SCIENTIFIC COMPUTING AND SIMULATION LAB BOOK

Introduction

This project will discuss about the first 10 week of the labs and will follow some aims of learning for the end of the term. This will also cover a mini project that will be shown at the end. The learning aims that will come from this is to manipulate scientific algorithms to exploit high performance and parallel computational platforms. To be able to assess and evaluate software that already exists for different kinds of scientific computing discussed within the module. Explain mathematical methods and formulae for said scientific methods. Finally, be able to implement algorithms to simulate and visualize selected physical and biological systems and be able to analyze the scientific data. If a figure states it is a gif and is not moving you can find it in the gif file that was sent in with the project,

Contents

Lab 1: Fourier Transforms for Image Filtering

In this lab we focus on coding a naïve two-dimensional Discrete Fourier Transform(DFT) based on a formulae in the lecture and apply some simple filtering on the given image.

Experiment 1.1

Within this experiment the purpose was to demonstrate that DFT could be implemented within a skeleton code provided to us with three classes that will support the program. Within the program there is some code absent **(Figure 1.1.1)** requiring use of three equations **(Figure 1.1.2-3)** that were given within the lecture slides.

Figure 1.1.1 – Absent DFT code

```
for(int m = ...) {
    for(int n = ...) {
        double arg = ... ;
        double cos = ... ;
        double sin = ... ;
        sumRe += cos * X [m] [n] ;
        sumIm += ... ;
    }
}
```

This involves two nested loops that will run through m and n to do sums within the equations over all elements of the input pixels within X being the image.

The first equation is the mathematical formula for calculating the Forward DFT that is stored in a 2D $N * N$ matrix or square matrix **(Figure 1.1.2)**.

Figure 2.1.2 – Forward 2D DFT

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

The second equation is Euler's relation **(Figure 1.1.3)**, which establishes the relationship between the sine and cosine/ trigonometric functions and the complex polar functions. This formula lets the DFT to be expressed in terms of a single set of complex amplitudes.

Figure 4.1.3 – Euler's Relation

$$e^{-ix} = \cos(x) - i\ \sin(x)$$

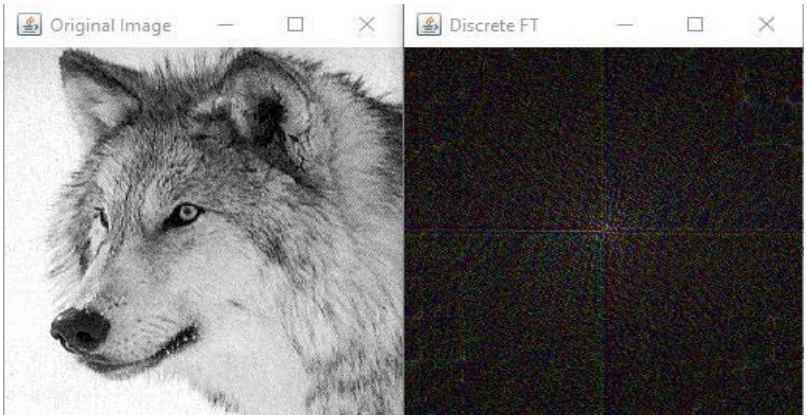
The forward 2D DFT can be further adjusted to use trigonometric functions, while also being able to go through **m** and **n** as loops. This would change $e^{-\frac{2\pi i(km+nl)}{N}}$ into $(\cos(\frac{2\pi(KM+nl)}{N\ 2\pi(km+nl)}) - (isin(\frac{}{N}))$ due to Euler's Relation. This translates to code within **(Figure 1.1.4).**

Figure 3.1.4 – Completed DFT

```
for(int k = 0 ; k < N ; k++) {
    for(int l = 0 ; l < N ; l++) {
        double sumRe = 0, sumIm = 0 ;
        // 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] ;
            }
        }
        CRe [k] [l] = sumRe ;
        CIm [k] [l] = sumIm ;
    }
    System.out.println("Completed FT line " + k + " out of " + N) ;
}
```

When the code is run and compiled there will be two images that will be generated. The original image given into the program that is a picture of a wolf in this instance. The second image that will be generated will be the discreet Fourier transformation of the original image **(Figure 1.1.5).**

Figure 5.1.5 – Original image and DFT



Experiment 1.2

Within this experiment the purpose was to demonstrate the inverse Fourier transform on the program discussed in the previous experiment, so that the image can be reconstructed from the DFT image. Within the program there is code absent **(Figure 1.2.1)** requiring the use of 3 equations **(Figure 1.2.2-3)** that were given within the lecture slides.

Figure 7.2.1 – Absent DFT code

```
double [] [] reconstructed = new double [N] [N] ;

for(int m = 0 ; m < N ; m++) {
    for(int n = 0 ; n < N ; n++) {
        double sum ;
        ... nested for loops performing sum over C elements
        reconstructed [m] [n] = sum ;
    }
    System.out.println("Completed inverse FT line " + m + " out of " + N) ;
}

DisplayDensity display3 =
        new DisplayDensity(reconstructed, N, "Reconstructed Image") ;
```

Figure 6.2.2 – Inverse 2D DFT

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

The first equation in **(Figure 1.2.2)**, which is similar to the equation in **(Figure 1.1.2),** however it shows the Inverse DFT of within a 2d square matrix. This will reconstruct the "original signal" or in this case the image. The output image should be the wolf image in this case. To inverse the process the equation will make the e^{-ix} to e^{ix}.

Figure 8.2.3 – Complex Number Multiplication

$$(a + ib)(c + id) = (ac - bd) + i(ad + bc)$$

The second equation in **(Figure 1.2.3)** is the complex number multiplication, which is used in complex DFT. Complex DFT is a more general version of real DFT and represents data using complex numbers. These can be subdivided into real and complex versions. Complex numbers have a general form of **z=a + bi** where **a** is the real part, **b** is the imaginary part, and I is the square root of -1. The complex conjugate of this number is: **z* = a – bi .** For the addition of complex numbers just add the real and imaginary parts that will lead to **(a + ib) + (c + id) = (a + c) + i(b + d)**. For multiplication you get **(Figure 1.2.3),** which will be using for the inverse 2D DFT adjustment. Using Euler's relation to adjust $e^{\frac{-2\pi i(km+nl)}{N}}$ to into $(\cos(\frac{2\pi(KM+nl)}{N})) + (isin(\frac{2\pi(km+nl)}{N}))$. This making the inverse 2D DFT function using trigonometric functions, while also being able to go through **m** and **n** as loops.

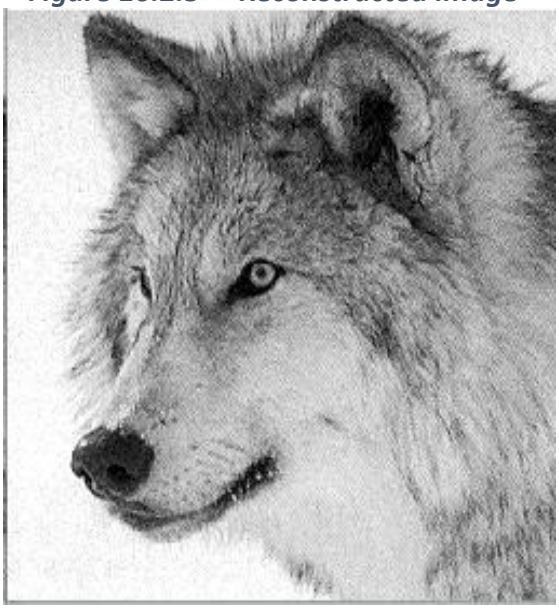
Experiment 1.3

Within this experiment the purpose was to demonstrate that many kinds of filters can be applied to images using the Fourier transformed. The experiment will show a simple kind of filter called the low pass filter where it will simply omit Fourier components with large wave numbers before the reconstruction of the image. The code within **(Figure 1.3.1)** was given in the experiment to see what the output would be like with a low pass filter.

Figure 1.3.1 – Low Pass Filter

```
Thread.sleep(1000);
int cutoff = N/8 ;  // for example
for(int k = 0 ; k < N ; k++) {
    int kSigned = k <= N/2 ? k : k - N ;
    int lSigned = l <= N/2 ? l : l - N ;
    if(Math.abs(kSigned) > cutoff || Math.abs(
        CRe[k][l] = 0 ;
        CIm[k][l] = 0 ;
    }
}

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

Figure 10.2.5 – Reconstructed Image



(Figure 1.3.2). The truncated FT is showing the low frequency that stayed when the filter is passed. As can be seen in the final image after the inverse DFT, it shows that a lot of the image data was not retained, which lead into the image looking blurry.

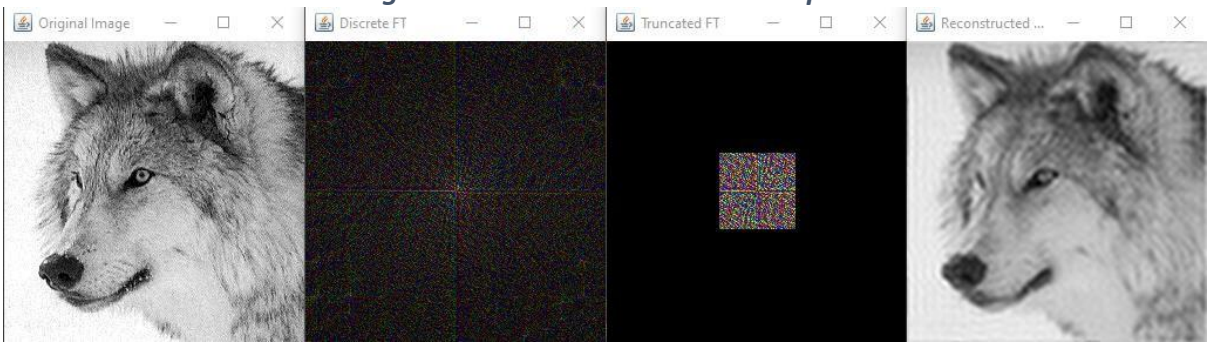
There are two outputs one is a truncated output, and one is the reconstructed output

Figure 9.2.4 – Completed Inverse DFT

```
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; k++){
            for(int l=0; l<N;  l++){
                double arg2 =  2 * Math.PI *((m*k)+(n*l))/N ;
                double cos2 = Math.cos(arg2);
                double sin2 = Math.sin(arg2);
                sum += (cos2 * CRe[k][l]) - (sin2 * CIm[k][l]);
            }
        }
        reconstructed [m] [n] = sum ;
    }
    System.out.println("Completed inverse FT line " + m + " out of " + N) ;
}

DisplayDensity display3 =
        new DisplayDensity(reconstructed, N, "Reconstructed Image") ;
```

The code in **(Figure 1.2.4)** is the completed inverse DFT code, when compiled and run the image that was generated in experiment 1.1 will appear and then process the inverse DFT and present an image similar to the original image show in **(Figure 1.2.5)**.

Figure 1.3.2 – Low Pass Filter Output



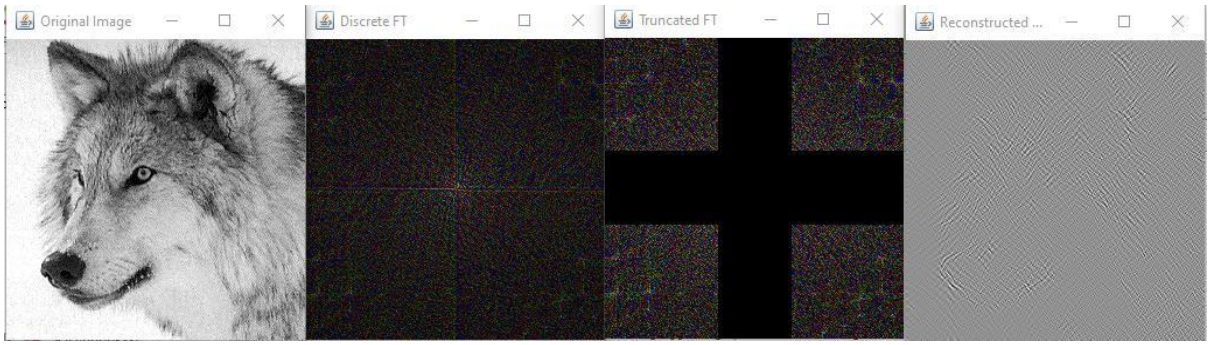
Inverting the if statement condition the filter would be adjusted to a high pass filter **(Figure 1.3.3)**.

Figure 1.3.3 – High Pass Filter

```
Thread.sleep(1000);
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") ;
```

The high pass filter will only retain the higher frequency components and will stop the lower frequency components. This will also lead to two outputs a truncated image, which will show the image after it has passed through the filter, and the output reconstructed. As can be seen in **(Figure 1.3.4)** the truncated image has only retained a small amount of the data as it is filter out the low frequency components leading to the reconstruction being a very faint image of the wolf. It is very hard to see, but there is a faint outline of the wolf within the image.

Figure 1.3.4 – High Pass Filter Output



The main problem with this filter is that it is not very efficient as it takes a very long time for a very small image.

Lab 2: The Fast Fourier Transform

In this lab we focus on looking at the Fast Fourier transform(FFT) and will apply the same image filtering used in experiment 1.1.

Experiment 2.1

Within this experiment the purpose was to implement the FFT into a program. The FFT version is a

Figure 2.1.1 – Absent FFT Code

```
public class FFTImageFiltering {

    public static int N = 256 ;

    public static void main(String [] args) throws Exception {

        double [] [] X = new double [N] [N] ;
        ReadPGM.read(X, "wolf.pgm", N) ;

        DisplayDensity display =
                new DisplayDensity(X, N, "Original Image") ;

        // 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, 1) ;  // Fourier transform

        Display2dFT display2 =
                new Display2dFT(CRe, CIm, N, "Discrete FT") ;

        // 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, "Reconstructed Image") ;
    }

    ... implementation of fft2d ...

}
```

lot more efficient and was provided. The main problem with the previous equation forms the last experiment was the time complexity in terms of computational terms. Due to there being a forward and inverse DFT both having. The program to make it a bit more efficient uses the Cooley-Tukey FFT algorithm. The Cooley-Tukey FFT represses the DFT into of a composite size of $N * N$ square matrixes of smaller DFTs, this is also known as divide and conquer. This is done recursively to reduce the time to $\mathrm{O}(\mathrm{N}\log N)$. The time to compute the transformation, assuming that that the formulae is representing as the function $T(N)$. The complex exponentials, the multiplications and the additions of the formula take the time complexity of $O(N)$ and calculating the two sizes of $N/2$ and the DFT's will takes time of $2T(N/2)$.

Meaning that $T(N) = O(N) + 2T(N/2)$. to simplify we can assume that N a power of 2. The recursion will be $\log_2 N$. This makes the overall complexity $T(N) = O(N \log N)$ **(Figure 2.2.1.2)**. To put it in perspective the Naïve DFT(Lab 1), if it takes 1 nanosecond to perform a single operation, with aa size of $N = 10^9$ it would take 31.2 years where the FFT would take 30 seconds. The first part of the experiment was to implement a static method for the 2dFFT as this will act as the initialisation of a 2d FFT while using code for a 1d FFT **(Figure 2.1.1)**. For the initialisation there will be 2D array for re, being the real number, 2D array for im, which is the imaginary number and sign is a binary value that checks if the transformation will be forward (value is 1) and inverses (value is -1) as can be seen in **(Figure 2.1.2)**.

Figure 2.1.1 – Absent FFT Code

```
static void fft2d(double [] [] re, double [] [] im, int isgn) {
```

The next part was to give the method some structure, by giving it the method for the transposing of

Figure 2.1.3 – Absent FFT Code

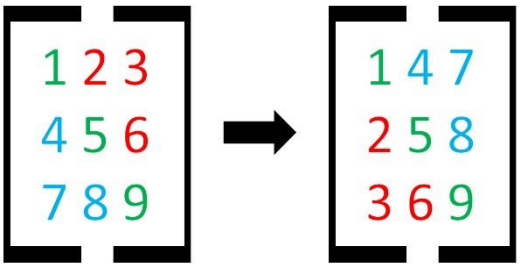
```
static void transpose(double [] [] a) {

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

the FFT **(Figure 2.1.4)**. the reasoning behind transposing the matrix is that the two-dimensional data is operationally a complex matrix and will be easier to run the FFT by rows, when you transpose, you can continue using the 1D FFD as the columns will then become rows allowing the FFT to run all the rows of the matrix then all the columns **(Figure 2.1.5)**.

Figure 2.1.4 – Transpose FFT Code

```
static void transpose(double [] [] a) {
    for(int i = 0 ; i < N ; i++) {
    for(int j = 0 ; j < i ; j++) {
        double temp=a[i][j];
        a[i][j]=a[j][i];
        a[j][i]=temp;
    }
}
```

Figure 2.1.5 – Transpose Matrix



To run through both the row and column it will have to transposes two separate times as it will need to transpose back after running **(Figure 2.1.6)**. This is done by having two separate loops running through the rows **(Figure 2.1.7)**.

Figure 2.1.6 – Absent Run through rows for FFT Code

```
static void fft2d(double [] [] re, double [] [] im, int isgn) {

    // For simplicity, assume square arrays

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

    transpose(re) ;
    transpose(im) ;

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

    transpose(re) ;
    transpose(im) ;
}
```

Figure 2.1.7 – Transpose both rows and columns for FFT Code

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

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

transpose(re) ;
transpose(im) ;
```

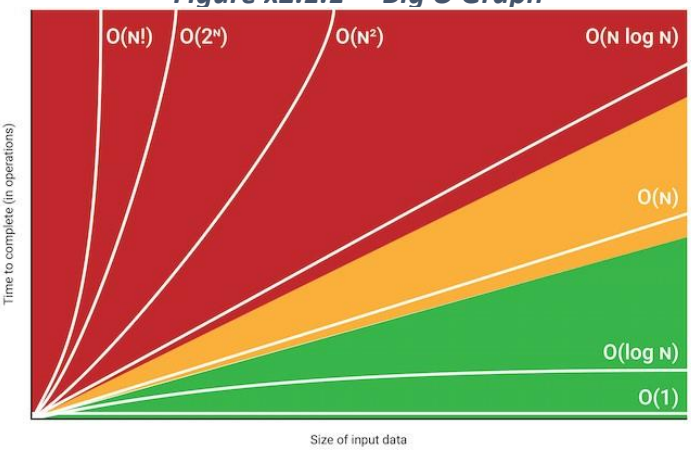
Exercises 2.1

Within this exercise there will be benchmark of the implementation of the 2D Fourier transform against the Naïve code from Lab 1. The naïve code was run without the filters involved. As can be seen in **(Figure 2.2.1.2)** it can be demonstrated well.

Figure x2.1.1 – Table Naïve code vs 2D FFT comparison

Test	Naïve code (milliseconds)	2D FFT (milliseconds)	Speedup
1	185883	358	519
2	191461	371	519
3	185613	364	520

Figure x2.1.1 – Big O Graph



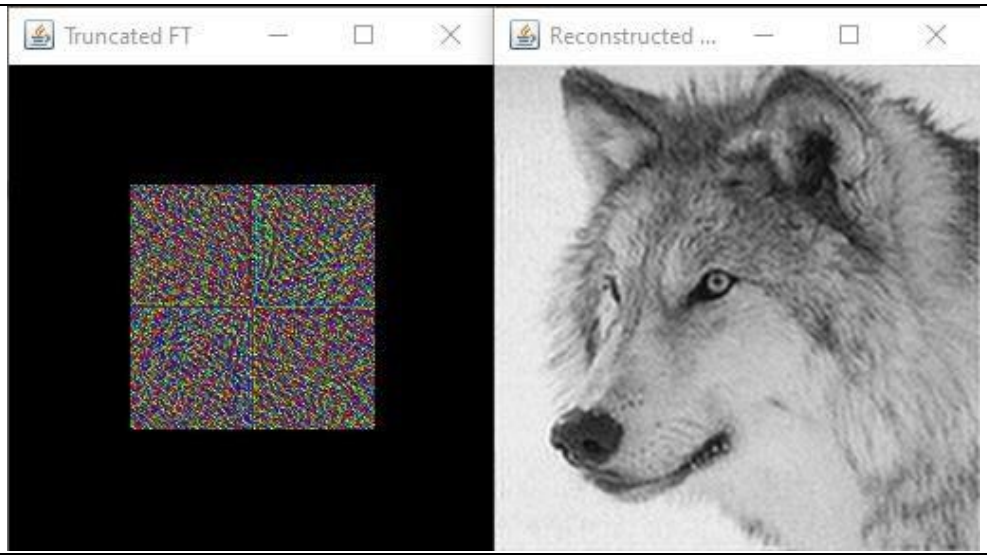
https://danielmiessler.com/study/big-o-notation/

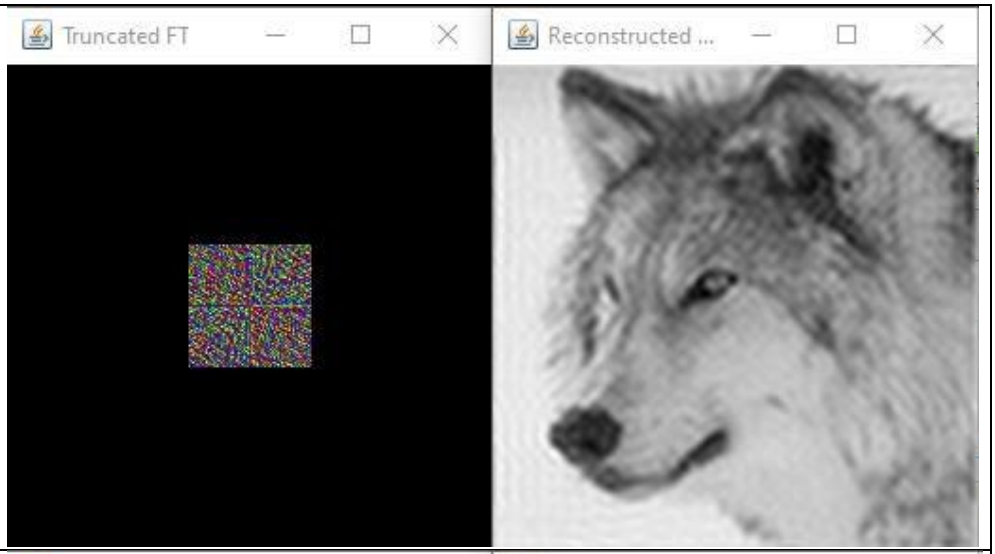
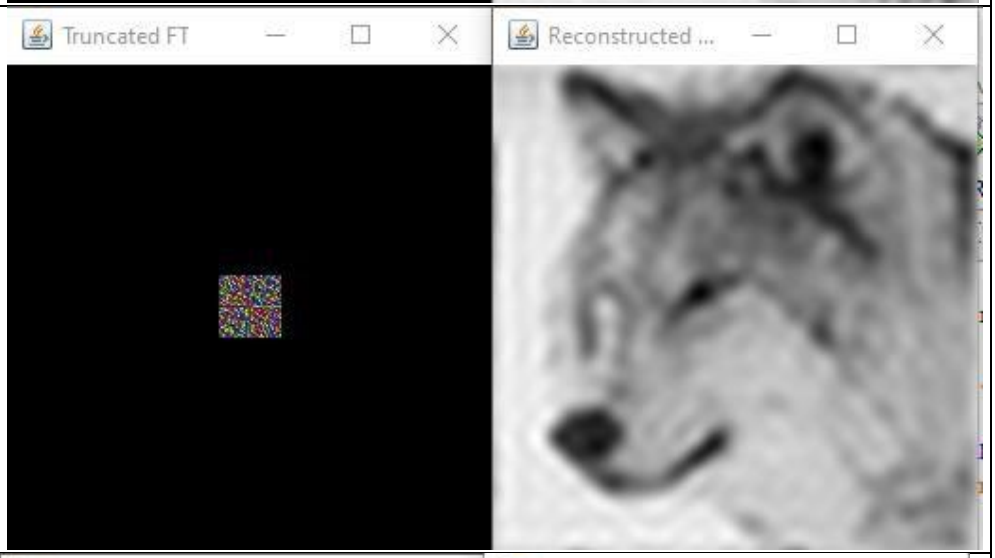
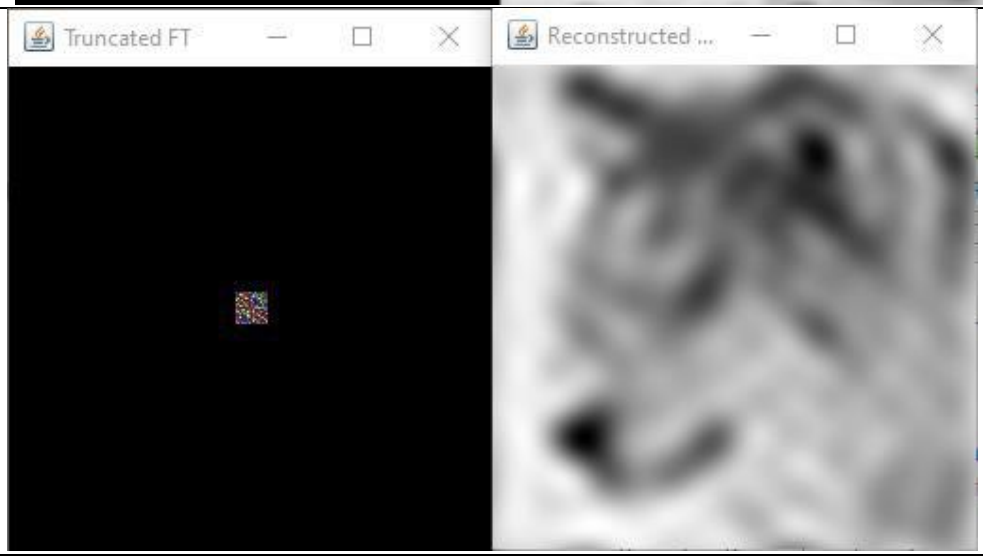
Exercises 2.2

Within this exercise I added the same filters within the lab 1 and seeing what effects and experimenting with different kinds of low pass and high pass filters.

Low pass filter

Figure x2.2.1– Table Low pass Filter Outputs

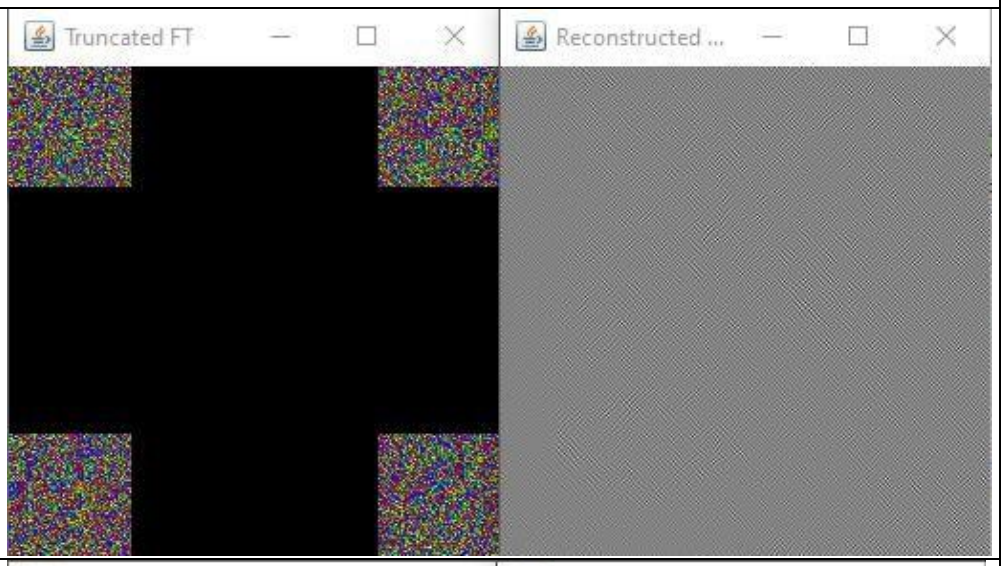
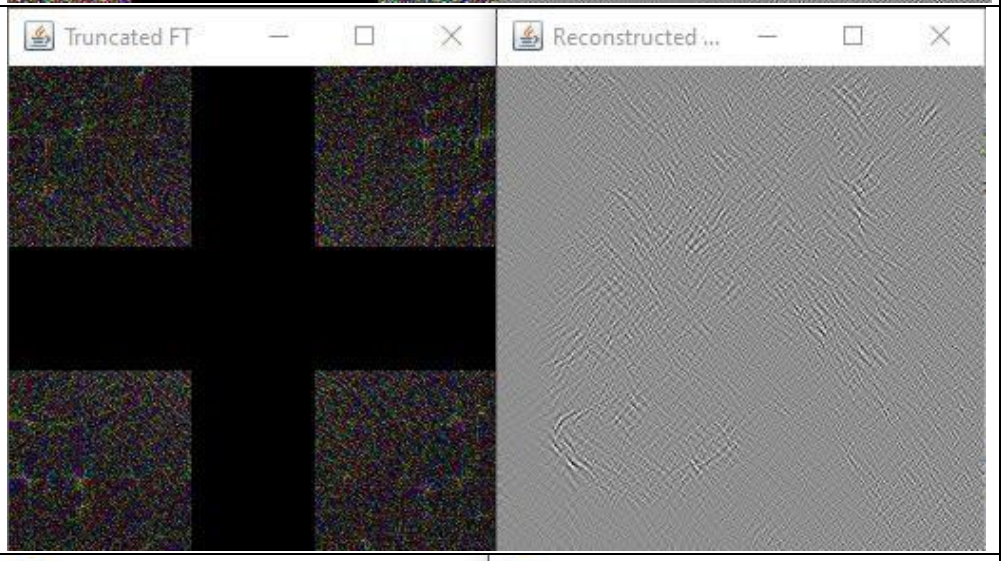
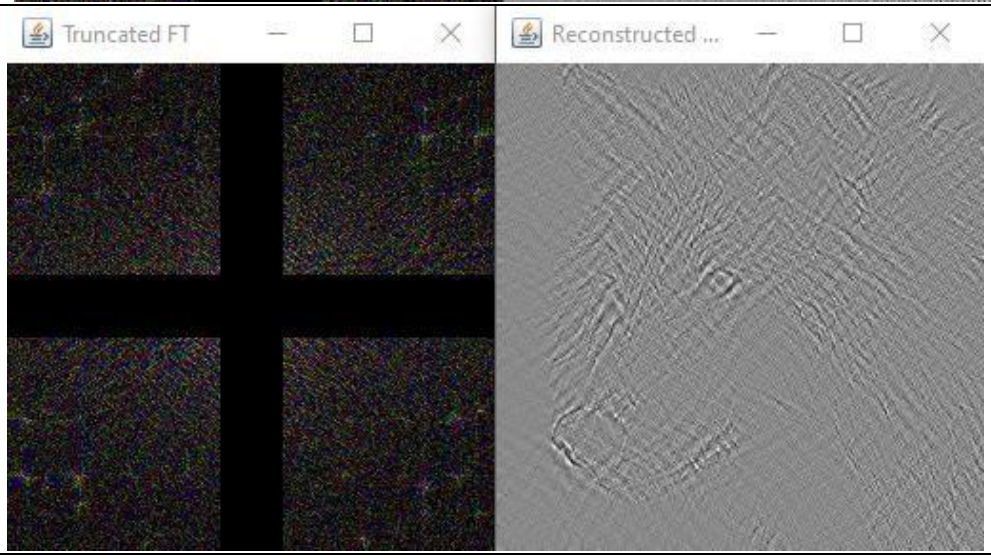
Test	**Cut off**	**output**
1	$N/4$	

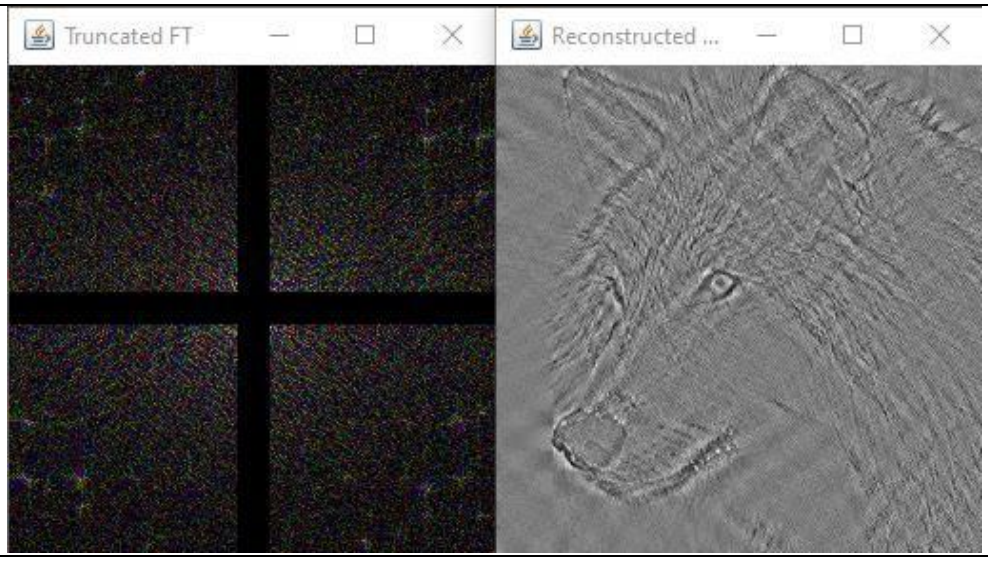
2	$N/8$	
3	$N/16$	
4	$N/32$	

As can be seen in **(Table 2.2.2.1)** as the cut-off divider increases more of the higher frequency components were not retained and filtered out leading to more image data loss as can be seen in the $N/32$. Within $N/32$ the image is a lot blurrier than $N/4$ as only have the image data was retain rather than a 32 .

High pass filter

Figure x2.2.2 – Table High pass Filter Outputs

Test	Cut off	output
1	$N/4$	
2	$N/8$	
3	$N/16$	

4		

As can be seen in **(Table 2.2.2.2)** as the cut-off divider increases more of the Lower frequency components were retained and filtered out leading to less image data lost as can be seen in the $N/32$ compared against $N/4$, as the image of the wolf increasingly gets more visible the higher the cut-off divider is.

Lab 3: Inverting the Radon Transform

In this lab we focus on a simulated density distribution for the interior of a subject's skull and it deriving the radon transform, which will present itself as a sinogram. Main challenge being able to reverse that process and recover the original model from the sinogram.

Figure 3.1.1 – Absent sinogram code

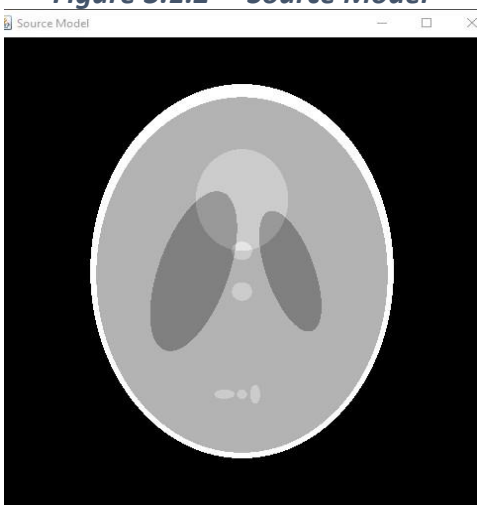
```
static final float GREY_SCALE_LO = 0.95f, GREY_SCALE_HI = 1.05f ;
    // Clipping, for display only.  See for example Figure 1 in:
    //    http://bigwww.epfl.ch/thevenaz/shepplogan/

public static void main(String [] args) {

    double [] [] density = new double [N] [N] ;

    for(int i = 0 ; i < N ; i++) {
        double x = SCALE * (i - N/2) ;
        for(int j = 0 ; j < N ; j++) {
            double y = SCALE * (j - N/2) ;

            density [i] [j] = sheppLoganPhantom(x, y) ;
        }
    }

    DisplayDensity display1 =
            new DisplayDensity(density, N, "Source Model",
                               GREY_SCALE_LO, GREY_SCALE_HI) ;

    // Radon tranform of density (as measured by detectors):

    double [] [] sinogram = new double [N] [N] ;

    for(int iTheta = 0 ; iTheta < N ; iTheta++) {
        double theta = (Math.PI * iTheta) / N ;
        double cos = Math.cos(theta) ;
        double sin = Math.sin(theta) ;
        for(int iR = 0 ; iR < N ; iR++) {
            double r = SCALE * (iR - N/2) ;
            double sum = 0 ;
            for(int iS = 0 ; iS < N ; iS++) {
                double s = SCALE * (iS - N/2) ;
                double x = r * cos + s * sin ;
                double y = r * sin - s * cos ;
                sum += sheppLoganPhantom(x, y) ;
            }
            sinogram [iTheta] [iR] = sum ;
        }
    }

    DisplayDensity display2 = new DisplayDensity(sinogram, N, "Sinogram") ;

    // inferred integral of density points (actually sum of density
    // points, here) for laternormalization of reconstruction

    double normDensity = norm1(sinogram [0]) ;


    // ... Insert sinogram filtering code here! ...


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

    // Normalize reconstruction, to have same sum as inferred for
    // original density

    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,
                               "Back projected sinogram") ;
}
```

Experiment 3.1

Within this experiment was given two new support classes one is the DisplaySinogramFT, which is almost identical to the Display2dFT that was used in the previous labs however it is more tailor-made to work with the Fourier transform displayed only applies in one of two dimensions.

This program is based on CT scans. Primarily a CT scan will create an axial slice through the body. The modern scanners will allow these slices to be stacked and be seen visually in a 3D space. This program will focus on a single slice is reconstructed. When the program is run it will first output the source model, which will act as the slice from the CT scan **(Figure 3.1.2)**.

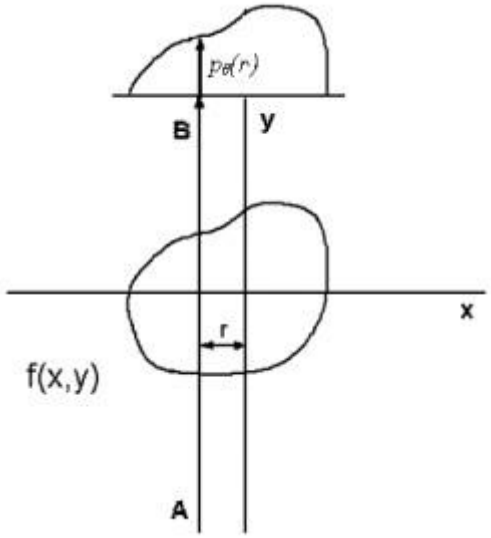
Figure 3.1.2 – Source Model



Then the calculated sinogram will appear. The sinogram is a way of presenting the raw output of a CT scan. It plots the value of the radon transform as a function of r and $\emptyset$ using a grey scale to present the values of $p_{\emptyset}(r)$. Where $p_{\emptyset}(r) =$ Strength of a signal with offset r and angle $\emptyset$. $\emptyset$ is the angle of the assembly and r labels a

particular detectors. The radon transform is the association between the offset r and the angle $\emptyset$ for each beam of the x-ray. The value of $p_{\emptyset}(r)$ can be found from the signal value that is recorded by the detector itself and will be mathematically related to the function $f(x, y)$ by the integral line **(Figure 3.1.3)**. Attenuation of rays a choice of angle for x-ray directions and the perpendicular detector row giving a one-dimensional x-ray slice of a two-dimensional body slice of interest. $Signal\ Strength = \sum^{B}_{x=A} f(x, y)$. $f(x, y)$ is the density of the object. The integral x, y are points on the ray along the line A and B.

Figure 3.1.3 – Attenuation of Rays



Experiment 3.2

Within this experiment was to fill in code for the filtered back projection of the sinogram using the FFTs. In **(Figure 3.2.2)** it is it is only transforming the inner dimension of the 2d array only single calls to the FFT and no need for any transposes **(Figure 3.2.7**

Figure 3.2.1 – Absent Complex Fourier transform

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

Figure 3.2.2 – Complex Fourier transform

```
FFT.fft1d(sinogramFTRe[iTheta], sinogramFTIm[iTheta], 1);
```

Figure 3.2.3 – Absent multiplier sinogram FT

```
for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    for(int iK = 0 ; iK < N ; iK++) {
        int kSigned = iK <= N/2 ? iK : iK - N ;
        ... multiply Sinogram FT by abs(kSigned) ...
    }
}
```

3rd window). The second part the filter is applied to the rows of the Fourier transform **(Figure 3.2.3/4)**. Both he real sinogram and the imaginary sinogram multiplied by the absolute value of kSigned. Without filtering, in this case $|K|$ filtering, on the radon transform

Figure 3.2.4 – Multiplier sinogram FT

```
sinogramFTRe [iTheta] [iK] = sinogramFTRe [iTheta] [iK] * abs(kSigned);
sinogramFTIm [iTheta] [iK] = sinogramFTIm [iTheta] [iK] * abs(kSigned);
```

there will often make the broadly recognizable form of the image will be very blurred. In another loop over iTheta it will invert the FFT on the sinogram to allow this to invert the final argument will need to set to -1

Figure 3.2.4 – Multiplier sinogram FT

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

(Figure 3.2.5). As can be seen in **(Figure 3.2.8)** first window is the output for the filtered sinogram. After this the main image will be presented and with the original main method the line. `backProject(backProjection, sinogram) ;` , which showed the image in **(Figure 3.2.5)**

showing a blurry image, so we changed the line to only using the real ft version as it will make the image a lot sharper, but the internal structure of the brain will be hard to make out **(Figure 3.2.6)**. this is due to using a linear grey scale, so change the display line to allow the larger and better grey

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

output with a lot of noise shown in **(Figure 3.2.8)**.

Figure 3.2.5 – Main back projected sinogram

Figure 3.2.6 – Real and back projected sinogram

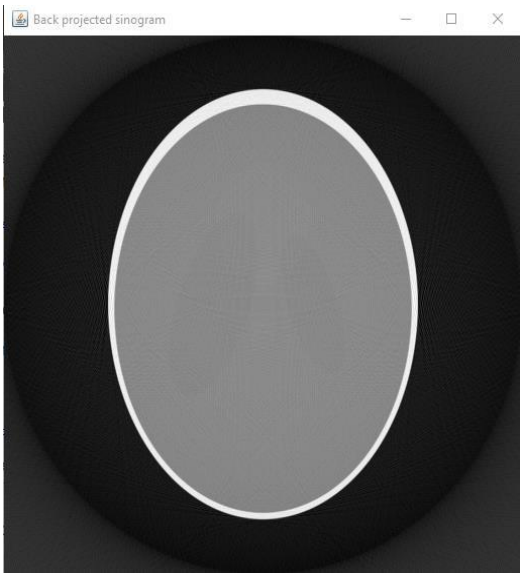


Figure 3.2.7 – Source Model, Sinogram, radial Fourier

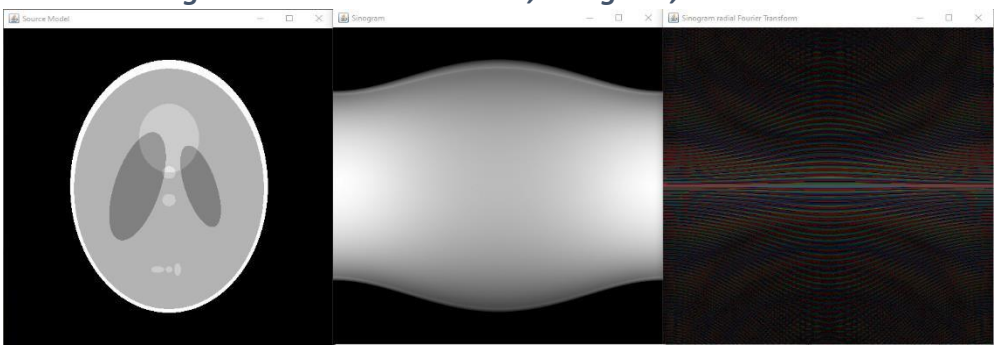
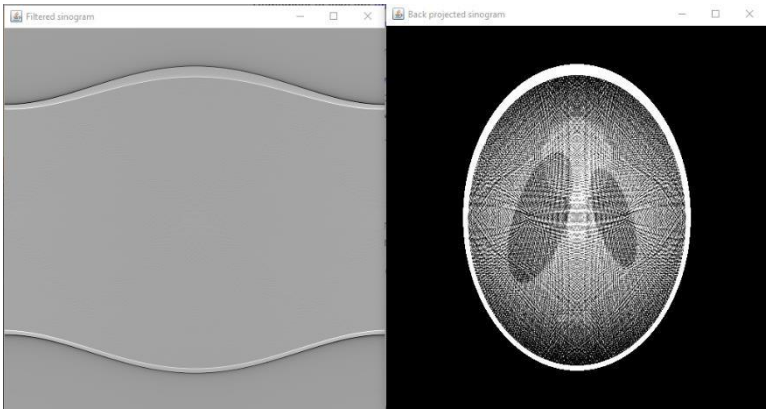


Figure 3.2.8 – Filtered and Back projected Sinogram



scale method. Leading to an

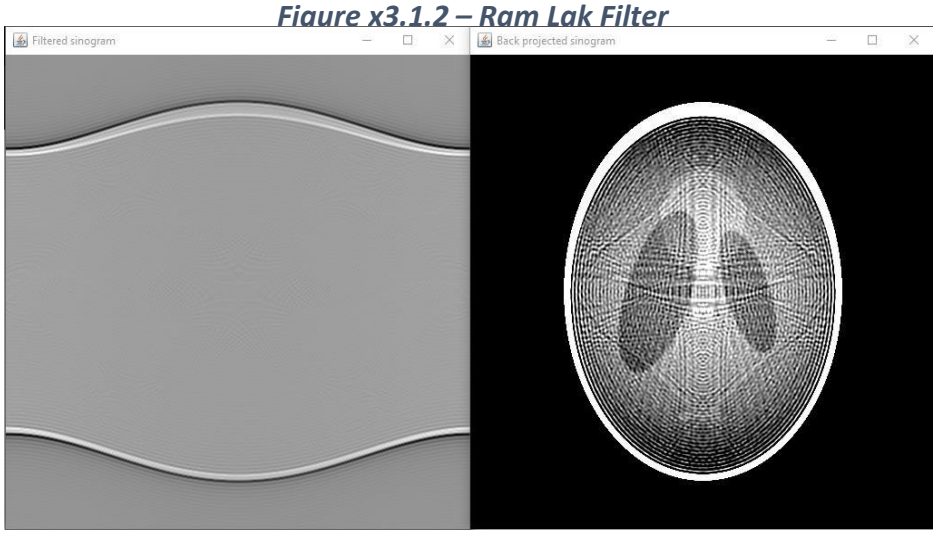
Exercises 3.1

In this exercise will add two filters, one being the Ram Lak filter that is the one, which is used through out experiment 3.1, but when the absolute of K is more than the CUTOFF it will zero the component **(Figure x3.1)**. The output for this filter is shown in **(Figure x3.1.2).** This was set with a cut-off of N/4. As can be seen from the difference between **(Figure x3.1.2)** and **(Figure 3.2.8)** as that the image is a lot less noisy with the Ram Lak Filtering.

```
boolean filter = false;
for(int iTheta = 0 ; iTheta < N ; iTheta++) {
    for(int iK = 0 ; iK < N ; iK++) {
        int kSigned = iK <= N/2 ? iK : iK - N ;
        double multi =1;
        if (filter){multi=cos(PI*iK / (2*CUTOFF));}

        if(abs(kSigned) > CUTOFF){
            sinogramFTRe [iTheta] [iK] = 0;
            sinogramFTIm [iTheta] [iK] = 0;
        }
        else{
        sinogramFTRe [iTheta] [iK] = sinogramFTRe [iTheta] [iK] * (abs(kSigned)*multi);
        sinogramFTIm [iTheta] [iK] = sinogramFTIm [iTheta] [iK] * (abs(kSigned)*multi);
        }
    }
}
```

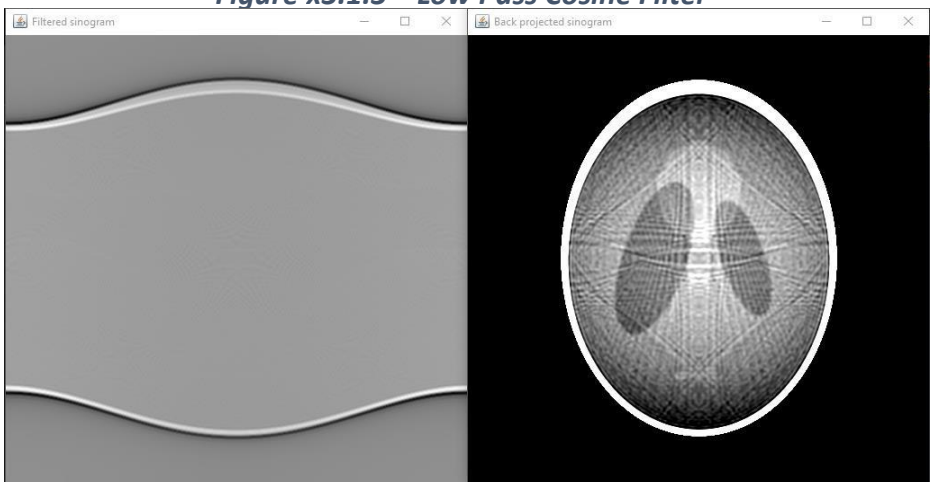
Figure x3.1.1 – Ram Lak and Low Pass Cosine Filters Code

Figure x3.1.2 – Ram Lak Filter



The second filter is Low Pass Cosine Filter where it multiplies the FT by $|K| * \cos(\pi K/(2 * CUTOFF))$ and also sets the components to zero if the $|K|$ is greater than the CUTOFF. With the cut-off also being N/4. As can be seen in **(Figure x3.1.3)** the image is even clearer and less noise than the ram Lak Filter.

Figure x3.1.3 – Low Pass Cosine Filter



Lab 4: An attempt at sky imaging

In this lab we applied principles of imaging from radiointerferometry data the data that is used is real data.

Experiment 4.1

Within this experiment we were provided a main method called imaging that will process the u, v coverage **(Figure 4.1.2)** and will calculate the inverse Fourier transform of some visibility. (u, v) is the two-dimensional displacement between the antennae P1 and P2 **(Figure 4.1.1)**, which will give a vector that is measured in units of the wavelength. Using this you can get the separate vectors. $u = (x_{p1} - x_{p2})/\lambda$ and $v = (y_{p1} - y_{p1})/\lambda$. This is considered the Van Cittert-Zernicke theorem. He then showed there was a correlation or visibility between the antennae, which can be calculated as

$V(u, v) \propto \sum_{\ldots l=\cdots} \sum_{\ldots m=\cdots} \mathrm{I(l, m)} \times \mathrm{e}^{-2\pi\mathrm{i(ul+vm)}}$, where $I(l, m)$ is the sky intensity as the picture envisions an object high in the sky over the antenna on the ground. l, m are angular measures of position in the sky from the centre of a field of view. This is usually referred to a direction cosines or angles in radians. What you can also notice in the equation is $\mathrm{e}^{-2\pi\mathrm{i(ul+vm)}}$, which is the Fourier transform. To recover the Sky Intensity can also give us measurable visibilities as a function for sky intensity. Since the expression for $V(u, v)$ is in the form of a Fourier transform we can invert it as it follows. $I(l, m \propto \sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} V(u, v) \times \mathrm{e}^{2\pi\mathrm{i(ul+vm)}})$. In this case of the program the transform is the sum of u, v for each pixel i, j this corresponding to some pair of direction cosines l, m. Due to the structure of the data that is inputted it suggests dealing with the u, v values in the outer loop and the inner loop i, j accumulate contributions from the u, v pair for each pixel. The data that is used for the program is the LOFAR data set, which has four complex visibilities corresponding to four possible polarizations of the radio waves received. By adding together, the XX and YY components of polarization we can get an overall unpolarized signal, which is computed in reVis and imVis. The array rawImage accumulates the inverse Fourier transform of the visibilities and the dirtyBeam array accumulates the "dirty beam". The dirty Beam comes from a dirty image $I(l, m)$ creates a blurred imaged of the desired target this is due to the u, v plane was a noncomplete coverage **(Figure 4.1.4)**.

Figure 4.1.1 – Demonstration of vectors and antennae positions

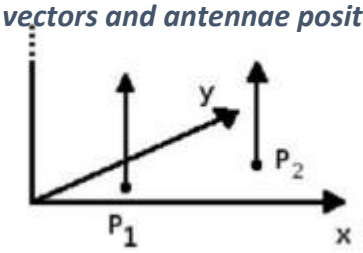


Figure 4.1.2 – u,v Coverage

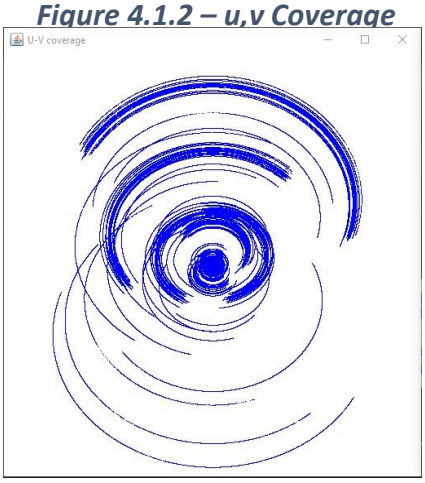


Figure 4.1.3 – Dirty Image

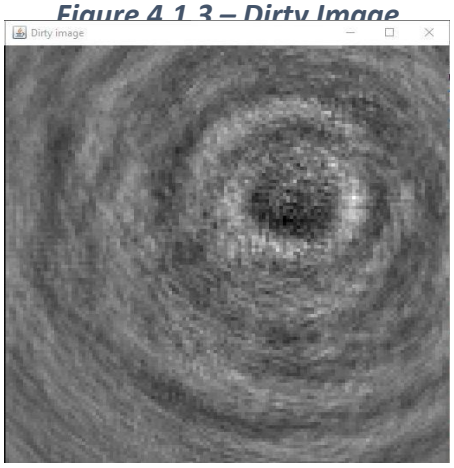
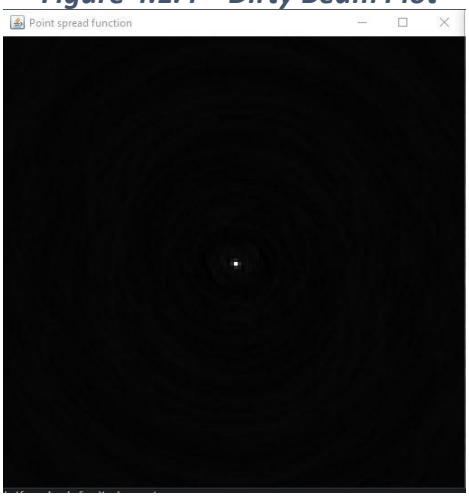


Figure 4.1.4 – Dirty Beam Plot



Lab 5: Latice gas models

In this lab it demonstrates an implementation of the simplest two-dimensional Lattice Gas Model (HPP Model)

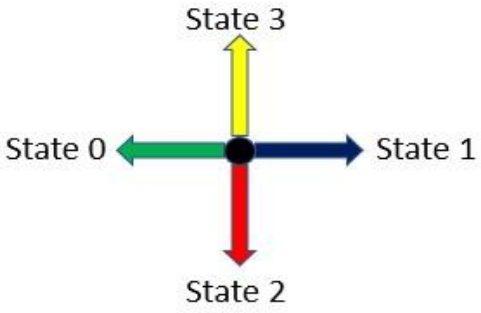
Figure 5.1.1 – Absent collision code

```
public static void main(String args []) throws Exception {

    // initialize - populate a subblock of grid
    for(int i = 0; i < NX/4 ; i++) {
        for(int j = 0; j < NY/4 ; j++) {
            boolean [] fin_ij = fin [i] [j] ;
            for(int d = 0 ; d < q ; d++) {
                if(Math.random() < DENSITY) {
                    fin_ij [d] = true ;
                }
            }
        }
    }

    display.repaint() ;
    Thread.sleep(DELAY) ;

    for(int iter = 0 ; iter < NITER ; iter++) {

        // Collision

        for(int i = 0; i < NX ; i++) {
            for(int j = 0; j < NY ; j++) {
                boolean [] fin_ij = fin [i] [j] ;
                boolean [] fout_ij = fout [i] [j] ;

                // default, no collisions case:

                fout_ij [0] = fin_ij [0] ;
                fout_ij [1] = fin_ij [1] ;
                fout_ij [2] = fin_ij [2] ;
                fout_ij [3] = fin_ij [3] ;

                // please add collisions as per lecture!
            }
        }

        // Streaming

        for(int i = 0; i < NX ; i++) {
            int iP1 = (i + 1) % NX ;
            int iM1 = (i - 1 + NX) % NX ;
            for(int j = 0; j < NY ; j++) {
                int jP1 = (j + 1) % NY ;
                int jM1 = (j - 1 + NY) % NY ;

                // no streaming case:

                fin [i] [j] [0] = fout [i] [j] [0] ;
                fin [i] [j] [1] = fout [i] [j] [1] ;
                fin [i] [j] [2] = fout [i] [j] [2] ;
                fin [i] [j] [3] = fout [i] [j] [3] ;

                // please add streaming as per lecture!
            }
        }

        System.out.println("iter = " + iter) ;
        display.repaint() ;

        Thread.sleep(DELAY) ;
    }
}
```

Experiment 5.1

In this experiment will fill in code for a program that is the hardy, Pomeau and de Pazzis model(HPP), which is defined on a square grid. The HPP model conceptually a grid is habited/ populated by a set of particles with different velocities. The particles can interact when traveling, which in turn will change its velocity into one of the four different states **(Figure 5.1.2)**.

Figure 5.1.2 – States HPP Model

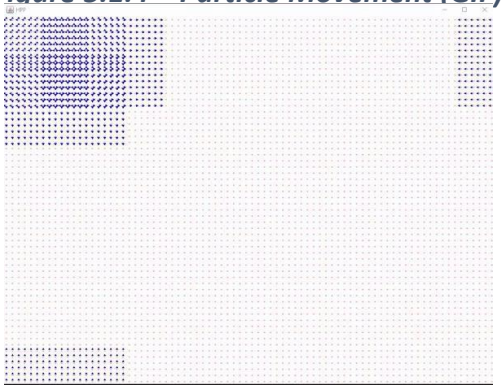


First was the implementation of the streaming, which will decide what states the particles it will go i.e., what change in velocity will it take **(Figure 5.1.3)**. If the state is 0 it will move negatively by 1 in the x direction, 1 it will move positively in the x direction, 2 move negatively in the y direction and 3 will move positively in the y direction. This can be seen in **(Figure 5.1.4)**.

Figure 5.1.3 – Streaming Code

```
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 5.1.4 – Particle Movement (GIF)



Once the behaviour was correct the collision was added **(Figure 5.1.5)** and to be able to show this more accurately, we changed the initialized starting position to only have two particles **(Figure**

5.1.6).

Figure 5.1.5 – Collision Code

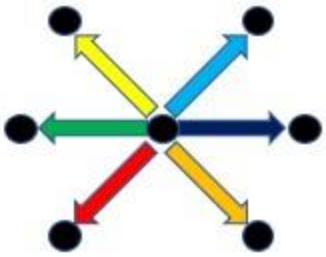
```
if (fin_ij[0] && fin_ij[1] &&
        !fin_ij[2]  && !fin_ij[3]){
    fout_ij [0] = false;
    fout_ij [1] = false;
    fout_ij [2] = true;
    fout_ij [3] = true;
}else if (fin_ij[2] && fin_ij[3] &&
        !fin_ij[0] && !fin_ij[1]){
    fout_ij [0] = true;
    fout_ij [1] = true;
    fout_ij [2]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

Figure 5.1.6 – Collision Gif

Experiment 5.2

In this lab it demonstrates an implementation of the simplest two-dimensional Lattice Gas Model using the FHP model. FHP works very similar to HPP but uses a triangular or hexagonal lattice instead of a square lattice This allowing six points of travel/states rather than four **(Figure 5.2.2)**.

Figure 5.2.1 – States FHP Model



Due to there being six states there will be six local Boolean variables that will describe the states. In this case we used triangle lattice, which is generated from two unit-length basis vectors with x, y components [1,0] and [0.5,0.5 sqrt(3)] respectively. The states 0 and 1 relate to the motion in the negative and positive in the horizontal (b(1)) direction **(Figure 5.2.2)**. The states 2 and 3 correspond to motion in the negative and positive motion in b(2) directions. The final states 4 and 5 move along the yellow and orange line shown in **(Figure 5.2.1)**. To show proof of collision it will be easier to present with a three-way collision as it has more complicated rules than the previous experiment.

Figure 5.2.2 – Basis Vectors

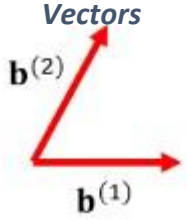


Figure 5.2.3 – Collison FHP Gif

Lab 6: Cellular Automata, Excitable Media, and Cardiac Tissue

In this lab will experiment and adjust a program based on the simple three state cellular automata and the Gerhardt Schuster Tyson(GST) model.

Exercise 6.1

In this exercise the SimpleThreeStateCA code will be run and execute with he plane wave and a spiral wave. This is initiated from the safe part of the code **(Figure x6.1.3)**. Originally the code

Figure x6.1.1 –Plane Wave Gif

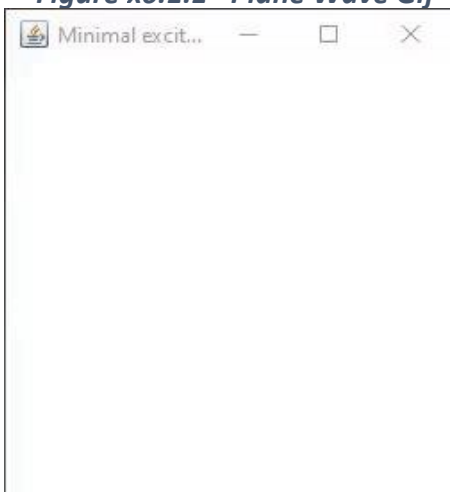
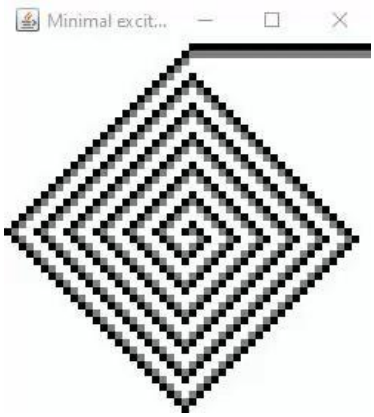


Figure x6.1.2 –Spiral Wave Gif

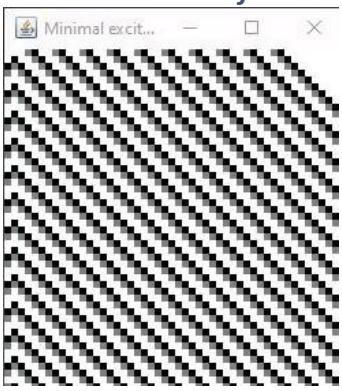
Figure x6.1.3 –Code to comment out

```
if (iter == N / 2) {
    for (int i = 0; i < N / 2; i++) {
        for (int j = 0; j < N; j++) {
            state[1][1] = 0;
        }
    }
}
```



iteration is half of N and the i variable that relates to the x value meaning if also the i is smaller than N/2 the spiral will

Figure x6.2.2 – Initial from corner Gif



Fiaure x6.2.3 – Initial from corner Code

```
if (iter == 2) {
    for (int i = 0; i < 1; i++) {
        for (int j = 0; j < N; j++) {
            state[i][j] = 0;
        }
    }

}
```

will spiral from the centre this was due to the iteration set to N/2 which will do a check at the middle of the screen when the wave goes up it will then set the state to 0, which will start to update the state **(Figure x6.1.2)**. To stop the spiral from happening you just comment out the loop as this will stop any position to set to state 0 **(Figure x6.1.1)**.

Exercise 6.2

In this exercise modified the code to act different where the wave is chopped, and the spiral has started. To set the spiral in the middle the ***Figure x6.2.1 –Initial from centre Gif***

start in the centre of the window **(Figure x6.2.1)**. If you set the iteration to two and the i is the less than one **(Figure x6.2.3)**, it will start the spiral will start in the bottom left **(Figure x6.2.2)**.

Exercise 6.3

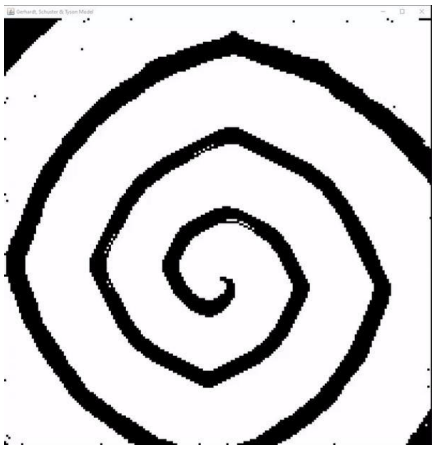
In this exercise the GerhardtSchusterTyson code will be run and execute with a plane wave and a spiral wave. This is initiated in the similar place as the SimpleThreeStateCA; however, it uses the values u and v, which are used for the waves instead of simpler square excited areas **(Figure x6.3.3)**. Originally the code will spiral from the centre **(Figure x6.3.2)** this can be achieved by the initial if state the state of u when the values within u are set to N/2 and the value of i is smaller than N/2 it will start the spiral. If this code **(Figure x6.3.3)** is commented out, it would lead to a plane wave **(Figure x6.3.1)**.

Figure x6.3.1 – Plane Wave Gif

Figure x6.3.3 – Chop wave

```
if (!chopped && u[N/2][N/2] == 1) {
    chopped = true;
    for (int i = 0; i < N / 2; i++) {
        for (int j = 0; j < N; j++) {
            u[i][j] = 0;
            v[i][j] = 0;
        }
    }
}
```

Figure x6.3.2 – Spiral Wave Gif

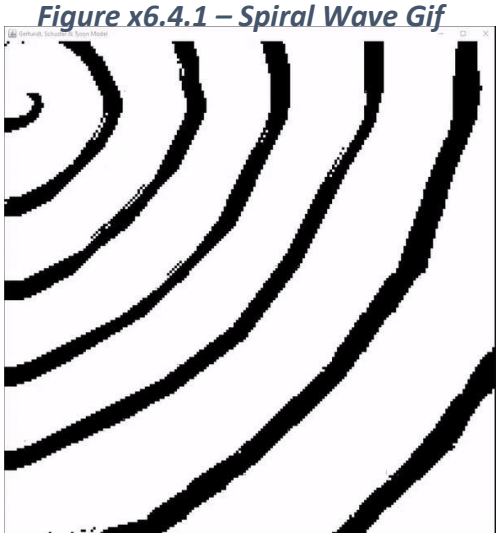


Exercise 6.4

in this exercise it was to initialise the stimulation at the

corner of the screen **(Figure x6.4.1)**. This is done by setting the values in u to 0 and giving only loop the values in i as they are less than 3 **(Figure x6.4.2)**.

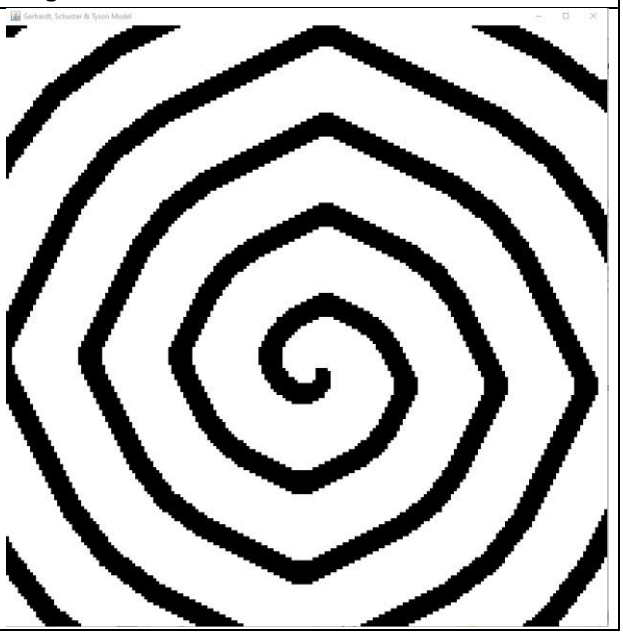
Figure x6.4.1 – Spiral Wave Gif

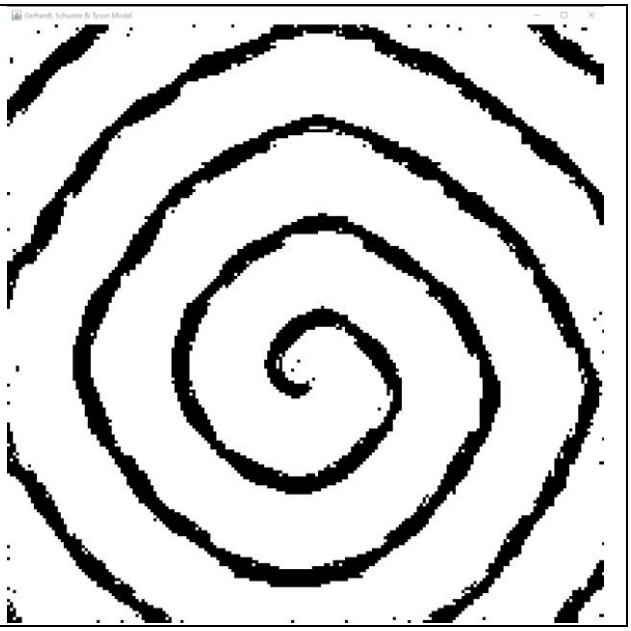
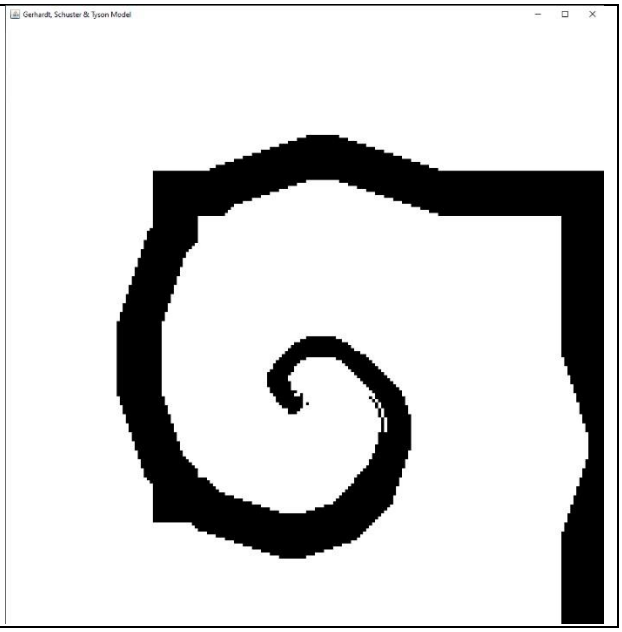


```
if (!chopped && u[0][0] == 1) {
    chopped = true;
    for (int i = 0; i < 3 ; i++) { //
        for (int j = 0; j < N; j++) {
            u[i][j] = 0;
            v[i][j] = 0;
        }
    }
}
```

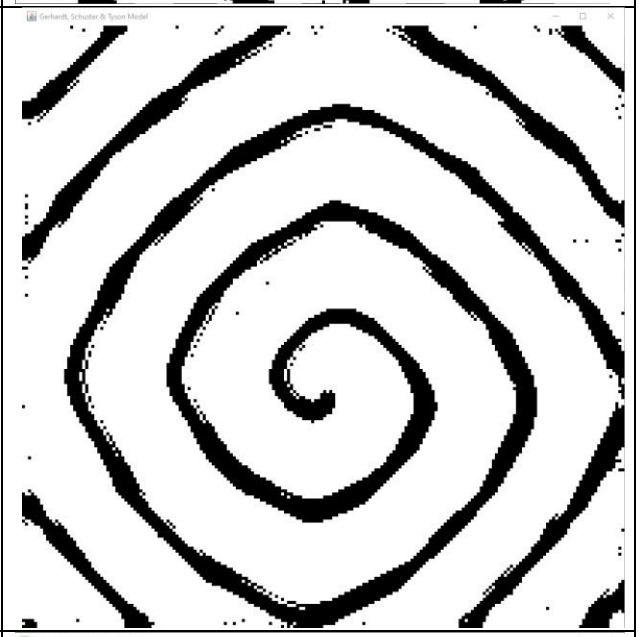
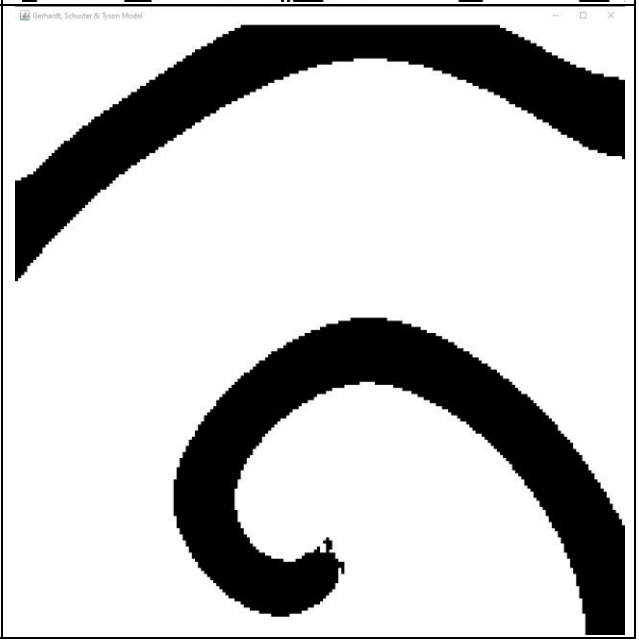
Figure x6.4.2 – Chop wave code

Exercise 6.5

Variable	Decryption	Change	Image
V_MAX	Maximum possible value for v	Decreasing the max possible value of v decreases the noise that is caused on the waves and made the wave thicker and consistent.	

V_RECO	In order to a cell jumps to the recovery state, V_RECO <	Decreasing this number causing more random cells to stay excited for longer and makes it a lot noisier. If increased, it smooths out the wave and less random cells stay excited.	
V_EXCI	To get a cell excited, v <= V_EXCI	Decreasing this value causes the wave to be a lot squarer. Due to it being easier getting the more cells excited. It also makes the wave thicker. When increased the wave is thinner and is more circular.	

G_UP	Increases the value of V when the cell is excited	Decreasing this value also decreases the value v when the cell is excited this causes a shark teeth like affect on the wave and also leaves a lot of excited cells randomly. When increased the wave becomes a lot more circular and there are less random cells excited and the wave is smooth.	
G_DOWN	Decreases the value of V when the cell is deexcited.	Increasing this value decreases the value of v when the cell is de excited this leads to the wave becoming thicker in the spiral. Decreasing this value will make the centre of the spiral move around the window.	

K0_EXCI	Minimum number of excited neighbours to make a cell excited	Increasing this number from 0 causing the waves increasing the frequency of the waves due to move excited cells and makes the waves less consistent in shape.	
K0_RECO	Minimum number of unexcited neighbours to make a cell jumps to recovery state	Increasing the value will lead to more recovery state cells meaning the frequency of the waves are increased and also a lot more random cells will be excited.	
R	Neighbourhood radius	Increasing the radius will increase the size of the wave and the plane wave as the neighbourhood would be increased whereas decreasing will do the opposite.	

Exercise 6.6

In this exercise we developed the simple three state CA to become a Four state CA. The four states will be 0 that will represent rest, 3 that will represent excited wave front, 2 represents excited that is plateau and 1 represents recovering wave back. Also need to introduce a time to state change that will represent the number of time-steps until the cells state variable should be decremented to the next state i.e., a timer till it moves down a state **(Figure x6.6.1)**. This leads to a lot of changes within the main program. The first change is that the state variable will need to increase to be from 0 to 3 **(Figure x6.6.2)**. This is for the initial state definition for the way that will sweep across the screen on start up. Where the wave gets chopped the timeToStateChange need to be set to 0, so it can also start working alongside the state variable **(Figure x6.6.3)**. After this the excited neighbour needs to be adjusted due to there being 8-cells that can be excited there will need to set the excited neighbours variable to work with all 8 positions around the cell. It also needs to make sure it works with both 3 and 2 as they are both states that mean the cell is excited **(Figure x6.6.4)**. When the state needs to be updated the timeToStateChange needs to be 0 as it means that state is not already excited. When the state is not 0 the timeToStateChange will need to decrease by one on

Figure x6.6.1–Time to state change

```
static int[][] timeToStateChange = new int[N][N];
```

Fiaure x6.6.2–initial state definition

```
state[i][j] = j == N - 1 ? 3 : 0;
```

Figure x6.6.3–Chop Wave

```
// Chop wave when half-way up.
if (iter == N/2) {
    for (int i = 0; i < N/2; i++) {
        for (int j = 0; j < N; j++) {
            state[i][j] = 0;
            timeToStateChange[i][j]=0;
        }
    }

}
```

Figure x6.6.4–Excited Neighbour

```
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (timeToStateChange[i][j]==0){
            switch (state[i][j]) {
                case 0:
                    if (excitedNeighbour[i][j]) {
                        state[i][j] = 3;
                        timeToStateChange[i][j] = 2;
                    }
                    break;
                case 3:
                    state[i][j] = 2;
                    timeToStateChange[i][j]=3;
                    break;
                case 2:
                    state[i][j] = 1;
                    timeToStateChange[i][j]=2;
                    break;
                default: // 1
                    state[i][j] = 0;
                    timeToStateChange[i][j]=0;
                    break;
            }
    }else{
    timeToStateChange[i][j]-=1;
    }
   }
}
```

```
excitedNeighbour[i][j]
        = state[i][jp] == 2
        || state[i][jm] == 2
        || state[ip][j] == 2
        || state[im][j] == 2
        || state[im][jp] == 2
        || state[im][jm] == 2
        || state[ip][jp] == 2
        || state[ip][jm] == 2
        || state[i][jp] == 3
        || state[i][jm] == 3
        || state[ip][j] == 3
        || state[im][j] == 3
        || state[im][jp] == 3
        || state[im][jm] == 3
        || state[ip][jp] == 3
        || state[ip][jm] == 3
        ;
```

each iteration. A new case was added, which is case 3, that will set the state to 2 and the time to state

change to 3

Figure x6.6.5–Update

(Figure x6.6.5).

To make it easier to watch I changed the third state to show red to show the initial excited wave **(Figure x6.6.6 and Figure x6.6.7)**.

Figure x6.6.6–Colour Update

```
if (state[i][j] > 0) {
    switch (state[i][j]) {
        case 2:
            g.setColor(Color.BLACK);
            break;
        case 3:
            g.setColor(Color.RED);
            break;
        default:
            g.setColor(Color.GRAY);
            break;
    }
```

Figure x6.6.7–Colour Update Gif

Lab 7: A Lattice Boltzmann Model

In this lab will experiment with a program that works with the Lattice Boltzmann code, which will simulate a two-dimensional fluid flow around a cylindrical obstacle.

Experiment 7.1

In this experiment we will be running a sample lattice Boltzmann code, which is based on the Boltzmann equation. Boltzmann showed the continuum function f would obey a partial differential equation: $\frac{\partial f}{\partial t} + v \times \nabla f + g \times \frac{\partial f}{\partial v} = \Omega(\mathrm{f})$ where ∇f is the vector $\left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial t}\right)$, is the vector $\left(\frac{\partial f}{\partial v_x}, \frac{\partial f}{\partial v_y}\right)$, g is the acceleration due to gravity and $\Omega(\mathrm{f})$ is the collision operator.

When the code is run it will run through 5000 iterations, which will display the velocity of the flow where red is fast flow and blue means slow (including fluid at rest). To show direction of the flow a coarser grid is superimposed display tags that will change in length and direction depending on the flow vector. The initial state of the fluid is in equilibrium at rest, which will start flowing in from the left of the window moving to the right. There will be a noticeable pattern of initial and steady flow where there is a symmetric pair of vortices attached that is immediately downstream of the obstacle i.e., flow going in circles that is on top of each other. These circles of flow will eventually become ovals once the simulation finishes at 5000 iterations **(Figure 7.1.1)**. Once is occurred it was time to optimise the program by using loop unrolling. It consists of taking a performance critical inner loop of a program and either partially replacing the loop with a sequence of operations or completely replace the loop and repeating the body of the loop in a constant number of times, in turn reducing overheads of branching looping. In this program it is effective due to the d loops only repeating 9 times overall and its particular to unroll the whole of these loops. If the program can be specialized to a particular set of velocity states, we can replace c and w elements with their known values which will simplify

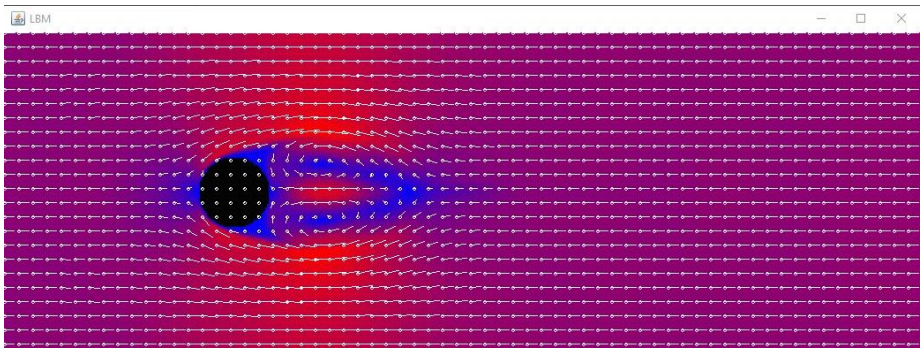


Figure 7.1.1 – 5000 iteration Lattice

the program. As can be seen below when it is unrolled it is faster with a speedup of 1.13. Niter = 5000

Unrolled - Total 80312 milliseconds

Calculate macroscopic: 9704 milliseconds

Collision steps: 34859 milliseconds

Streaming: 35709 milliseconds

Rolled – Total 90456 milliseconds

Calculate macroscopic: 19210 milliseconds

Collision steps: 27659 milliseconds

Streaming: 42557 milliseconds

The macroscopic time to calculated and streaming increased, but collision steps seem to increase.

Running the program to iteration 30,000.

After 5000 steps the two vortices (circular flow) behind the obstacle will slowly deform **(Figure 7.1.2)**. At 15000 steps the flow pattern will start to destabilise, and the obstacles will start to shed and deform the vortices that are sitting in its wake **(Figure 7.1.3)**. After a while it will disappear downstream and a new order will be made with two new vortices that are alternating between clockwise and anticlockwise, which will shed at regular intervals **(Figure 7.1.4)**.

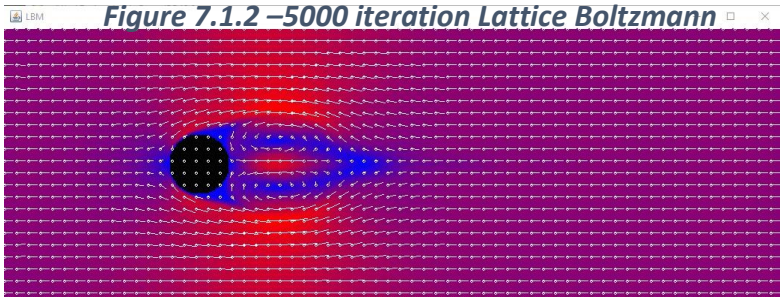


Figure 7.1.2 –5000 iteration Lattice Boltzmann

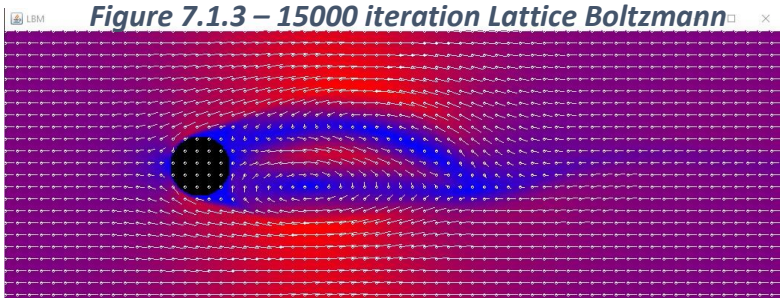


Figure 7.1.3 – 15000 iteration Lattice Boltzmann

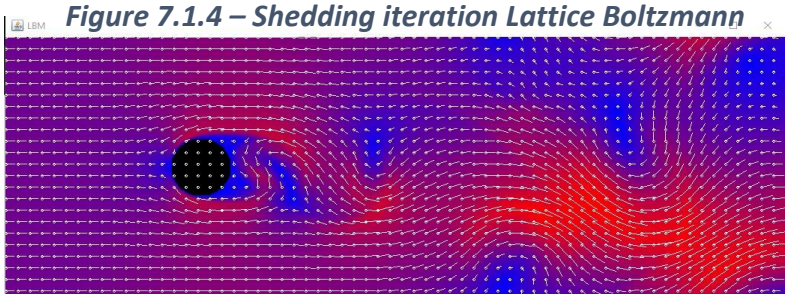
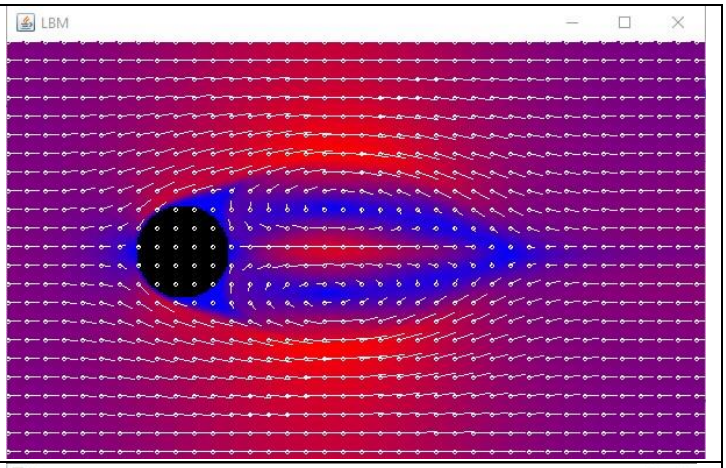
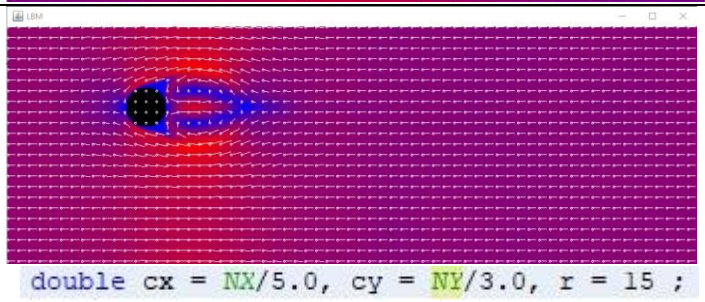


Figure 7.1.4 – Shedding iteration Lattice Boltzmann

Exercises 7.1

Variable	Change	Image

Lattice size NX, NY	Changing these values will keep the same affect however it will decrease or increase the size of the vortices depending on if you increase or decrease the volume.	
Obstacle size	In this case I move the obstacle up and made its radius smaller this cause the vortices that were made to become significantly smaller and also appear and disappear a lot earlier than iteration 5000. It then stretches at around 8000 iteration and deforms the vortices allow the alternating clockwise and anticlockwise vortices to occur.	 `double cx = NX/5.0, cy = NY/3.0, r = 15 ;`

Lab 8: An Attempted Application to Aerodynamics

In this lab we will be using an extended version of lattice Boltzmann program. If we consider the using the Boltzmann models for an aircraft wing, we will be restricted from achieving Reynold numbers, but the results are illustrative.

Experiment 8.1

In this experiment we will be run the air foil code and see it run under certain conditions. As mentioned before Reynolds number of a flow is denoted as $Re = |\mathrm{u}|\mathrm{L}/\mathrm{v}$. $|u|$ is some characteristic velocity of the flow and L some characteristic length scale. High Reynold numbers of a flow tends to associate to turbulent flow and low Reynolds number with laminar flow.

Within this new code the boundary conditions were revisited where the most important section would be the no-clip condition at the obstacle. Within the collision code there is a no slip obstacle code that simple replaces the normal collision step inside the obstacle by a step that replaces each distribution component by the value of the component with opposite velocity state. In the initialisation code there are three arrays that define the three indices of each of the states negative, zero and positive velocity components in x direction. The first section applies the moving planar

$$^{out}(x, y) = f_i{}^{in}(x, y) \quad \text{———————} \quad - f^{i}{}_{in}{}^{(x,y)} - {}_{\tau} f^{i}{}_{eq}{}^{(x,y)}$$ and the second part applies the Zou-He boundary f_i

boundary conditions or the Non-Equilibrium Bounce back method $f_i{}^{neq} = f_i - f_i{}^{eq}$. Due t there being no vertical y component of velocity at the inlet the transverse momentum correction is zero.

The method in the code that calculates the profile of an unrealistic symmetrical aerofoil with no camber is y_t(). Symmetrical aerofoil meaning that the profile of the top surface of the wing is the same as the button surface. The symmetrical shape is bent around a camber curve which is convex upwards, so that it will yield a more traditional looking aerofoil.

When the obstacle is created for the flow, the aerofoil needs to be tilted into the incoming flow with an angle of attack, which was defined by the alpha, so we apply a rotation matrix. Around the bounce back boundary conditions at the obstacle we have added code to calculate the momentum transferred to the obstacle in a single time step.

The AirFoil code will be run with an iteration of 50,000. As the code runs the average of the x direction and the y direction forces on the wing will be printed out into the console showing the drag and the lift from the aerofoil (Figure 8.1.1).

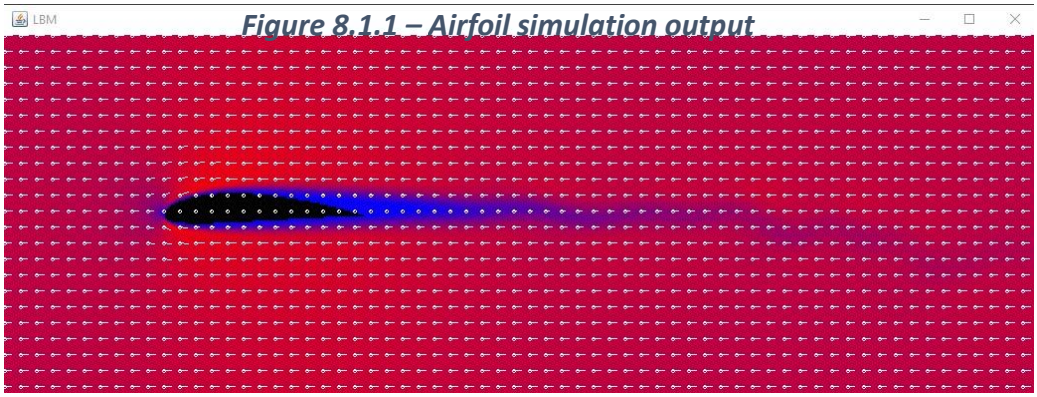


Figure 8.1.1 – Airfoil simulation output

Exercise 8.1

In this exercise I changed the iterations and set it to 50000 and change the angle of attack or the alpha and the value of lift dependant on those angles. Shown in the table below you can see that the alpha being at around 30 degrees was a lot more efficient than the lower values, however if you continue making the angle higher you notice that at 90 degrees the value of lift becomes inconsistent and suddenly drops off and cannot continue through the program.

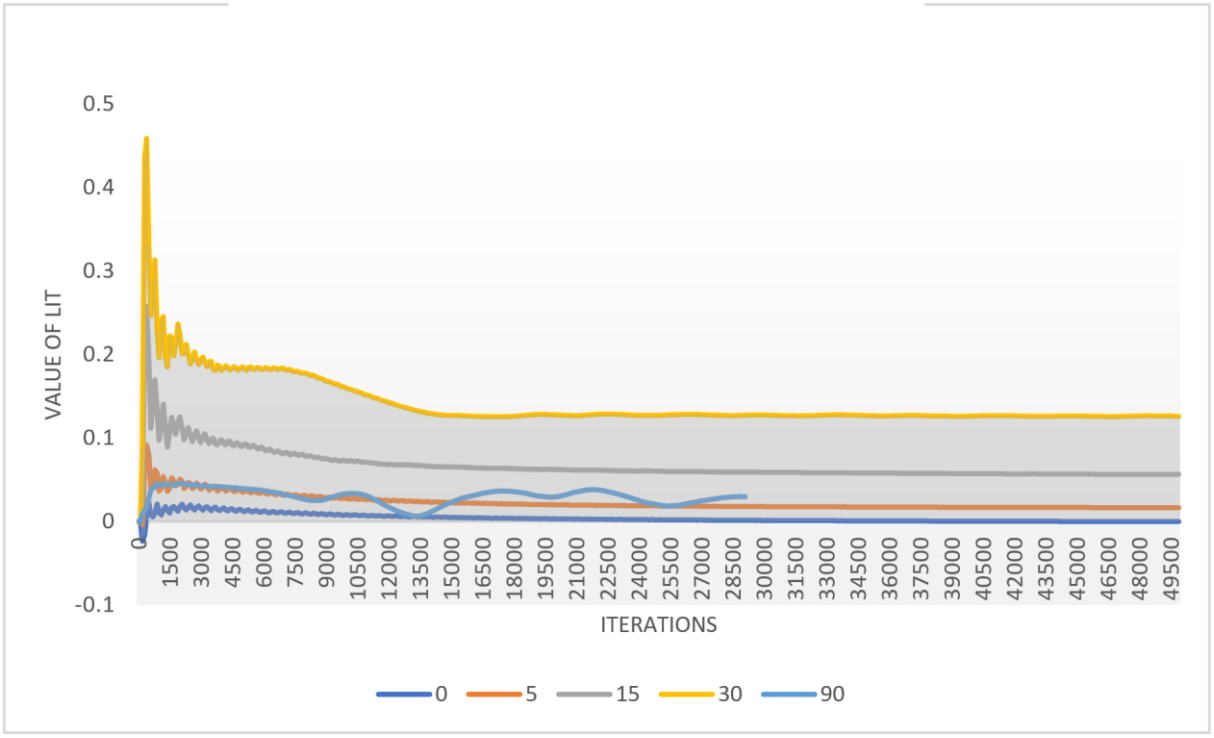


Figure 8.1.2 – Graph of Alpha over iterations

Lab 9: Solution of Differential Equations

In this lab we will be working with Newton equations of motion and demonstrating how these will affect two bodies that will be acting under force of gravity against each other.

Experiment 9.1

In this experiment we will be running a bit of code that is not accurate to real life but will demonstrate the laws of motion very well. The bodies in the code both will have a mass given to them in the case of the code they are defined as m1 and m2, they also will have a 2-dimensional representation respectively $(x1, y1)\ and\ (x2, y2)$ there velocities will be $(vx1, vy1)\ and\ (vx2, vy2)$. This means there are overall eight independent variables that will affect the movement of the bodies **(Figure 9.1.1)**. As the system is run longer and longer it will slowly

Figure 9.1.1 – Orbiting bodies Gif

decay due to the simulated system and this would not occur in real life.

Experiment 9.2

In this experiment we will use the Velocity Verlet algorithm. This is based on the leapfrog characteristics, which is visualised in **(Figure 9.2.1)**. An advantage of using leapfrog characteristics over Euler's method is that it has an accuracy of $O(h^2)$ rather than $O(h^2)$. This also has physics advantages where it has more accurate systems numerically. It's an invariant under time reversal, it's a symplectic update, which will cause consequence that it conserves the approximation to the total energy of the system. To start up the leapfrog we will need the velocity v_0 and x_0 at $time$ t_0. Leading to us doing half the step of Euler $v_{12} = v_0 + 12hA(x_0)$. (Figure 9.2.2) the simulation will show that it will not decay, and the energy is being conserved. As can be seen in the gif the orbiting bodies stay at the same distance and do not increase or decrease in speed showing they do not lose energy as the simulation continues.

Figure 9.2.1 – Leap Frog Characteristics

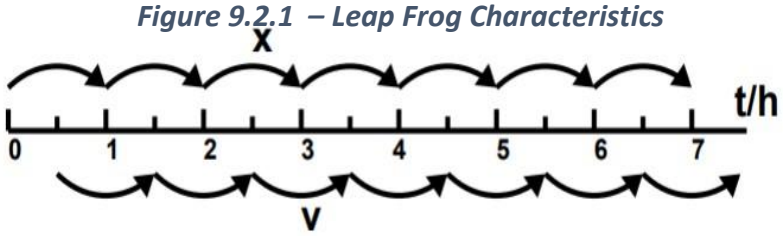


Figure 9.2.2 – Orbiting bodies Gif

Mini Project: Thread Parallel version of Lattice Boltzmann Model

The project I chose was to parallelise the Lattice Boltzmann Model, which was discussed in lab 7. The aim of this is to show an improved speed-up from the sequential version of the program.

Implementation M.1

When implementing the parallelised version of the code a cyclic barrier was put in place **(Figure M.1.2)**, so when the flow of the fluid is split between the threads it will synchronise and will wait for a set of threads to reach a common execution point rather than the threads working on their own. This stops the flow to act as one even though they are in different threads. Once this was

Figure M.1.2 – Cyclic Barrier Code

```
import java.util.concurrent.CyclicBarrier ;
static CyclicBarrier barrier = new CyclicBarrier(P) ;
```

function **(Figure M.1.2)** was implemented, so the threads can be synched together, when each of the threads are done each part that is calculated. This will also ask the threads to wait for the barrier. The class will be extended to work with threads. A lot of the variables that were within the code will also need to become global variables as the main calculation code will be moved out of the main function to a run functions. There are 9 variables that become global variables **(Figure M.1.3)**.

Figure M.1.2 – synch functions

```
static void synch() {
    try {
        barrier.await() ;
    }
    catch(Exception e) {
        e.printStackTrace() ;
        System.exit(1) ;
    }
```

Figure M.1.3 – Global Variables

```
static double [] [] [] fin = new double [NX] [NY] [Q] ;
static double [] [] [] fout = new double [NX] [NY] [Q] ;
static double [] [] rho = new double [NX] [NY] ;
static double [] [] vel = new double [NY] [2] ;
static int [] i1 = new int [3], i2 = new int [3], i3 = new int [3] ;
static int [] noslip = new int [Q] ;  // index in c of negative velocity state
static double cx = NX/4.0, cy = NY/2.0, r = 20 ;
static double nulb = uLB * r / Re ;
static double omega = 1.0 / (3 * nulb + 0.5) ;  // Relaxation parameter
```

A constructor for the class is used **(Figure M.1.4).** This will be used as a representation of a thread as it runs through the run function. The three calculation that will be separated into the threads is the macroscopic density and velocity calculation, the collision step, and the streaming step. As these will affect the lattice directly and the other calculation will be static and will not change dependant on the threads. The beginning and end of the calculation will be defined by the NY value as the flow will be moving in the x direction and will split better if the calculation was split in the y direction **(Figure M.1.5)**. Each loop that will work on the calculation will run from the begin and end variable in the y direction and the x direction will be unchanged. This parallelism was run with unrolled and rolled code **(Figure M.1.6)**.

Figure M.1.4 – Constructor

```
int me ;
public LBM (int me) {
        this.me = me ;
}
```

implemented a synch

Figure M.1.5 – Run function and thread start and stop

```
final static int B = NY / P;
public void run() {
    int begin = me * B ;
    int end = begin + B ;
```

Figure M.1.6 – Calculation Loop

```
for(int i = 0 ; i < NX ; i++) {
    for(int j = begin ; j < end ; j++) {
```

Results M.2

Both rolled and unrolled code was run with an iteration of 5000. As can be seen in **(Figure M.2.1)** the fastest number of threads for the rolled code was 10 threads with a speed-up of 4.26. The range Niter = 5000 of speed-ups is between 3 and 4 mostly, however the efficient decreases rapidly the more threads you add. For

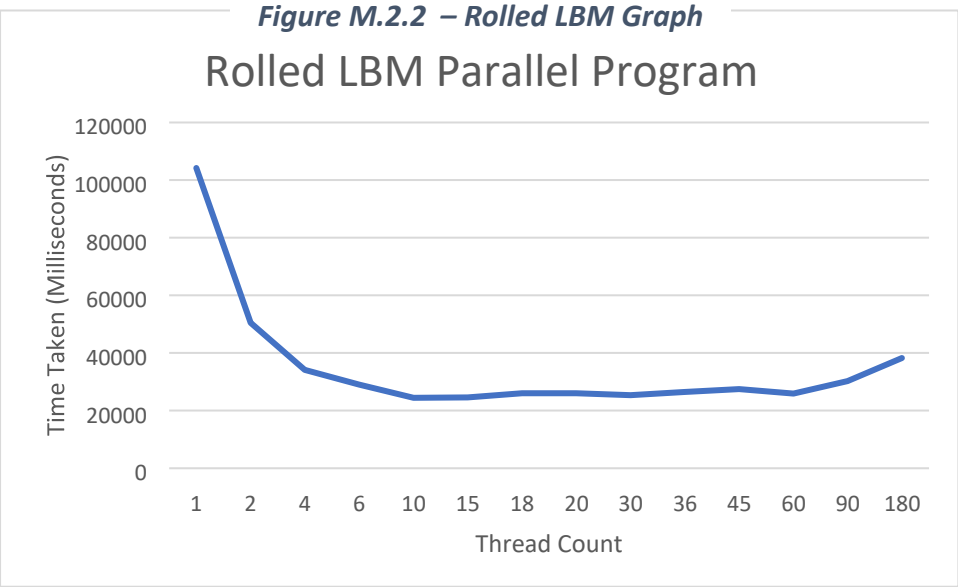
Figure M.2.1 – Rolled Table

Rolled	Time taken (Milliseconds)	Speed-up	Efficiency
1	104209	-	-
2	50431	2.07	1.04
4	34167	3.05	0.76

6	29047	3.59	0.6	
10	24450	4.26	0.43	example, with 180 threads the efficiency
15	24601	4.24	0.28	is 0.02 meaning its
18	26087	3.99	0.22	
20	25994	4.01	0.2	more efficient to use
30	25405	4.1	0.14	less threads rather
36	26522	3.93	0.11	than maxing out. The
45	27528	3.79	0.08	time taken seems to
60	25962	4.01	0.07	plateau after 10
90	30298	3.44	0.04	threads as it stays
180	38227	2.73	0.02	around the 25000

milliseconds makes as can be seen in **(Figure M.2.2),** however it starts to increase as you get above NY/2.

Figure M.2.2 – Rolled LBM Graph



As can be seen in **(Figure M.2.3)** there are two sets of threads that have the highest speed-up for unrolled code 15 threads and 10 threads, however the 15 threads have the lowest time taken by 40 milliseconds, which also proves that between 10 to 15 threads the time does not decrease that much. The efficient for 15 thread is low with 0.18 and as the threads increase the lower the efficiency as the time taken seems to plateau similarly to the rolled version **(Figure M.2.4)**. You can also notice as once the threads was more than the NY/2 it also increased the time taken and decreased the speed-up time as shown in 180 threads the time taken increases to 30734 milliseconds and the speed-up decreases to 1.57 where at 90 threads it was 2.37. There is a noticeable difference in the rolled and unrolled as the rolled version of the program was always about 10000 milliseconds longer to run, however the speed-up of the rolled code was a lot higher as it was mostly between 3 and 4 whereas the unrolled stay between 2 and 3. The efficiency followed the decreases over the increase of threads.

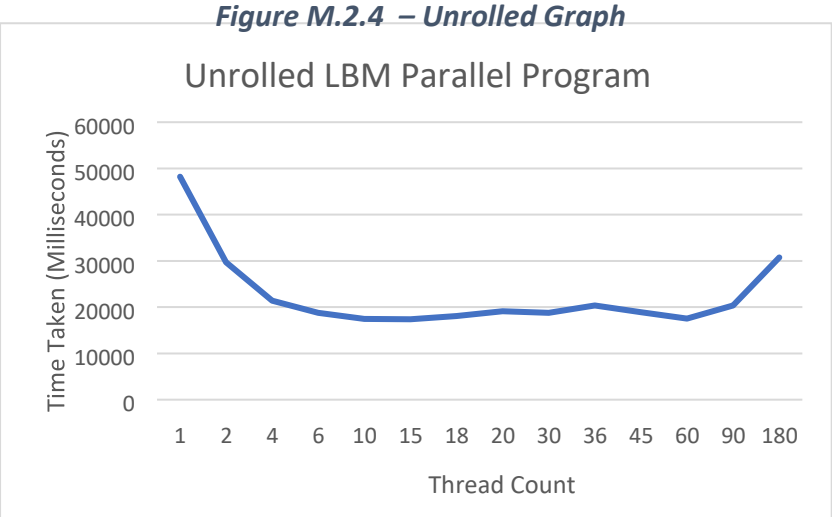
Niter =

5000

Figure M.2.3 – Unrolled Table

Unrolled	Time taken (Milliseconds)	Speed-up	Efficiency
1	48215	-	
2	29712	1.62	0.81
4	21424	2.25	0.56
6	18762	2.57	0.43
10	17425	2.77	0.28
15	17385	2.77	0.18
18	18073	2.67	0.15
20	19100	2.52	0.13
30	18776	2.57	0.09
36	20396	2.36	0.07
45	18882	2.55	0.06
60	17555	2.75	0.05
90	20382	2.37	0.03
180	30734	1.57	0.01

Figure M.2.4 – Unrolled Graph



Conclusion

The main lessons that were learnt from this course and labs was to be able to adapt scientific algorithms, such as Lattice Boltzmann algorithm, and be able to exploit high performance and parallel computation platforms, such as using unrolled and rolled code and implementing a working parallel version of said code. Assess and evaluating software platforms for kinds of scientific computing, such as using programs to filter images for CT scans or the use of radio interferometry data. Was able to explain to an appropriate level mathematical methods such as Cookley-Tukey FFT and was able to explain it in a lot of detail and how it is broken down. Was also able to implement algorithms that will simulate and visualize selected physical, Black whole filtering system, or biological systems, CT scan image filtering, and analyse the data scientifically.