinogramFTRe[iTheta], sinogramFTIm[iTheta], isgn:1); // 1D FFT of Sinogram
    FFT.fft1d(sinogramFTRe2[iTheta], sinogramFTIm2[iTheta], isgn:1); // 1D FFT of Sinogram Low pass Cosine
}

DisplaySinogramFT display3 = new DisplaySinogramFT(sinogramFTRe, sinogramFTIm, N,title: "Sinogram radial Fourier Transform") ;
```

Figure 1.17: Sinogram Fourier Transform Code

The code below, Figure 1.18, shows the filtering operation of the Sinogram FT. Within the first loop, the multiplication of the FT by abs(kSigned) is done to enhance the high frequency details of the image and the if statement is done to remove any coefficient greater than the

cut-off, a low pass filter removing high frequency noise. The second loop is used to perform an inverse FT using the `fft1d` function from the FFT class.

```
for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    for(int iK = 0 ; iK < N ; iK++) {
        int kSigned = iK <= N/2 ? iK : iK - N ;

        // multiply Sinogram fourier transform by abs(kSigned)
        sinogramFTRe [iTheta] [iK] *= Math.abs(kSigned) ;
        sinogramFTIm [iTheta] [iK] *= Math.abs(kSigned) ;

        //zero components with Math.abs(kSigned) > CUTOFF
        if (Math.abs(kSigned) > CUTOFF) {
            sinogramFTRe [iTheta] [iK] = 0;
            sinogramFTIm [iTheta] [iK] = 0;
        }
    }
}

for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    FFT.fft1d(sinogramFTRe[iTheta], sinogramFTIm[iTheta], -1); // Normal Filter
}

DisplayDensity display6 = new DisplayDensity(sinogramFTRe, N, title:"Filtered sinogram") ;
```

Figure 1.18: Filtered Sinogram Code

The images below show the output for both the FT and inverse FT from both Figure 1.17 and Figure 1.18 respectively.

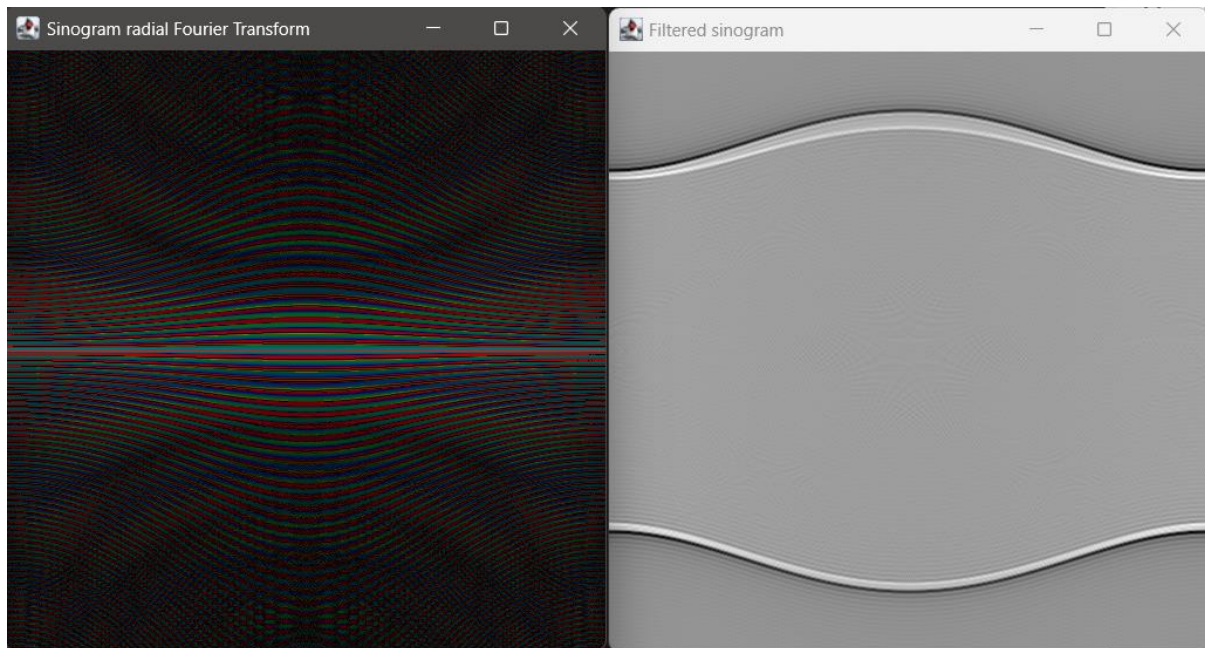


Figure 1.19: Fourier Transform (Left) & Filtered Sinogram (Right)

The code below shows the back projection of the sinograms. This done by a factor value by dividing the known density of the object by the L2 norm of the back projection. The factor value is then applied to the back projection image. The reason for the normalisation is so that we can obtain an image that shows the physical density of the image with each pixel value.

```

double [] [] backProjection = new double [N] [N] ;
backProject(backProjection, sinogramFTRe) ;

double factor = normDensity / norm2(backProjection) ;
for(int i = 0 ; i < N ; i++) {
    for(int j = 0 ; j < N ; j++) {
        backProjection [i] [j] *= factor ;
    }
}

DisplayDensity display5 = new DisplayDensity(backProjection, N, title:"Back projected sinogram",
GREY_SCALE_LO, GREY_SCALE_HI) ;

```

Figure 1.20: Back Projected Sinogram Code

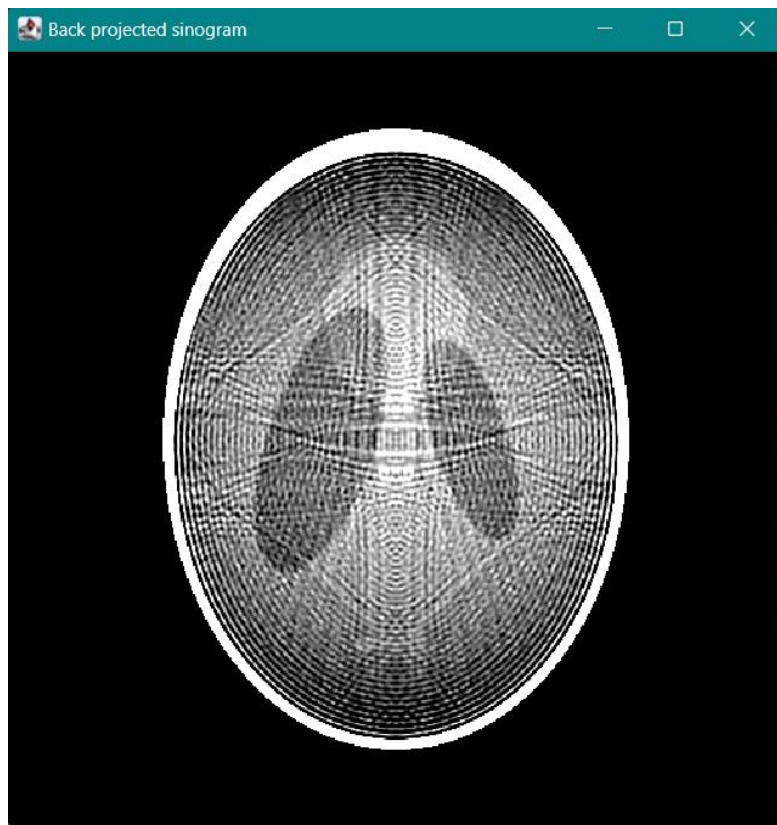


Figure 1.21: Back Projected Sinogram Output

1.3.3 Exercise

1.3.3.1 Low Pass Cosine Filter

The code below, Figure 1.22, show how the low pass cosine filter is done on the sinogram FT. The filter code is added within the nested for loop shown in Figure 1.18. The code works in a similar fashion to previous filter code where the only significant difference would be the addition of multiplication of the FT, which is already multiplied by $\text{abs}(k\text{Signed})$, by $\cos(\pi K / (2 \text{ CUTOFF}))$.

```

// ---- LOW PASS COSINE FILTER ---- // ---- |K| cos( $\pi K / (2 \text{ CUTOFF})$ ) ---- //
sinogramFTRe2 [iTheta] [iK] *= Math.abs(kSigned) ;
sinogramFTIm2 [iTheta] [iK] *= Math.abs(kSigned) ;
sinogramFTRe2 [iTheta] [iK] *= Math.cos(Math.PI * kSigned / (2 * CUTOFF));
sinogramFTIm2 [iTheta] [iK] *= Math.cos(Math.PI * kSigned / (2 * CUTOFF));

if (Math.abs(kSigned) > CUTOFF) {
    sinogramFTRe2 [iTheta] [iK] = 0;
    sinogramFTIm2 [iTheta] [iK] = 0;
}
}

for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    FFT.fft1d(sinogramFTRe[iTheta], -1); // Normal Filter
    FFT.fft1d(sinogramFTRe2[iTheta], sinogramFTIm2[iTheta], -1); // Low Pass Cosine Filter
}

DisplayDensity display6 = new DisplayDensity(sinogramFTRe, N, title:"Filtered sinogram") ;
DisplayDensity display7 = new DisplayDensity(sinogramFTRe2, N, title:"Filtered sinogram, Low Pass Cosine Filter") ;

```

Figure 1.22: Low Pass Cosine Filter Code

The back projection code below works in the same manner as the one that was presented above however the only difference is that it uses the data related to low pass cosine filtering rather than the normal filtering data.

```

double [] [] backProjection2 = new double [N] [N] ;
backProject(backProjection2, sinogramFTRe2) ;

double factor2 = normDensity / norm2(backProjection2) ;
for(int i = 0 ; i < N ; i++) {
    for(int j = 0 ; j < N ; j++) {
        backProjection2 [i] [j] *= factor2 ;
    }
}

DisplayDensity display8 = new DisplayDensity(backProjection2, N, title:"Back projected sinogram, Low Pass Cosine Filter",
GREY_SCALE_LO, GREY_SCALE_HI) ;

```

Figure 1.23: Low Pass Cosine Filter Back Projected Code

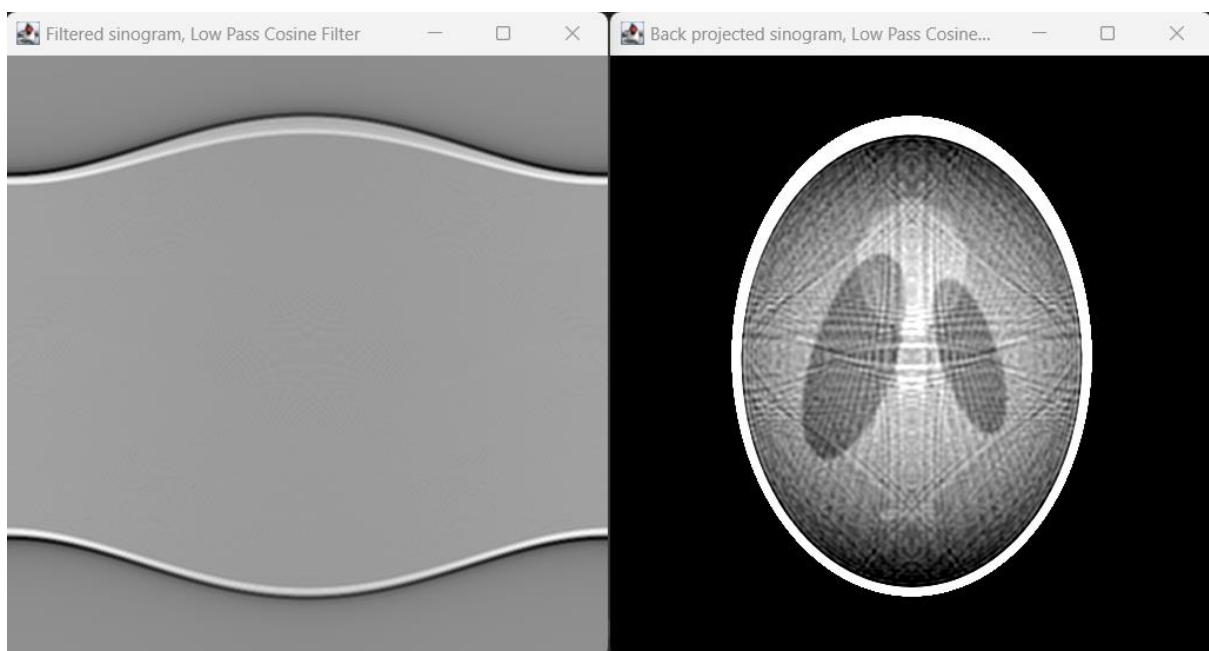


Figure 1.24: Low Pass Cosine Filter Output

1.4 Lab 4 – An Attempt at Sky Imaging

The goal of this lab work is to apply the principles of imaging from radio interferometry data.

1.4.1 A Simple Imaging Program

The code that was given in the lab worksheet is used to perform an imaging simulation of radio interferometry data. This involves taking the FT of measured visibilities and constructing images using dirt map and dirty beam (point spread function).

The `main()` method code reads the data from a `vis-and-uv.txt` file that was given to us to use. The file contains measured visibilities and their corresponding (u,v) coordinates. The program is used to calculate the real and imaginary parts of the visibilities, the U-V coverage window in the figure below. Using a 2D Inverse FT, a raw image (dirty image) and a point spread function (PSF) is created. The PSF represents the pattern of light that is produced by a single point source in the image plane.

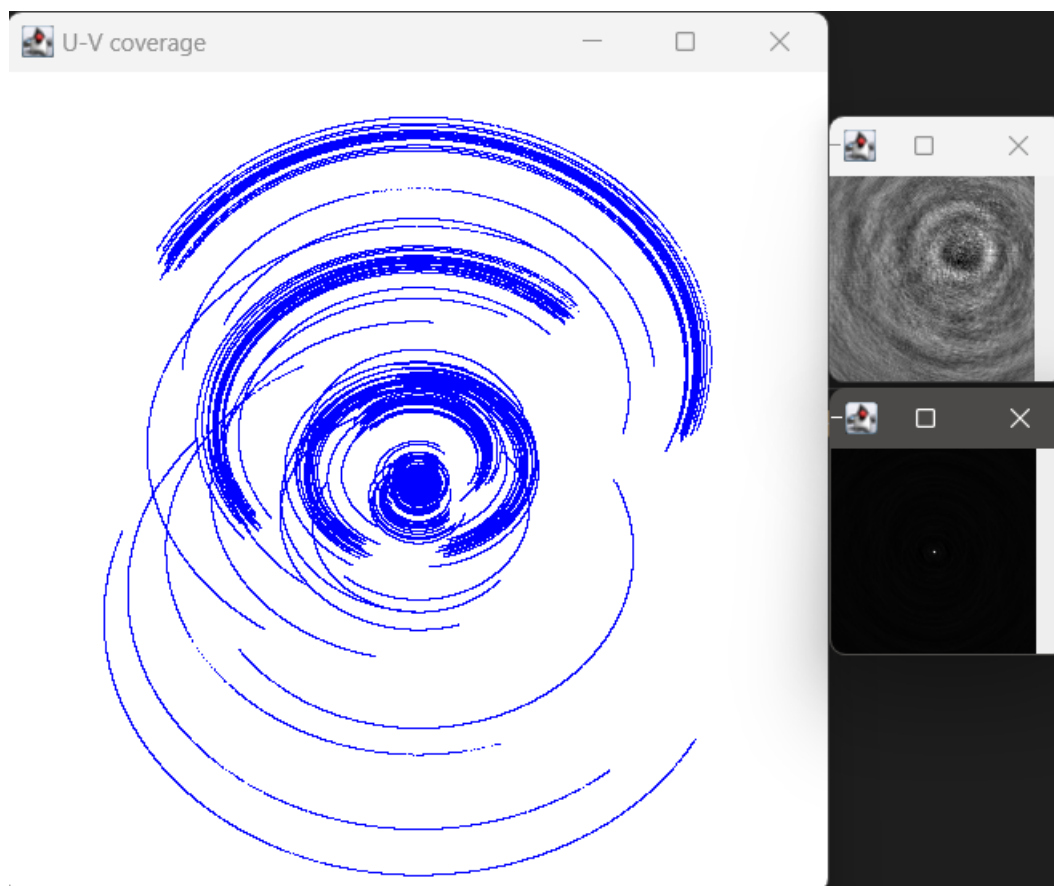


Figure 1.25: U-V Coverage (Left), Dirty Image (Top Right) & Point-Spread Function (Bottom Right)

1.5 Lab 5 – Lattice Gas Models

1.5.1 A Lattice Gas on a Square Grid – The HPP Model

The code snippet below, represents the collision rules that is used to simulate fluid flow in the lattice gas model. The first part of the code initialises the head on collision of the two particles. The code after that is used to calculate the new distribution of particles after the occurrence of the collision.

```
// initialize - staged head on collision in x direction.
fin[NX/2+1] [NY/2] [0] = true;
fin[NX/2-1] [NY/2] [1] = true;

// make only the states
if(fin_ij [0] && fin_ij [1] && ! fin_ij[2] && ! fin_ij[3]) {
    fout_ij [0] = false ;
    fout_ij [1] = false ;
    fout_ij [2] = true ;
    fout_ij [3] = true ;
} else if(fin_ij [2] && fin_ij [3] && ! fin_ij[0] && ! fin_ij[1]) {
    fout_ij [0] = true ;
    fout_ij [1] = true ;
    fout_ij [2] = false ;
    fout_ij [3] = false ;
} else {
    fout_ij = fin_ij;
}
```

Figure 1.26: Collision Rules Code

The code snippet below is implementing the streaming steps of the model.

```
// streaming using HPP model
fin [i] [j] [0] = fout [iP1] [j] [0] ;
fin [i] [j] [1] = fout [iM1] [j] [1] ;
fin [i] [j] [2] = fout [i] [jP1] [2] ;
fin [i] [j] [3] = fout [i] [jM1] [3] ;
```

Figure 1.27: Streaming Step Code

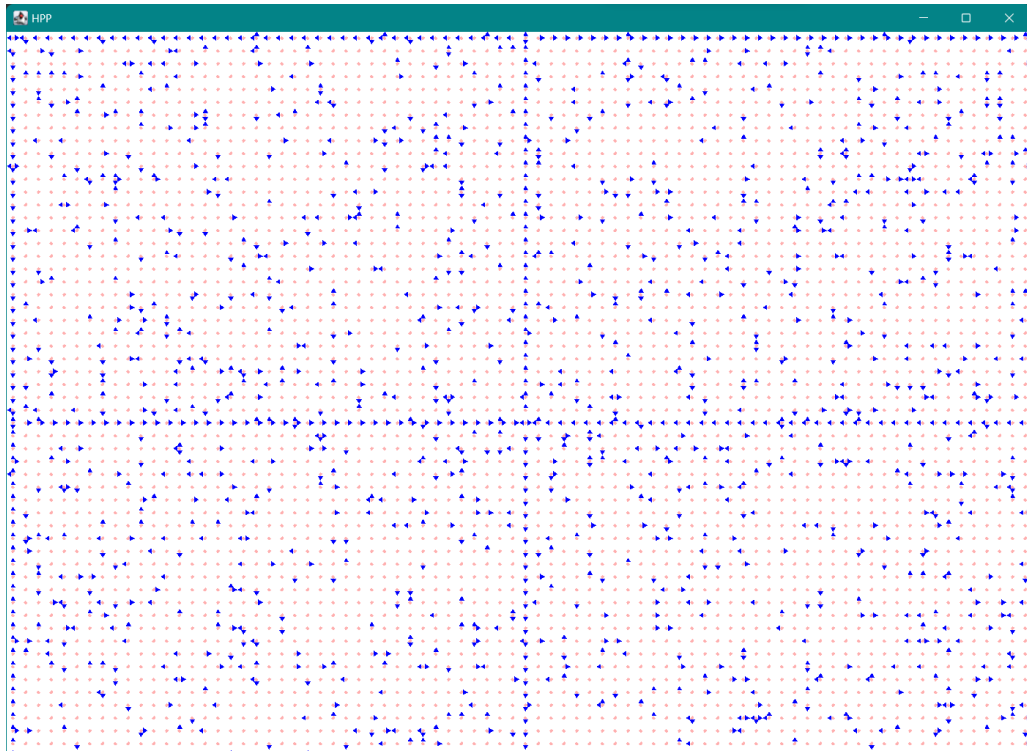


Figure 1.28: HPP Simulation

1.5.2 An Approach to Coding the FHP Model

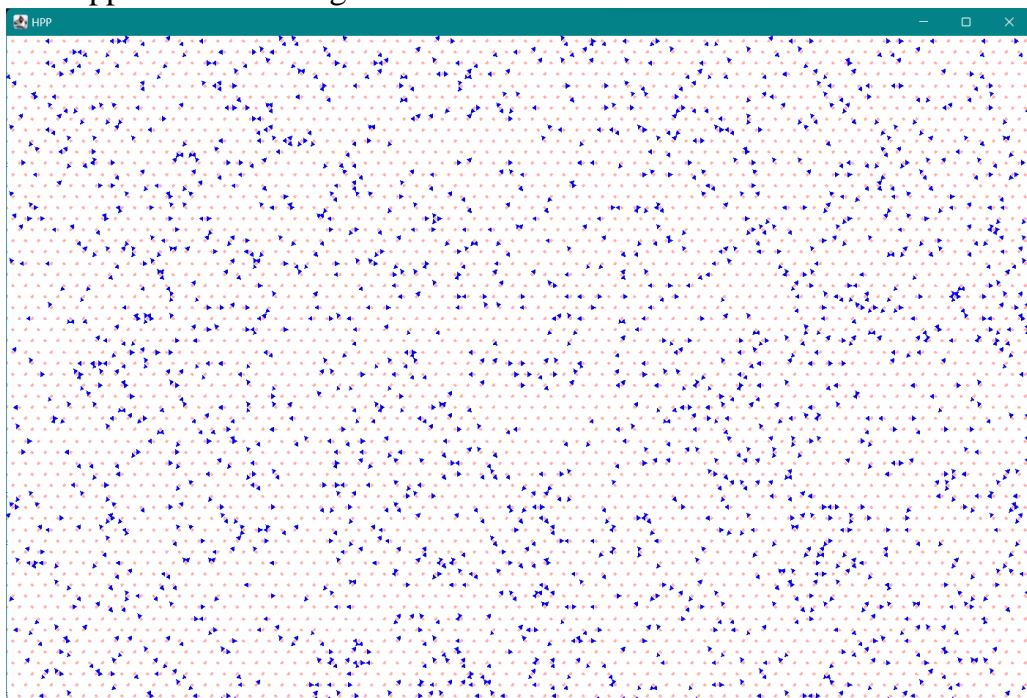


Figure 1.29: FHP Simulation

1.6 Lab 6 – Cellular Automata, Excitable Media & Cardiac Tissue

1.6.1 Excitable Media

The images shown below, represent the different stages of the output of the code that was provided in the lab sheet where none of the code has been altered. When the code is executed, the plane wave starts from the bottom and moves it way up (left window) then halfway up it starts to turn into a spiral wave (centre window) where finally, it will constantly iterate a spiral wave (right window).

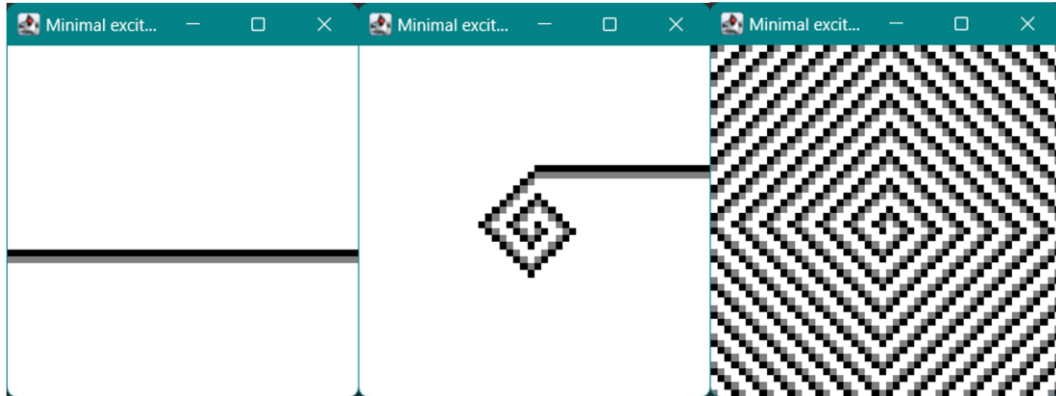


Figure 1.30: Excitable Media Model

1.6.1.1 Code Modification – Start at Centre

The code below is used so that when the simulation is executed it will start from the centre of the grid/window.

```
// Define initial state - excited center cell / resting elsewhere.
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        state[i][j] = 0;
    }
}
state[N/2][N/2] = 2; // set center cell to excited
```

Figure 1.31: Code Modification - Start at the Centre

The image below shows how the cells started at the centre of the grid (left window) and the right window show that patter that is formed.

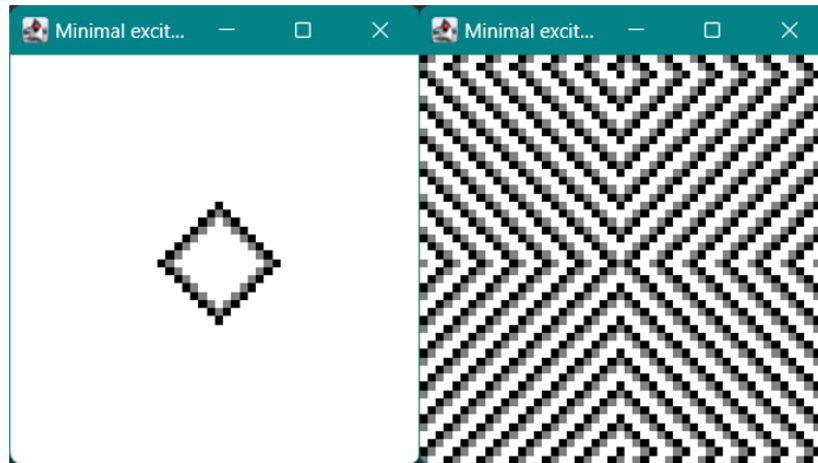


Figure 1.32: Excitable Media Model - Centre Start

1.6.1.2 Code Modification – Start at Bottom Right

The code below is used to have the cells start at the bottom right corner of the grid. However, unlike the single wave from the original code, the one below does not split off into a spiral.

```
// Define initial state - excited bottom right corner cell / resting elsewhere.
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        state[i][j] = 0;
    }
}
state[N-1][N-1] = 2; // set bottom right corner cell to excited
```

Figure 1.33: Code Modification - Start at Bottom Right

As seen below, the simulation starts from the bottom right and continues to the top left.

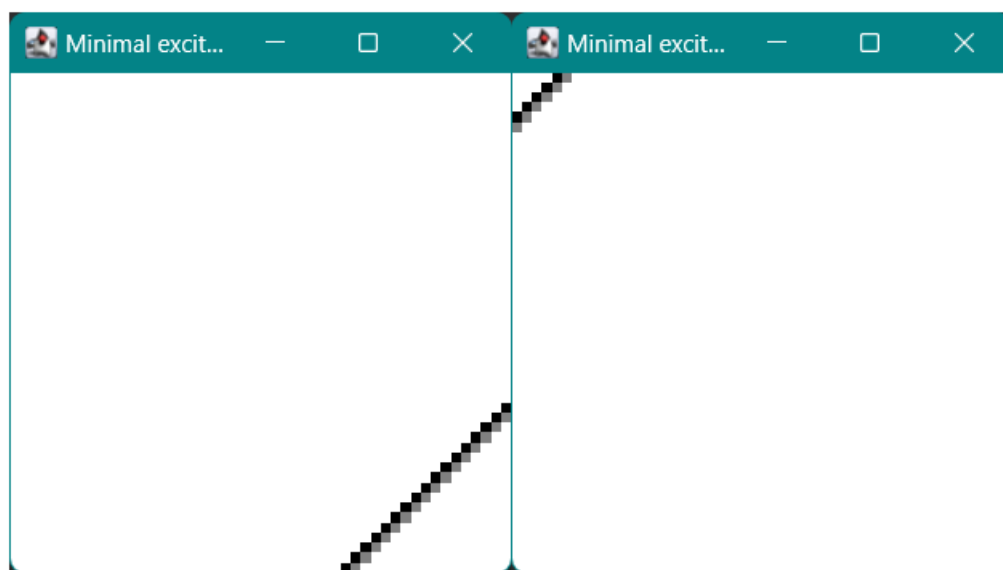


Figure 1.34: Excitable Media Model - Start at Bottom Left

1.6.1.3 Code Modification – Start with Combination of Centre and Bottom Right.

The code below is used to execute the simulation with starting at both the centre and bottom right.

```
// Define initial state - excited in center and bottom right corner.
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if ((i == N / 2 && j == N / 2) || (i == N - 1 && j == N - 1)) {
            state[i][j] = 2;
        } else {
            state[i][j] = 0;
        }
    }
}
```

Figure 1.35: Combination of Bottom Right and Centre

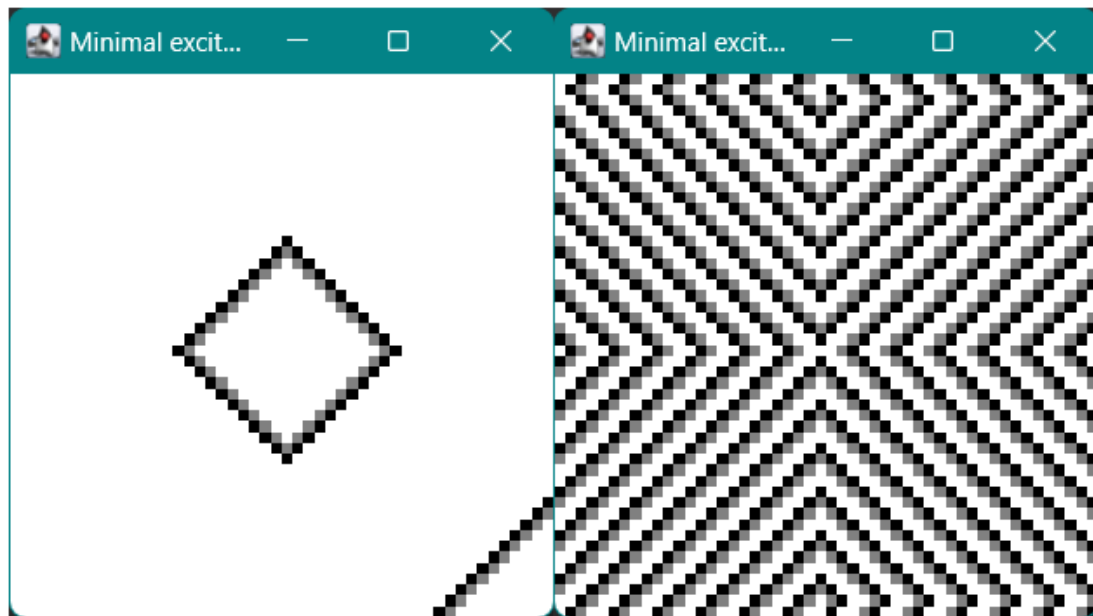


Figure 1.36: Start of Simulation (Left), Continuous Iterations (Right)

1.6.2 Gerhardt Schuster Tyson (GST) Model

Using the code that was provided, the simulation that occurred when executing the code works in a similar fashion to the excitable media model shown previously however, the GST model is shown to be simulated in a larger scale with finer details.

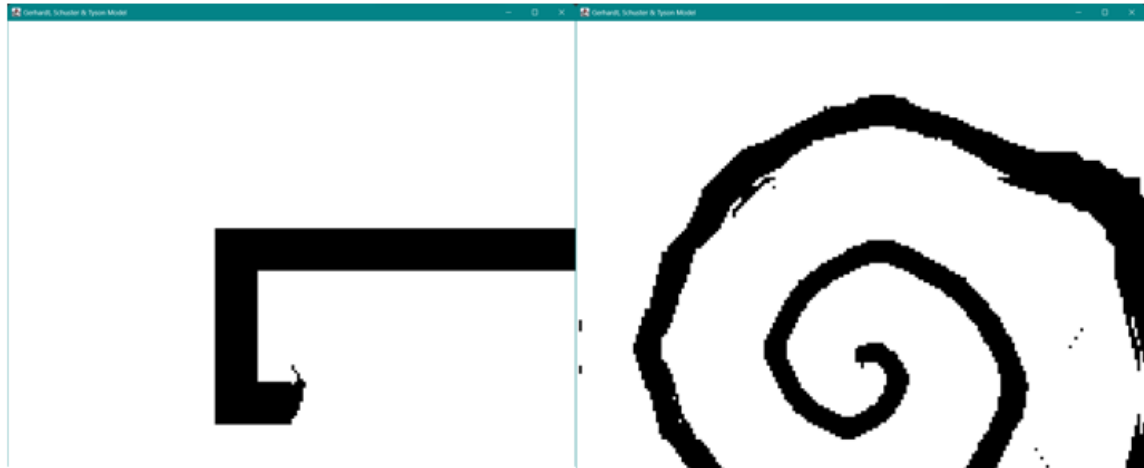


Figure 1.37: GST Model Simulation

1.6.2.1 Code Modification – Start in the Top Right

```
// Define initial state - excited top right cell / resting elsewhere.
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        u[i][j] = 0;
        v[i][j] = 0;
    }
}
u[N - 1][0] = 1; // set top right cell to be excited
```

Figure 1.38: Code for Top Right Initial State

Similarly, to the previous code modification in the previous section, when executing the code, the simulation starts in the top right corner and continues towards the bottom left corner.

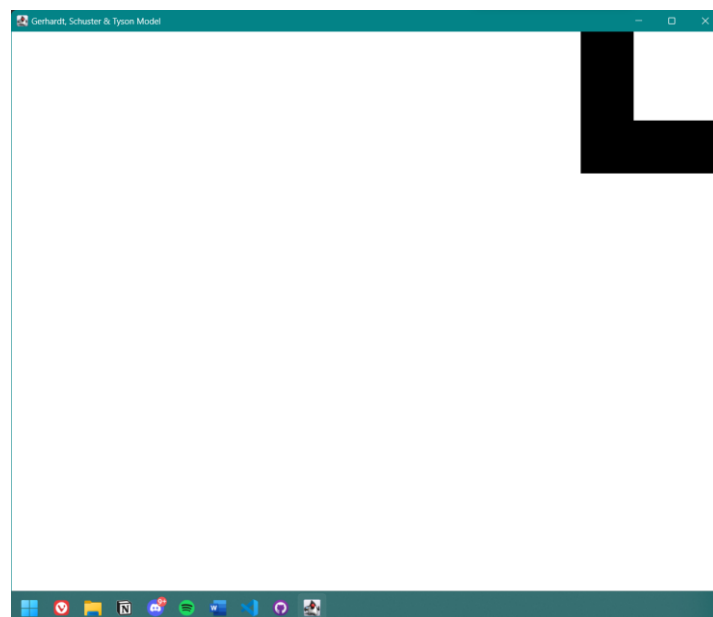


Figure 1.39: Simulation Start in Top Right

1.6.3 FourStateCA

The FourStateCA is a program that has been modified from the SimpleThreeStateCA program that was used in section 1.6.1. The improved version allows for a better reproduction of cardiac potentials.

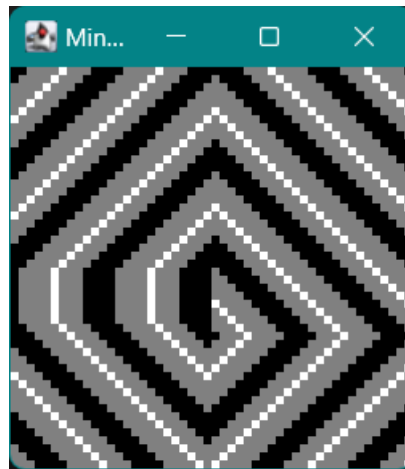


Figure 1.40: FourStatesCA Simulation

1.7 Lab 7 – A Lattice Boltzmann Model

1.7.1 Running The Code – 5000 Iteration

The figure below displays the results of the simulation of the LBM at the end of the 5000 iterations. The red sections represent a fast flow of fluid, and the blue sections represent a slow flow. At the end of the simulation of the model, below, represents a steady flow of fluid.

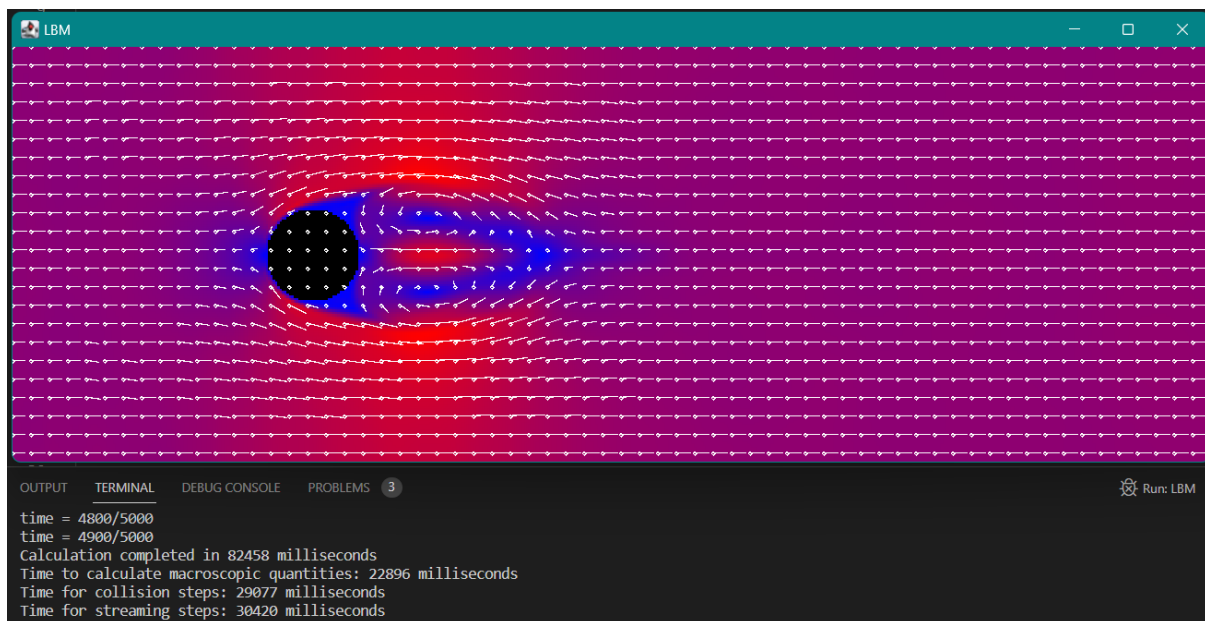


Figure 1.41: LBM - End of Simulation

1.7.2 Code Optimisation – Loop Unrolling

The simulation below was done by changing the code to use loop unrolling which replaces the loops in the program with a sequence of operations repeating a constant number of times, which helps reduce overheads of branching and loops.

Comparing the time taken between the model without loop unrolling (Figure 1.41) and with loop unrolling (Figure 1.42), using loop unrolling the time taken to complete the simulation, at 5000 iterations, is comparatively faster than without.

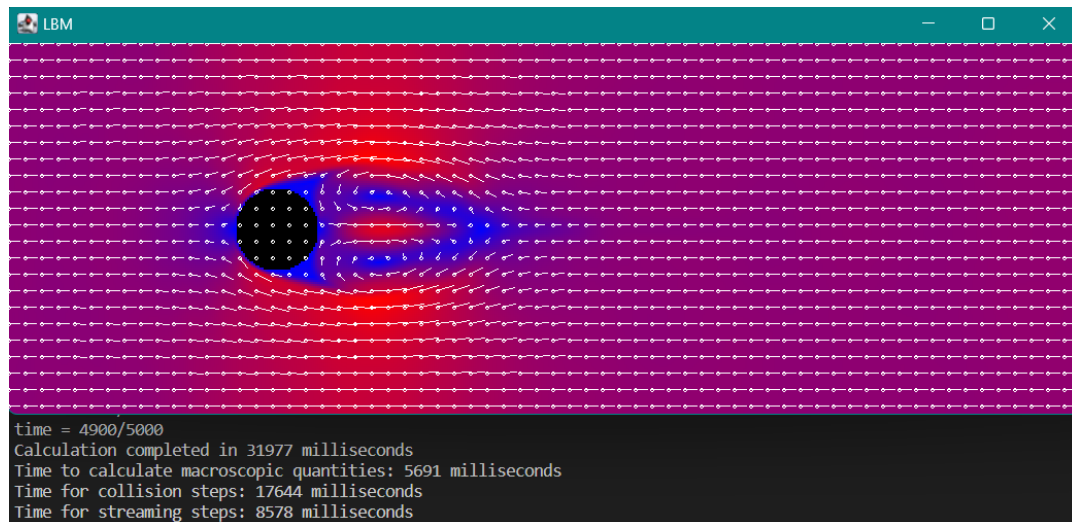


Figure 1.42: LBM - Loop Unrolling

When changing the iteration count to 30,000, you start to notice how the model starts deforming and create clockwise and anti-clockwise vortices, at regular intervals, this is the von Kármán vortex street. Simply this is caused by Vortex Shedding, which is responsible for the unsteady separation of flow of a fluid around blunt bodies ^[1].

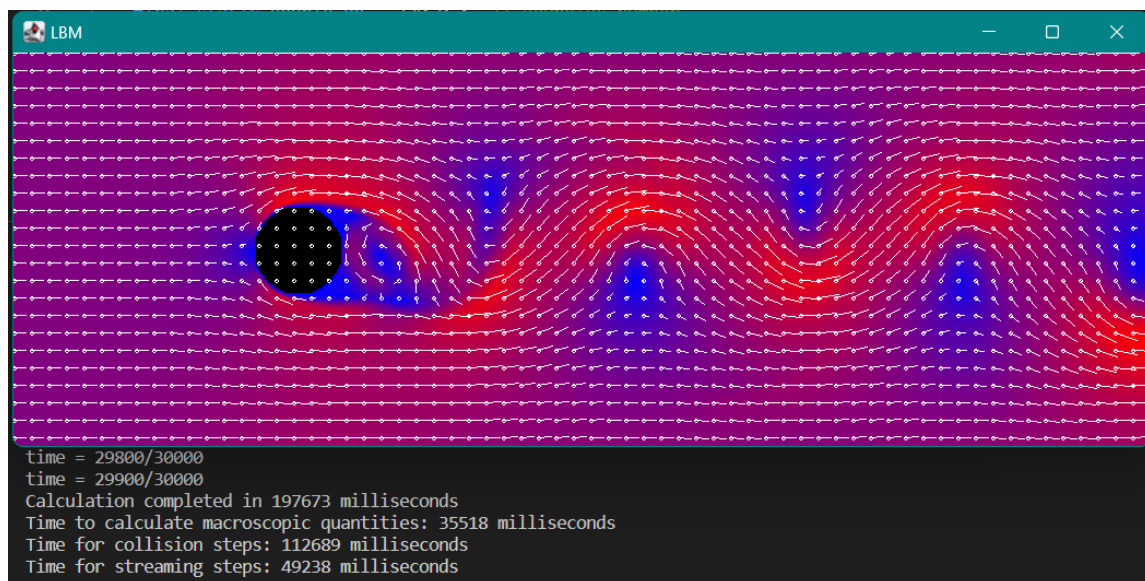


Figure 1.43: von Karman Vortex Sheet

1.8 Lab 8 – An Attempted Application to Aerodynamics

1.8.1 Running the Code – Angle of Attack: 0 Degrees

The image below represents the modified LBM code that was used in the previous lab. This lab modifies the LBM code so that it simulates the lift and drag force on a notional aircraft wing.



Figure 1.44: 0 Degrees Angle of Attack

At each iteration of the simulation, both the average drag force and average lift are calculated; the image below represent the average drag force and life calculated in the final iteration. The reason for the calculation of the negative lift is due conventional use of the y axis that includes the negatives.

```
time = 49900/50000
Average drag = 0.010689937046805415
Average lift = -1.4629376123858301E-5
```

Figure 1.45: Time Taken, Average Drag & Average Lift

1.8.2 Exercise

1.8.2.1 Modifying the Angle of Attack

The previous run of the simulation was done on an angle of attack of 0 degrees; however, the table show how the simulation been executed with three different angles of attack: 3 degrees, 6 degrees and 9 degrees.

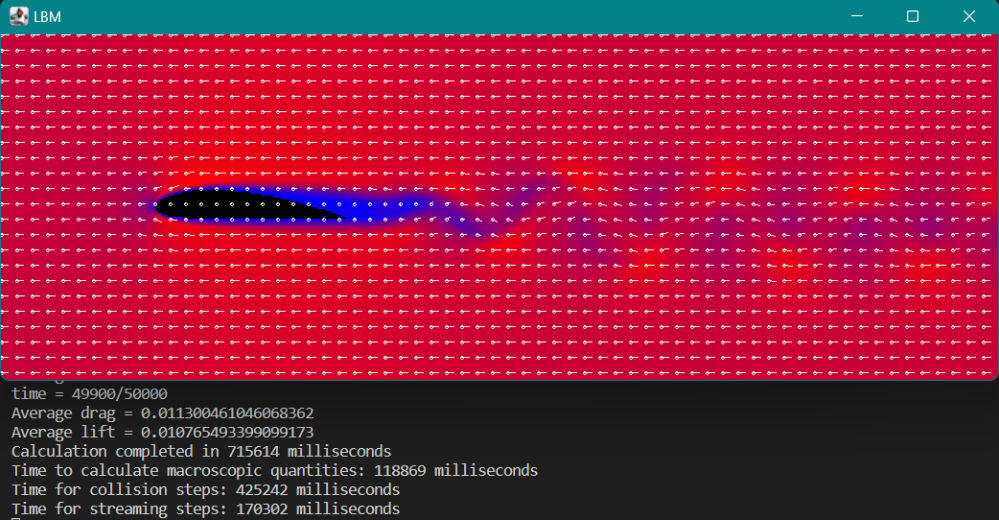
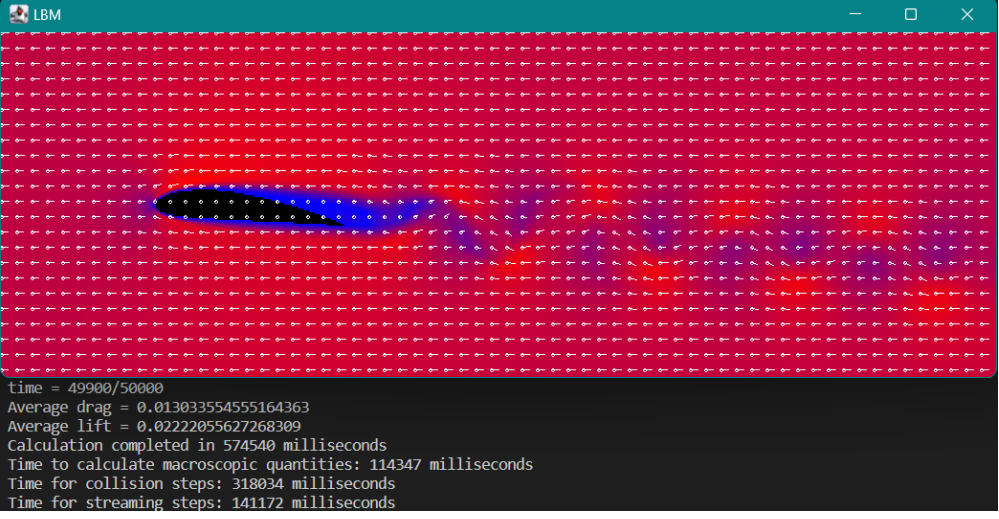
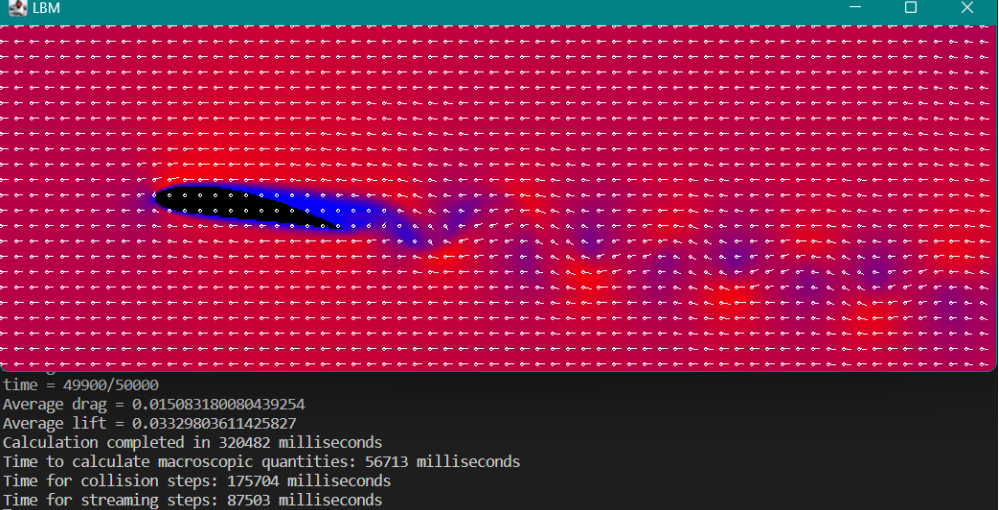
Angle of Attack	Image
3 Degrees	 <pre> time = 49900/50000 Average drag = 0.011300461046068362 Average lift = 0.010765493399099173 Calculation completed in 715614 milliseconds Time to calculate macroscopic quantities: 118869 milliseconds Time for collision steps: 425242 milliseconds Time for streaming steps: 170302 milliseconds </pre>
6 Degrees	 <pre> time = 49900/50000 Average drag = 0.013033554555164363 Average lift = 0.02222055627268309 Calculation completed in 574540 milliseconds Time to calculate macroscopic quantities: 114347 milliseconds Time for collision steps: 318034 milliseconds Time for streaming steps: 141172 milliseconds </pre>
9 Degrees	 <pre> time = 49900/50000 Average drag = 0.015083180080439254 Average lift = 0.03329803611425827 Calculation completed in 320482 milliseconds Time to calculate macroscopic quantities: 56713 milliseconds Time for collision steps: 175704 milliseconds Time for streaming steps: 87503 milliseconds </pre>

Figure 1.46: Different Angles of Attack

1.9 Lab 9 – Solution of Differential Equations

1.9.1 Newton's Equation of Motion

The code that was given in the lab is an example of Newton's equations of motion for motion of two bodies around one another under the force of gravity. The image below is part of the simulation that was run after executing the code. Both the blue and red objects constantly orbit around each other, but they never collide.

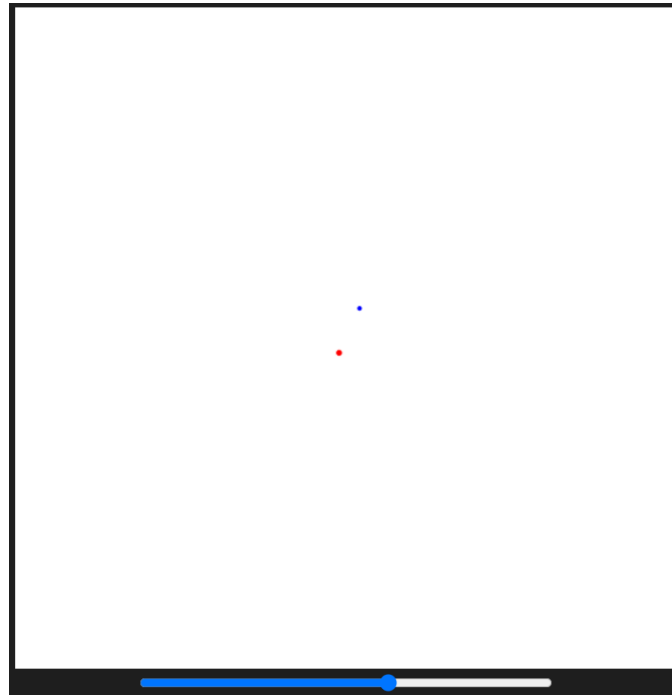


Figure 1.47: Equation of Motion Simulation

1.9.2 Using the Velocity Verlet Algorithm

The code given for this section of the lab is used to simulate the Velocity Verlet algorithm. The code simulates the motion of two particles under the influence of gravity and their corresponding trajectories. The original code that was given in the lab sheet, can be described as a simplified version of the velocity Verlet algorithm as the original would have additional steps that would update the velocity of the objects based on their updated positions whereas here the positions are updated using a basic numerical integration; as shown in the figure below.

```
vx1 += 0.5 * dt * ax1
vy1 += 0.5 * dt * ay1
vx2 += 0.5 * dt * ax2
vy2 += 0.5 * dt * ay2
x1 += dt * vx1
y1 += dt * vy1
x2 += dt * vx2
y2 += dt * vy2
```

Figure 1.48: Basic Numerical Integration for Simplified Velocity Verlet Algorithm

The simulation itself would run in a similar fashion to the simulation in Section 1.9.1.

2 Mini Project

2.1 Simultaneous Algebraic Reconstruction Technique (SART) for Reconstruction of CT Scan Images

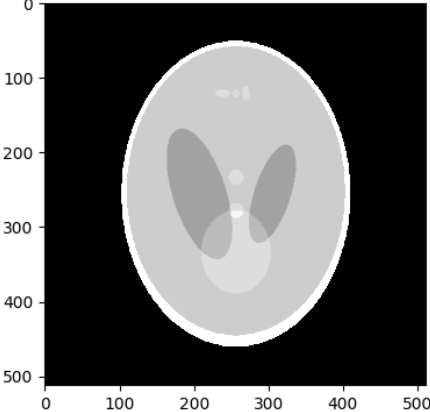
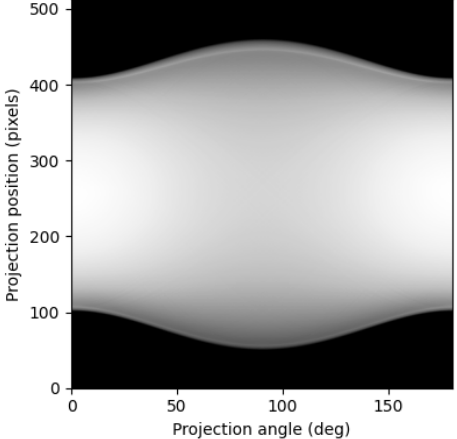
2.1.1 Original Code Execution

The code used for this section can be found on the scikit-image.org website ^[2]. The entire program is a Radon Transform, that includes a forward transform, reconstruction with the Filtered Back Projection (FSB) - an inverse transform using a ramp filter and a reconstruction using SART.

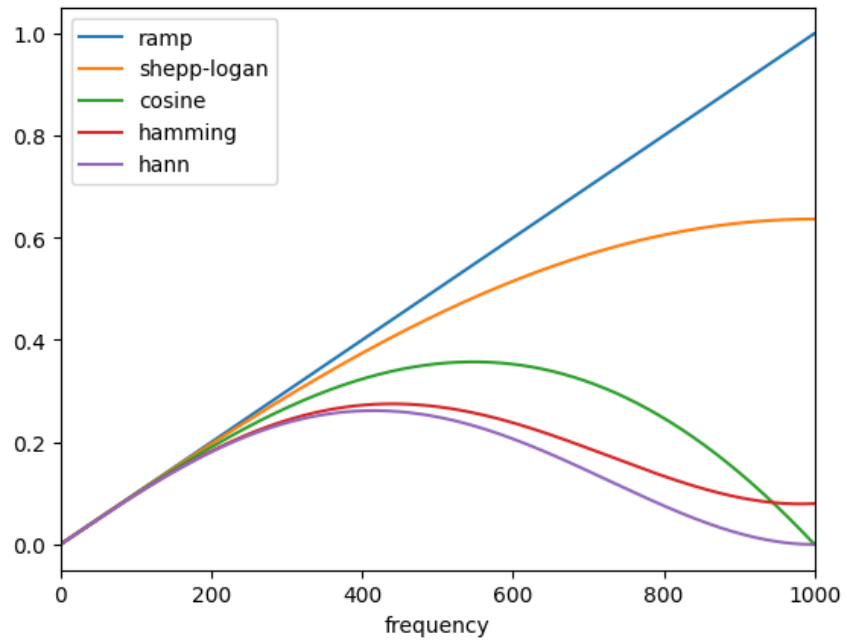
The original code used a Shepp-Logan Phantom from an imported from the `skimage` library, however we were advised to change code so that instead of the library imported image, an image created using the `SheppLoganImage.java` class is used.

The images below show the final output of each of the different sections within the program. The frequency response plot visualises how the filter affects the different frequencies allowing it to show the magnitude response of each filter as a function of the frequency.

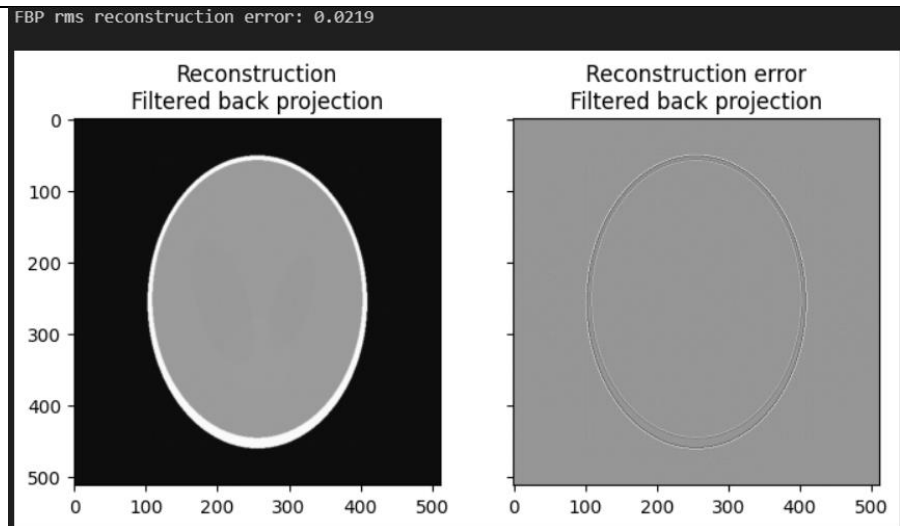
The frequency response plot that was created, shown in the figure below, shows that the 'ramp' filter has a linear frequency response as this would compensate for the attenuation of higher frequencies in the sinogram. Therefore, the choice for the filter used for the inverse transform is the 'ramp' filter.

Program Type	Output
Forward Transform	<div><div><p>Original</p></div><div><p>Radon transform (Sinogram)</p></div></div>

Frequency Response Plot



Reconstruction with FSB - Inverse Transform with 'ramp' filter



Reconstruction using SART

SART (1 iteration) rms reconstruction error: 0.0265
SART (2 iterations) rms reconstruction error: 0.0215

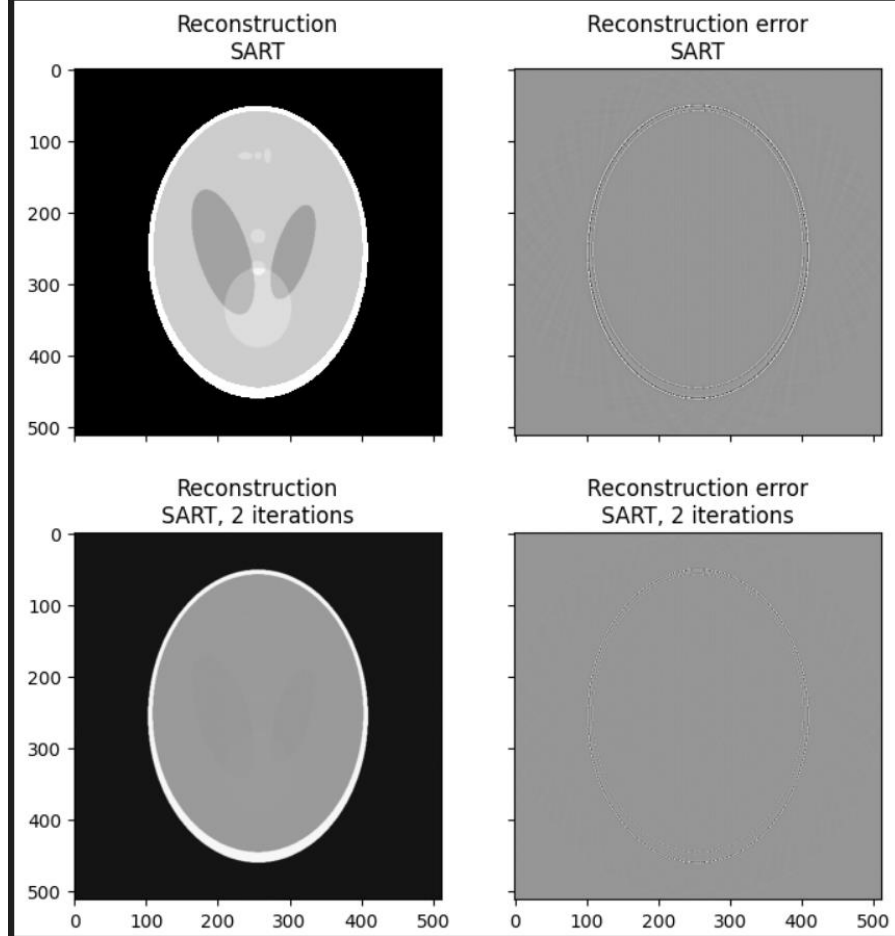


Figure 2.1: Frequency Response Plot & Transform Outputs

2.1.2 Benchmarking

For benchmarking, the code used in Figure 2.2 shows how the time is taken at the start of the reconstruction and then the elapsed time at the end, which is then printed to the user.

Reconstruction Type	Benchmarking Code
'Ramp' Filter	<pre>start_time = int(round(time.time() * 1000)) reconstruction_fbp = iradon(sinogram, theta=theta, filter_name='ramp') elapsed_time = int(round(time.time() * 1000)) - start_time print(f"Time Taken in Milliseconds: {elapsed_time}")</pre>
SART (1 Iteration)	<pre>start_time = int(round(time.time() * 1000)) reconstruction_sart = iradon_sart(sinogram, theta=theta) elapsed_time = int(round(time.time() * 1000)) - start_time print(f"Time Taken in Milliseconds (Iteration 1): {elapsed_time}")</pre>
SART (2 Iteration)	<pre>start_time2 = int(round(time.time() * 1000)) reconstruction_sart2 = iradon_sart(sinogram, theta=theta, image=reconstruction_sart) elapsed_time2 = int(round(time.time() * 1000)) - start_time print(f"Time Taken in Milliseconds (Iteration 2): {elapsed_time2}")</pre>

Figure 2.2: Benchmarking Code

Reconstruction Type	Time Taken (milliseconds/ms)
'Ramp' Filter	2912
SART (1 Iteration)	9217
SART (2 Iteration)	18573

Figure 2.3: Benchmarking Results

2.1.3 More Iterations of SART Reconstruction

Below show how further iterations of the SART reconstruction has reduced the error value, as seen in Figure 2.4. The image of the SART reconstruction can be found in Figure 2.5

```
Time Taken in Milliseconds (Iteration 1): 9733
SART (1 iteration) rms reconstruction error: 0.0265
Time Taken in Milliseconds (Iteration 2): 19194
SART (2 iterations) rms reconstruction error: 0.0215
Time Taken in Milliseconds (Iteration 3): 19194
SART (3 iterations) rms reconstruction error: 0.0193
Time Taken in Milliseconds (Iteration 4): 19194
SART (4 iterations) rms reconstruction error: 0.018
```

Figure 2.4: Benchmark Results & Reconstruction Error Values

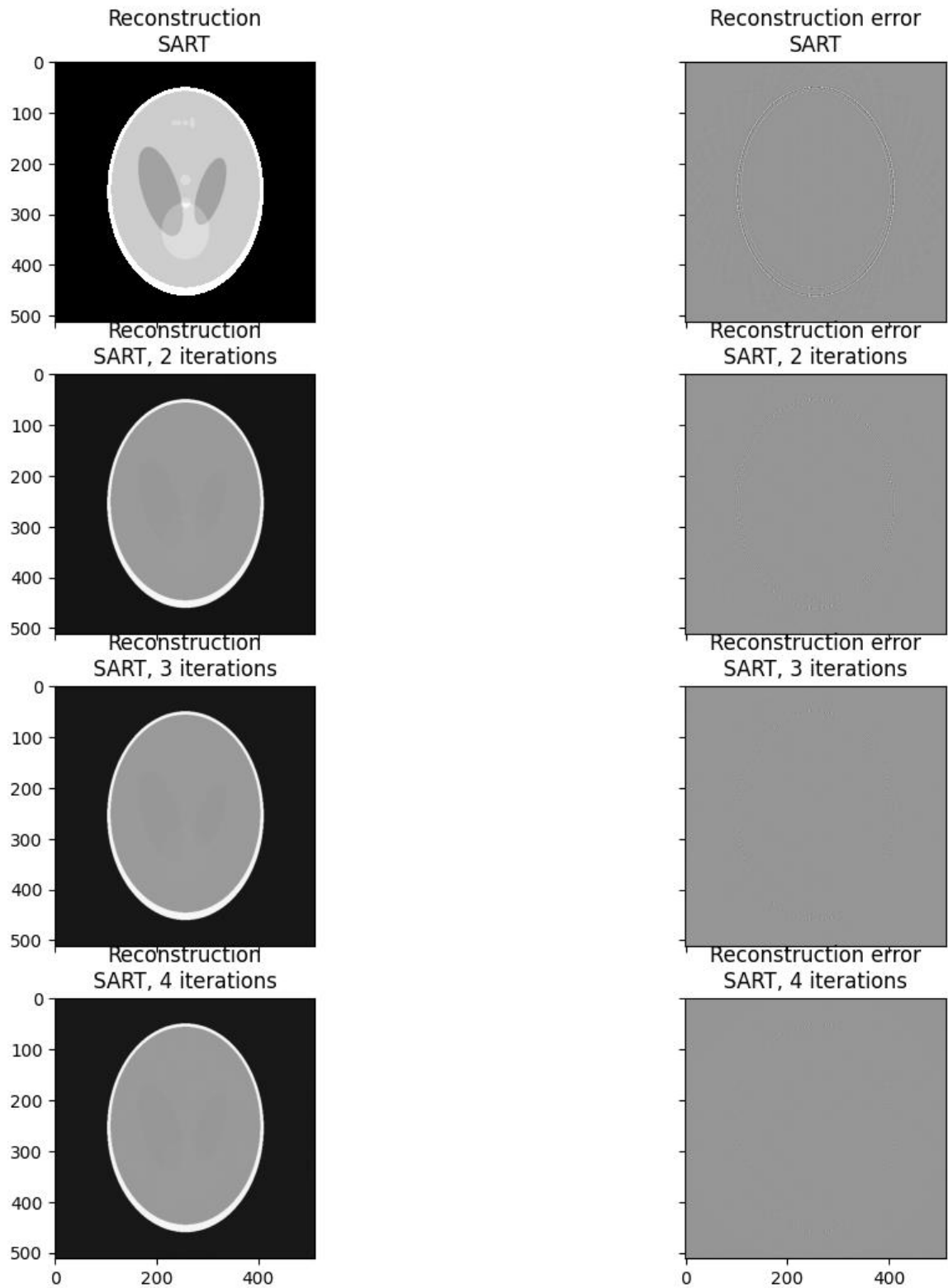


Figure 2.5: Reconstruction SART Images

3 Conclusion

In conclusion, throughout the period of learning of this module, I have come to have a better understanding of the topic area of scientific computing and simulation. During the first three lab sheets, the concept of Fourier transformations was introduced, showing how a simple and naive 2D discrete FT performs on an image to how it can be further improved upon into a fast Fourier transform (FFT), which eventually shows how much faster FFT can be when compared to the naive 2D DFT, they were compared in Figure 1.14. After that, the lab sheet then showed how an inverse FT can be used on a sinogram to retrieve a back projected sinogram.

From lab 4 onwards, each lab contained a different simulation that was provided to us for experimentation. Lab 4 was the most constricted for experimentation as the data required for it was truncated due to upload limits, only allowing for a partially simulated program. Lab 5 onwards, were all complete forms of simulations. Lab 5, allowed for a simulation of a Lattice Gas Models, HHP model and FHP model. Lab 6 allowed for the simulation of excitable media and a more detailed version, the Gerhardt Schuster Tyson (GST) Model, which show the cellular automata for the effects of population movement and vaccination on epidemic propagation. Both labs 7 and 8 used the Lattice Boltzmann Model to simulate the flow of fluid and the lift and drag forces on a notional aircraft wing, respectively. Finally, Lab 9 presented us with the simulation of Newtons Equation of Motion. With each lab simulation experiment, it can be seen how the field of computer simulation can help real world scenarios. Furthermore, this has greatly helped widen my perspective on the field of scientific computing and simulation.

Finally, the mini project chosen was Simultaneous Algebraic Reconstruction Technique (SART) for reconstruction of CT scan images. The program taken from the [scikit.image.org](http://scikit-image.org) website, was a full program about a radon transform on Shepp-Logan Phantom. One of the main points of this program is the ability to do continuous iterations of the SART reconstruction allowing for error values to decrease significantly overtime which would allow for a much more accurate reconstruction of the original image.

4 References

- [1] Wikipedia Contributors. (2020, September 29). *Kármán vortex street*. Wikipedia; Wikimedia Foundation.
https://en.wikipedia.org/wiki/K%C3%A1rm%C3%A1n_vortex_street
- [2] *Radon transform — skimage 0.21.0rc2.dev0 documentation*. (n.d.). Scikit-Image.org. Retrieved May 20, 2023, from https://scikit-image.org/docs/dev/auto_examples/transform/plot_radon_transform.html