Scientific Computing and Simulation Coursework

M30249

UP938652

Scientific Computing and Simulation

May 26, 2023

Contents

1 Introduction **2**

2 Labs **3**

2.1 Lab 1 - Fourier Transforms for Image Filtering . . . 3

2.1.1 Fourier Transforms . . . 3

2.1.2 Image Filtering . . . 4

2.2 Lab 2 - Fast Fourier Transforms . . . 5

2.2.1 Implementation and benchmarks . . . 5

2.2.2 Image Filtering . . . 6

2.2.3 Parallel implementation . . . 6

2.3 Lab 3 - Inverting the Radon Transform . . . 8

2.3.1 Unfiltered Back projection . . . 8

2.3.2 Filtered Back Projection . . . 8

2.4 Lab 4 - An attempt at sky imaging . . . 9

2.5 Lab 5 - Lattice gas models . . . 10

2.5.1 The Hardy, Pomeau, and de Pazzis model . . . 10

2.5.2 The Frisch, Hasslacher, and Pomeau model . . . 12

2.5.3 Comparison . . . 12

2.6 Lab 6 - Cellular Automata, Excitable Media and Cardiac Tissue . . . 13

2.6.1 Simple Three State Cellular Automata . . . 13

2.6.2 Gerhardt, Schuster, and Tyson Model . . . 13

2.6.3 4 State Automata . . . 14

2.7 Lab 7 - A Lattice Boltzmann Model . . . 16

2.7.1 Optimisations . . . 16

2.7.2 Obstacle shape . . . 16

2.7.3 Exploration of Reynolds Numbers . . . 17

2.8 Lab 8 - An Aerodynamics Application . . . 17

2.9 Lab 9 - Solution of Differential Equations . . . 17

2.9.1 SciPy Intergrate . . . 17

2.9.2 Velocity Verlet method . . . 18

2.9.3 Different conditions . . . 18

3 Personal Development Project **18**

3.1 Identifying areas for speedup . . . 18

3.2 Initial decomposition . . . 19

3.3 Cyclic barriers . . . 20

3.4 Benchmarking . . . 21

3.4.1 Cyclical decomposition . . . 21

3.4.2 Horizontal block decomposition . . . 21

4 Conclusion **22**

1 Introduction

Since their inception, computers have been used for scientific simulations. The algorithms and systems used to facilitate this is a strong field of study and prove invaluable for enabling insights into the complex nature of our world

My goal in this module is to gain a comprehensive understanding of how scientific computation can aid in solving these complex problems. Throughout this lab book, I will explore the theoretical and practical foundations of a range of topics, ranging from Fourier series and fluid simulation to radiointerformetry and cardiac tissues. Delving into their practical implementation. Throughout this module, I wish to develop a foundation in the applications of scientific computation to aid in simulating and processing real-world problems.

For my personal project, I will explore the area that piqued my interest the most, Lattice Boltzmann fluid simulation. This method proves powerful in its predictive power and seems able to be optimised and parallelised effectively. I wish to explore these systems in detail to gain insights into the behaviour of complex fluid systems.

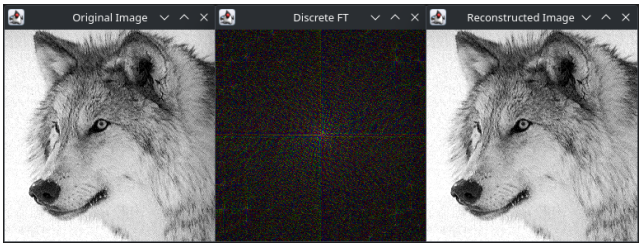

Figure 1 An image passed through a Forward and Inverse Fourier transform

2 Labs

2.1 Lab 1 - Fourier Transforms for Image Filtering

2.1.1 Fourier Transforms

This week I am investigating a Fourier Transform and how it can be used in image filtering. But first I need to complete the Fourier code from the skeleton provided. The code needs to calculate the formula: $C_{kl} = \frac{1}{N^2} \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} X_{mn}.e^{\frac{-2\pi i(km+nl)}{N}}$ I will break it down into calculating the real and imaginary parts separately.

```
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*(((double)k*m)/N + ((double)l*n)/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 ;
    }
}
```

This code runs, but it is hard to verify it is running correctly, the easiest way to do this would be to run it through an Inverse Fourier transform.

The equation for this is $X_{mn} = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} C_{kl}.e^{\frac{-2\pi i(km+nl)}{N}}$ As the result will be a real number we can forgo calculating the imaginary part in this step

```
for(int m = 0 ; m < N ; m++) {
    for(int n = 0 ; n < N ; n++) {
        double sum = 0;
        for(int k = 0; k < N ; k++) {
            for(int l = 0; l < N ; l++) {
                double arg = (2*Math.PI/N)*(k * m + n * l);
                double cos = Math.cos(arg);
                double sin = Math.sin(arg);
                sum += cos * CRe[k][l] - sin * CIm[k][l];
            }
        }
        reconstructed [m] [n] = sum ;
    }
}
```

as you can see in 1 the algorithm appears to be correct, or at least a reversible algorithm. Testing on an online IFFT I get the same wolf image out. This proves that my code is correct.

2.1.2 Image Filtering

With an image in the frequency domain, there are many advanced filters that can be applied, most simply a High-pass and a Low-pass filter. These only let the High or Low frequencies respectably.

Some of the simplest effects to achieve are high-pass and low-pass filters. This involves selectively removing parts of the frequency domain to remove noise or highlight grain in the image. More interesting effects will be explored in the next lab as the algorithm becomes faster to process.

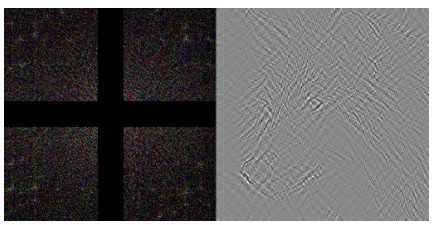

Figure 2 A high pass filter removing the lower $1/16^{th}$ of the image

Figure 3 A low pass filter keeping the lower $1/8^{th}$ of the image

2.2 Lab 2 - Fast Fourier Transforms

2.2.1 Implementation and benchmarks

Building off of the skeleton code given in the example I have made this transpose and 2D FFT function

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

public static void fft2d(double[][] re, double[][] im, int isgn) {
    int N = re[0].length;

    // Apply 1D FFT on each row
    for (int i = 0; i < N; i++) {
        fft1d(re[i], im[i], isgn);
    }

    transpose(re);
    transpose(im);

    // Apply 1D FFT on each column
    for (int i = 0; i < N; i++) {
        fft1d(re[i], im[i], isgn);
    }

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

This code produced identical results to the code in Lab 1 but in much less time. So this seems successful.

Algorithm	Time (ms)
Naive Fourier Transform	18457
Fast Fourier Transform	57

Table 1 speed comparison of Fourier and Fast Fourier

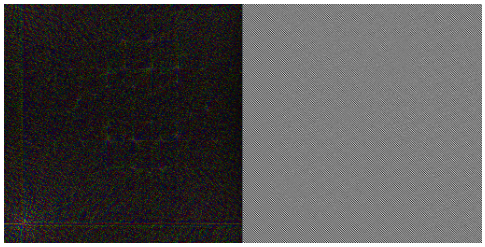

Figure 4 The image has been offset in the frequency domain before reconstruction

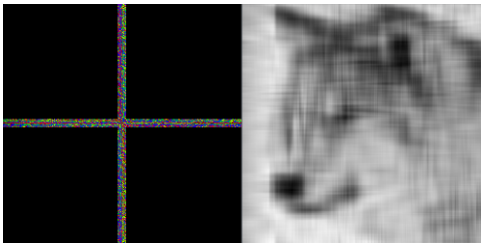

Figure 5 A selective low pass. It filters out all non-axis high frequency

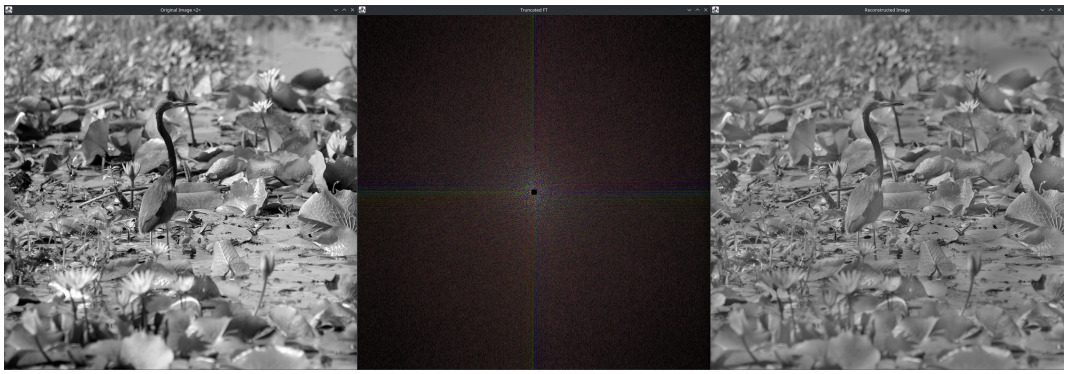

Figure 6 A small high pass to enhance detail in shadows

2.2.2 Image Filtering

To get an accurate speed comparison I disabled drawing of the results to the screen as they were taking up most of the processing time in the FFT example. In Table 1 I ran each program 10 times and took the minimum, the FFT ran > 300x faster than the Naive approach.

With it running faster I messed around with other filters and larger images. I experimented with transformations other than lowpass and highpass (see Figure 4) but to limited artistic effect. So I experimented with different effects that a high/low pass can get. With a larger more detailed image. (see Figure 6) I ran a very conservative high-pass, this had the effect of ”enhancing” details in the image such as those in shadow, see the bird’s neck. I also tried to go for stronger more artistic effects. (see Figure 5) In this I limited it to pixels close to the x-y axes, this gives a stretched-out blur effect that is quite striking.

2.2.3 Parallel implementation

I will go threaded, as the overall time is low I want a method with minimal overhead. I will use a shared memory approach with in-place operations as the rows will later affect the columns. I will try a cyclic decomposition as each 1dfft should take the same amount of time and cyclic can scale very well.

Number of Threads	Time (ms)	Speed Up	Parallel efficiency
1	253	–	–
2	231	1.10	54.8%
4	209	1.21	30.3%
8	211	1.20	15.0%

Figure 7 Parallel speed up and efficiency with barrier

Number of Threads	Time (ms)	Speed Up	Parallel efficiency
1	253	–	–
2	239	1.05	52.9%
4	200	1.27	31.6%
8	203	1.25	15.6%

Figure 8 Parallel speed up and efficiency without barriers

```
public void run() {
    // Apply 1D FFT on each row
    for (int i = me; i < N; i+=P) {
        FFT.fft1d(re[i], im[i], isgn);
    }
    synch();
    if (me == 0) {
        transpose(re);
        transpose(im);
    }
    synch();
    // Apply 1D FFT on each column
    for (int i = me; i < N; i+=P) {
        FFT.fft1d(re[i], im[i], isgn);
    }
    synch();
    if (me == 0) {
        transpose(re);
        transpose(im);
    }
}
```

In my first implementation, I forgot to add cyclic barriers. This caused the rows and columns to be flipped by thread 0 before other operations were finished. Sadly I couldn't get away with any fewer than 3. This however did not limit performance as it was already poor. see Tables 7 and 8. Through experimenting I got slightly better performance when removing the list transpositions, however, the improvement was marginal at best, and likely not worth the effort of implementing a column-wise 1dfft.

2.3 Lab 3 - Inverting the Radon Transform

2.3.1 Unfiltered Back projection

I first ran pure back projection. (See Figure 9) this produced a very hazy result with almost no resolution of the details. I was told that the contrast had to be kept low, or else filtering would not be needed. So I tested this hypothesis. (See Figure 10) No additional filtering was performed, but the contrast between the parts inside the head was increased from 0.02 units, up to 2 units. Even with this massive contrast, comparable to the difference between rock and air, only some of the details were resolved with the blur still present.

2.3.2 Filtered Back Projection

All processing in this section was done on the low-contrast version of the Shepp–Logan phantom to provide a greater difficulty in restoration.

Of these three filtering methods, I found the Ramp filter, seen in figure 11 to provide the most clarity. It has sharp edges to all the features, however, the concentric circles from the scanning process are also very sharp.

The lowpass cosign filter, seen in figure 13 maintains structural details while smoothing over some of the dense visual noise in the image.

The Ram Lak filter, seen in figure 12 seems almost too blurred and unclear. It does a great job of smoothing the jarring lines in the image but loses some of the clarity in the shape.

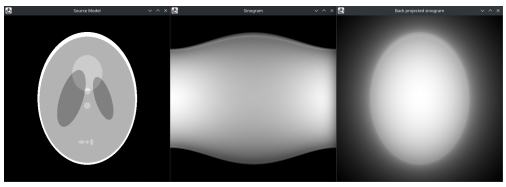

Figure 9 An unfiltered back projection with low contrast density (0.02)

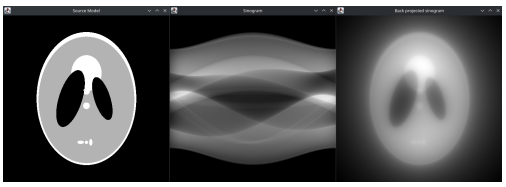

Figure 10 An unfiltered back projection with very high contrast density (2.0)

Figure 11 A filtered image using a Ramp Filter

Figure 12 A filtered image using a Ram Lak Filter

Figure 13 A filtered image using a Low pass Cosine Filter

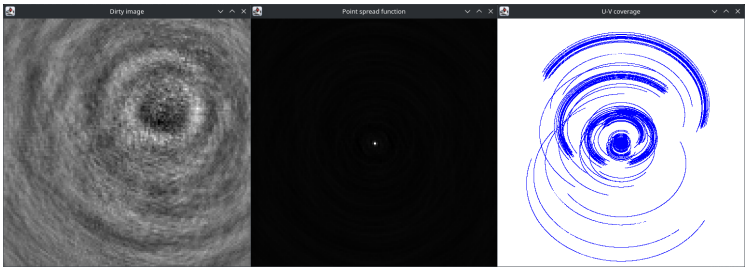

Figure 14 The sample VIS and UV file processed but not cleaned

2.4 Lab 4 - An attempt at sky imaging

In this lab, we will be processing data from a radiointerformetry telescope network. Samples are collected over a period of time to cover more of the UV plane. The earth's rotation around its axis and the sun can be used to move the position of the telescope.

Our sample here is small, only 100MB, Those that I found online cover from 10s of gigabytes to 100 terabytes, this program would take far too long to process that amount of data. We also lack any image-cleaning processes. So the raw information seen in figure 14 is about as good as we can expect from this sample size and processing method.

2.5 Lab 5 - Lattice gas models

2.5.1 The Hardy, Pomeau, and de Pazzis model

In the Hardy, Pomeau and de Pazzis model (HPP) particles can only exist on a grid, with two axis of movement. The model is broken down into two main phases, a collision phase, and a streaming phase. These simulate the collision of particles and then move the particles ready for the next iteration.

The collision has 3 main types, vertical, horizontal, and then the default state for every other possible particle arrangement at a point, no collision.

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

Then for the steaming step, you propagate particles into the new location.

```
// Streaming
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];
```

The encoding used here for the state is rather inefficient, An array of Booleans could easily be a bit string, and as there are only 4 states, it could easily fit in an int, using bit-wise operations to isolate the value I need.

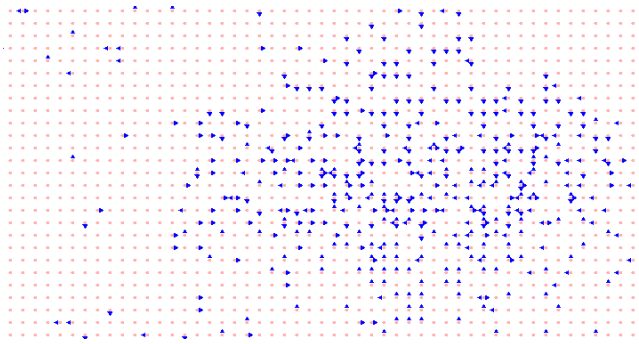

Figure 15 A subsection of the HPP simulation

Here are the key sections in the new multi-spin binary encoding

```
// Collisions
if (fin_ij == 3) {          // 1100
    // Verticle Collision
    fout [i] [j] = 12;      // 0011
} else if (fin_ij == 12){   // 0011
    // Horisontal Collision
    fout [i] [j] = 3;       // 1100
} else {
    // default, no collisions case:
    fout [i] [j] = fin_ij ;
}
```

```
// Streaming
fin [i] [j] =
    (fout [iP1] [j]   & 1) |
    (fout [iM1] [j]   & 2) |
    (fout [i]   [jP1] & 4) |
    (fout [i]   [jM1] & 8) ;
```

After implementing the faster multi-spin encoding I was able to simulate a much larger area in real time. This becomes impossible to visually interpret in the current rendering system, (See 15) To remedy this I implemented a density rending that supports densities over multiple cells. (See 16 and 17)

Figure 16 HPP Fluid sim with 5x5 super-cells at iteration 300

Figure 17 HPP Fluid sim with 5x5 super-cells at iteration 500

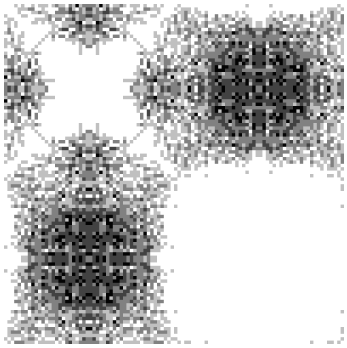

Figure 18 HPP Fluid on a 100x100 grid at iteration 150

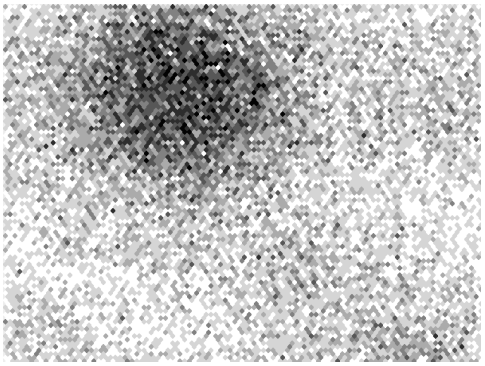

Figure 19 FHP fluid sim on a 100x100 lattice at iteration 150

2.5.2 The Frisch, Hasslacher, and Pomeau model

The Frisch, Hasslacher, and Pomeau model (FHP) differs from HPP in that it has 6 possible directions, with many more possibilities for collisions that change directions. The collisions are also probabilistic. This can apparently lead to more accurate simulations of fluids.

2.5.3 Comparison

To properly compare the two methods I needed to render a density map of FHP, this is done by rendering some small hexagons in a lattice.

```
static int [][] hex_lines = {{0,3,8,3},{1,2,6,5},{2,1,4,7},{3,0,2,9}};

public void draw_hex(Graphics g, int n, int x, int y){
    g.setColor(greys[n]);
    int originX = ((HEX_W * x + HEX_HW * y) % displaySizeX );
    int originY = y * HEX_HH;

    // Draw hex out of overlaying hexagons
    for (int i = 0; i < hex_lines.length; i += 1){
        g.fillRect(
            hex_lines[i][0] * CELL_SIZE + originX,
            hex_lines[i][1] * CELL_SIZE + originY,
            hex_lines[i][2] * CELL_SIZE,
            hex_lines[i][3] * CELL_SIZE
        );
    }
}
```

One method I devised to test the functions was to simulate some real-world situations. In the real world, gasses diffuse to fill space. In this case with some high-pressure areas and some low due to the initial shock-wave. If you look at figure 18 you can see there are locations with zero gas and other areas with very high concentrations. Whereas in figure 19 the lattice is filled in most areas with high-pressure areas and low-pressure areas as expected. Here the FHP method seems to be more accurate.

Figure 20 Three state automata linear wave

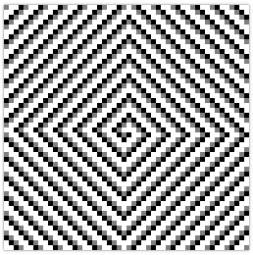

Figure 21 Three state automata circular wave

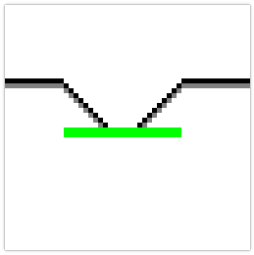

Figure 22 Three state automata linear wave and a wave break (in green)

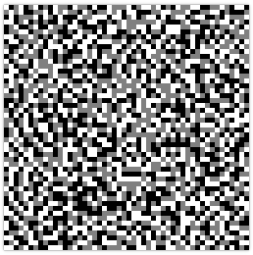

Figure 23 Three state automata after many iterations with a random start

2.6 Lab 6 - Cellular Automata, Excitable Media and Cardiac Tissue

2.6.1 Simple Three State Cellular Automata

This model is designed to model excitable systems a simple method for this is the Three state system. Each cell has 3 states, Resting (Excitable), Excited, and Recovering. The system can model many natural systems such as wildfires, but in it's current state has some limitations.

These automata are mainly limited to two modes of operation. A cycle or a single wave moving across the media. (See Figures: 20 and 21) The simulation is able to simulate the propagation of waves through the media even with a simulated "break" in the media, (See Figure: 22) This acts as I expect a single linear wave propagating across the media, only delayed by the break. As another exploration of its simulation power, I tested a random start. (See Figure: 23) This causes the media to create regions that are thin, and have a stable shape, yet continually cycle between states.

2.6.2 Gerhardt, Schuster, and Tyson Model

The main difference between the GST model and the Three State model is how cells transition. Cells need a given number of neighbours in a given radius to be excited. Each cell instead of just having a state, also has an excitement value associated with it, this changes as it excites and rests.

I modified the rendering code to visualise the excitement of a cell as greyscale where V_MAX is black and Zero is white. See Figures 24 and 25.

To get a better understanding of this model I have had to make a new visualisation. Red cells are active, blue are recovering and white are resting with the brightness visualising the value of V. The white cells still have a V that could be visualised, but it has no effect on the simulation so has been omitted. There are many variables to play with but I found it comes down to 4 main ones. For a visual representation of the effects see figure 26

After experimentation, I gained an understanding of the variables and their effect on the simulation. See Table 2 With these I can control the length of both wavefronts and recovery times.

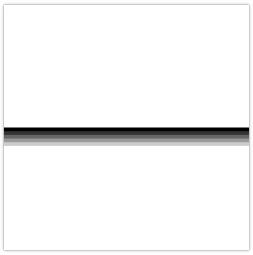

Figure 24 GTS automata linear wave

Figure 25 GTS automata circular wave

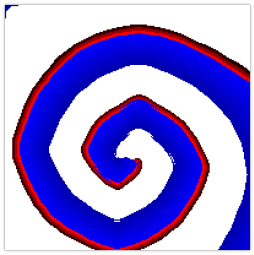

Figure 26 GTS automata with short excitement period and long recovery

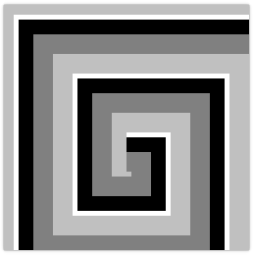

Figure 27 4-state automata with a circular wave

Variable	Effect
G_UP	Excited wavefront, lower is longer
G_DOWN	Recovering wavefront, lower is longer
V_RECO and V_EXCI	Recovering wavefront, greater distance between is longer

Table 2 Caption

2.6.3 4 State Automata

In the production of the 4-state automata from the source of the 3-state only a couple of changes needed to be made.

```
static int[][] timeToStateChange = new int[N][N];
final static int[] transition_delays = {0,3,3,2};
...
// Changes to excitedNeighbor for 8 neighbours
excitedNeighbour[i][j] =
       state[i][jp] == 2 || state[i][jp] == 3
    ... checking all neighbours...
    || state[im][jp] == 2 || state[im][jp] == 3;
...
// A re-write of update state
switch (state[i][j]) {
    case 0:
        if (excitedNeighbour[i][j])
            { state[i][j] = 3;
            timeToStateChange[i][j] =  transition_delays[3];}
        break;
    default:
        switch (timeToStateChange[i][j]) {
            case 0:
                state[i][j] -= 1;
                timeToStateChange[i][j] = transition_delays[state[i][j]];
                break;
            default:
                timeToStateChange[i][j] -= 1;
                break;
        }
        break;
}
```

The main visual effect of changing to a 4-state system is slowing the whole thing down, each wave now takes many more iterations to propagate, see Figure 27. The changes do however allow for more control over the periods of these effects like in the GTS model. But in a clearer more easily understandable format.

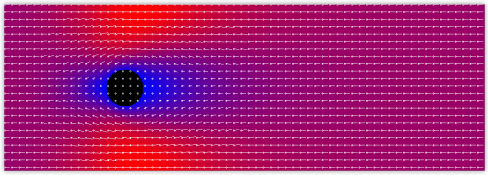

Figure 28 A simulation with a Reynolds number of 1.

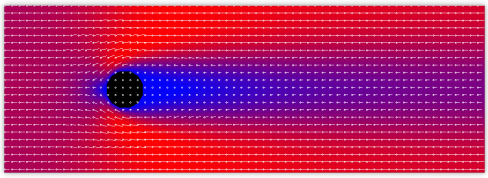

Figure 29 A simulation with a Reynolds number of 10.

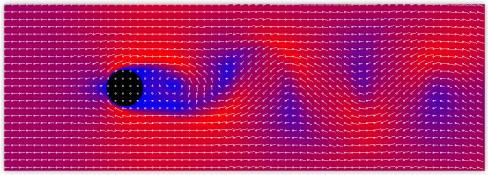

Figure 30 A simulation with a Reynolds number of 100.

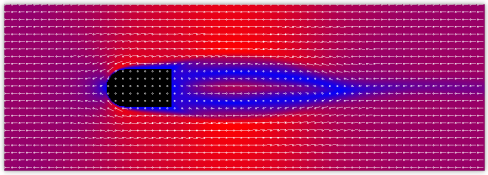

Figure 31 A bullet-shaped obstacle at 15,000 steps.

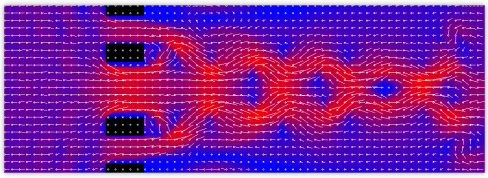

Figure 32 Interference between the shed vortexes of 3 gaps.

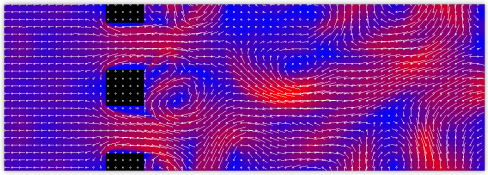

Figure 33 Highly chaotic flow behind 2 gaps.

2.7 Lab 7 - A Lattice Boltzmann Model

2.7.1 Optimisations

The code runs quite slowly with some simple avenues for optimisation such as loop unrolling, dropping time to run down from 67 seconds to 36. There are many reasons why unrolling loops can lead to a speed-up, a lot of them have to do with optimisations on modern CPUs such as better cache utilisation and not needing branch predictions or context switching.

2.7.2 Obstacle shape

I have tested a few interesting obstacle shapes.

Firstly the bullet. See figure 31 It is still stable at 15000 steps, and doesn't start vortex shedding till 25000 steps. Opposed to the cylinder that was already shedding by 15000 steps. As we are simulating at such slow speeds and low Reynolds numbers there are likely more aerodynamic shapes available, but this was the best I could come up with.

Secondly, I tried a series of gaps and tuned their gaps to produce resonance after the obstacle, see figure 32. Here you can see the waves from the top gap and the bottom gap constructively and destructively. as they are both 180° out of phase.

Finally, I have tried to simulate a chaotic system. To achieve this I went down from 3 gaps to two and tweaked sizes till it flowed wildly. The system quickly devolves into many

Angle of attack	Lift to Drag ratio	Angle of attack	Lift to Drag ratio
-20	-1.57	15	-0.68
-15	-2.45	20	1.32
-10	-2.71	25	1.50
-5	-1.77	30	1.46
0	-1.93	35	1.25
5	-0.45	40	1.06
10	-0.56	45	0.92

Table 3 Angle of attack vs Lift to drag ratio

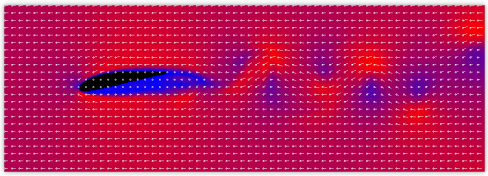

Figure 34 Airflow over an aerofoil with a -10° angle of attack

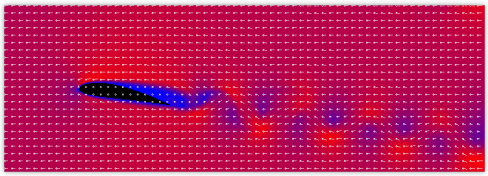

Figure 35 Airflow over an aerofoil with a +10° angle of attack

interacting vortexes. See figure 33. With this simulation, I was able to see many vortexes join and destroy each other.

2.7.3 Exploration of Reynolds Numbers

To test how accurate the simulation is I found a paper that has a similar setup and varies the Reynolds number to explore how the simulation changes. See Tansley and Marshall, 2001. My simulations in Figures 28, 29, and 30 closely match those from the paper. This shows that the simulation is very good at simulating realistic fluids. The issue is that the simulation breaks down with high Reynolds numbers or flow velocity.

2.8 Lab 8 - An Aerodynamics Application

After many simulations of various angles of attack, See table 3. Quite surprisingly the wing is significantly more effective when pitching down, see Figure 34 There is a much bigger negative pressure void under the wing when it is at -10°, this could be the cause of the significant increase in lift. The wake that it leaves behind is also lower frequency.

2.9 Lab 9 - Solution of Differential Equations

2.9.1 SciPy Intergrate

The SciPy integration function produces an animation of the two orbiting bodies. One peculiar fact about the animation, as it is played, is that the speed of the bodies is inverse to what would be expected of bodies in motion. They are travelling slowest at their periapsis and fastest at apoapsis. This leads me to believe that the method is using a dynamic time step to help reduce instability. Reducing the time step between samples as the system becomes more unstable.

The orbital decay is due to a known issue in integrating Hamiltonian systems. For no energy loss, a symplectic integrator needs to be used. The SciPy default settings for solve_ivp are not the correct choice here. (See, Saha and Tremaine, 1992)

```
res = scipy.integrate.solve_ivp(fun, (0, 50), y, method="Radau")
```

By updating this one line of code to change the integration method energy is conserved. It is still using the Runge-Kutta method, but of a different order. (See, Wanner and Hairer, 1996)

2.9.2 Velocity Verlet method

This method has a fixed time step, so the time step has to be tuned so that accuracy is maintained at periapsis.

For any value of $\Delta t > 0.12$ then energy is added to the system and either immediately or after a few orbits the bodies are lost. With a $\Delta t < 0.01$ the simulation appears stable. At least for this setup.

2.9.3 Different conditions

Changing the initial velocity of m1 to a (0,0) causes them to become extremely close at a point in their orbit. The dynamic timestep method applied in the SciPy method allows the system to stay stable under these extreme conditions. Whereas any time step that can be computed in a reasonable amount of time using the Velocity Verlet method is completely unstable. However, the Velocity Verlet has a huge benefit in that it is simple to understand and implement.

3 Personal Development Project

Parallel Lattice Boltzmann Model

I took a liking to this gas simulation in the weekly labs and would love to experiment with it over larger time scales and with larger scenes. But this would require some speed up of the algorithm so I have chosen to make it parallel.

3.1 Identifying areas for speedup

To make a parallel version of the code we need to first logically deconstruct the program flow to find areas that can be done in parallel

- Setup initial conditions
- For each time step
  - Calculate macroscopic density and velocity for each point
  - Calculate Collisions for each point
  - Streaming for each point, done row-wise

- Right wall outflow
- Displaying flow

The initial setup is only run once, whereas the rest is run for every step, so the setup can be ignored as it would have a negligible effect on the run time.

Displaying is only run every 100 steps and will be made more infrequent as the program runs faster, so I can ignore it.

The last 4 sections are run for every cell, every single iteration, so those are the best candidates for parallelisation.

3.2 Initial decomposition

There is some preparation that needs to be done before I can start running it in parallel.

```
public class LBMPar extends Thread{
    ...
    //extra static declarations
    ...

    public static <bool> void main(String args []) throws Exception {
        ...
        //Setup initial conditions
        ...

        LBMPar [] threads = new LBMPar [P] ;
        for(int me = 0 ; me < P ; me++) {
            threads [me] = new LBMPar(me) ;
            threads [me].start() ;
        }

        for(int me = 0 ; me < P ; me++) {
            threads [me].join() ;
        }

    }

    public void run(){
        ...
    }

    ...

    int me;
    LBMPar(int me){
        this.me = me;
    }
}
```

Many snippets have been omitted as they are the same as the serial version. When moving the main code into the run block, lots of variables have to be statically declared to allow shared memory access between the threads.

For my first test, all of the 4 blocks will be cyclically decomposed.

Section	Read	Write
Calculate densities	fin, vel	u
Collisions	fin, u	fout
Steaming	fout	fin
Overflow	fin	fin

Table 4 Memory access in key functions

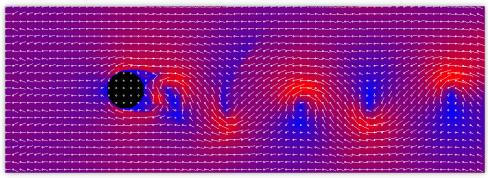

Figure 36 The expected result after 30,000 steps

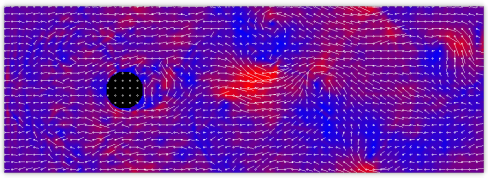

Figure 37 The result with a missing barrier between Overflow and Density.

```
for(int i = me ; i < NX ; i+=P) {
    ...
}
```

This is a good start as it should be a mostly even split amongst the threads, it is also one of the simplest to implement.

3.3 Cyclic barriers

To find out where the barriers are needed each section should be broken down into what memory it reads and writes to, there are 4 main arrays that are in use, fin, fout, vel, u.

Between the first and second item, there is one access collision, u is written to in the densities step and then read in collisions. However, u is never read at neighbouring cells, so threads will not collide. These first sections are parallel safe.

A barrier is needed between collision and streaming both the collision and streaming steps read and write to neighbouring cells which could contaminate the memory other threads are using.

Streaming step accesses neighbouring cells in all directions and overflow accesses neighbours only within the row. But these do need to be separated as you can see in Figure.37 This leads to a minute error that will compound over the simulation.

After overflow writes to fin density reads neighbouring cells from fin, so the last barrier must go here.

It would initially appear that you can get away with only two, but forgoing the barrier between streaming and overflow adds a subtle error to the simulation at the right-hand boundary. See Figures.36 and 37

To test all the barriers work I will execute the code with 180 threads, one for each line. Any errors should be multiplied as there are not enough physical cores to execute them, they will happen sequentially till they hit the barrier, so any possible memory conflicts should occur. After running this test the results are identical to the serial version.

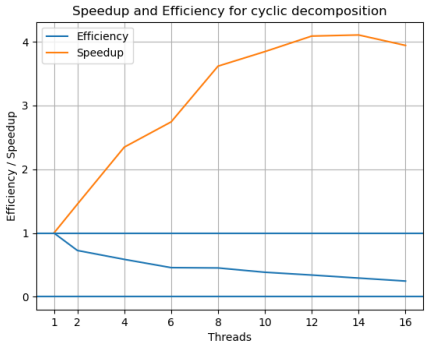

Figure 38 The speedup and efficacy for cyclic decomposition.

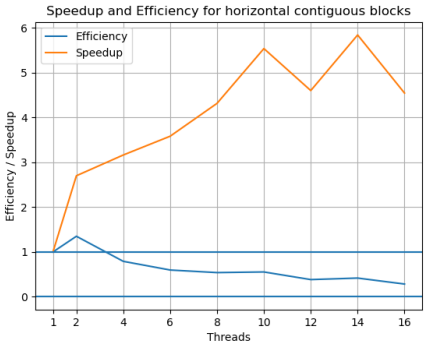

Figure 39 Speed up and efficiency for horizontal blocks

3.4 Benchmarking

I ran into an issue when benchmarking. The temperature of my CPU greatly affected the performance. With high summer temperatures, my CPUs speed fell quickly when running large threaded workloads. To try and standardise the results I ran each test 10 times and took the minimum. I waited between tests for the CPU to cool down again. I avoided running benchmarks at the warmest times of the day. However, there is still a high amount of variation in the results.

All the benchmarks will be at a grid size of $520x180$ and with a cylindrical obstacle.

3.4.1 Cyclical decomposition

```
for(int i = me ; i < NX ; i+=P) {
    ...
}
```

See Figure.38 for results. This method of decomposition was my first instinct for this problem. It seems like it would provide a good workload split over the threads. This plays out in the speed-up and thread efficiency. The graph is exactly as I would expect for a problem with multiple cyclic barriers. And on a CPU with 8 Physical and 16 logical cores. The performance steadily increases till it hits the physical core limit and then plateaus. I was pleasantly surprised by how high the efficiency stays.

3.4.2 Horizontal block decomposition

```
int blockSize = (NX + P - 1) / P;
int startIndex = me * blockSize;
int endIndex = Math.min(startIndex + blockSize, NX);

for (int i = startIndex; i < endIndex; i++) {
    ...
}
```

I next tried splitting it into blocks stretching the whole length of the simulation and vertically contiguous. I felt that this might lean more into the CPUs optimisations in path

prediction. See Figure.39 for these results. However, the results have left me baffled. I have re-run the tests, making sure the CPU is at the same temperatures each time. To further mitigate any error I have set my computer's CPU to a locked clock speed and launched Linux in a minimal terminal mode with no GUI or other major programs running. I have been running the same code and setting the thread number to 1. Not a special serial version of the code. And I have increased to 30 samples for each number of threads.

I have attempted to control every possible external variable I can think of, yet I am still getting 135% threading efficiency. The single-threaded version is taking 72 Seconds to calculate so it is not a trivial amount of time where simple variance could explain it. Apparently, this problem works extremely well under these conditions giving me a speed-up of almost 6x.

Other tests were not performed under such strict conditions as they made using my computer incredibly awkward. I will leave this result as an anomaly and just accept that this method of decomposition performs better than cyclic.

4 Conclusion

In conclusion, the field of scientific computation has proven to be invaluable for gaining an understanding of complex real-world systems.

Throughout the module, my primary objective has been to gain an understanding of how scientific computation can be used to model and predict complex systems and help process scientific data. I feel as though I have a good knowledge of these systems and how they work. From designing my own image filters in the frequency domain I have developed insights into how the Fourier Series can be applied to a staggering range of problems.

Delving into the many methods of fluid simulations and how staggeringly accurate they can be with such a simple rule set. My understanding of fluid simulation using many different methods has given me great confidence in the field.

I can say that I have fulfilled all my goals with this module and have thoroughly enjoyed this broad exploration of scientific computation.

References

Saha, P., & Tremaine, S. (1992). Symplectic integrators for solar system dynamics. *Astronomical Journal (ISSN 0004-6256), vol. 104, no. 4, p. 1633-1640.*, *104*, 1633–1640.

Tansley, C. E., & Marshall, D. P. (2001). Flow past a cylinder on a plane, with application to gulf stream separation and the antarctic circumpolar current. *Journal of Physical Oceanography*, *31*(11), 3274–3283. https://doi.org/10.1175/1520-0485(2001)031<3274:FPACOA>2.0.CO;2

Wanner, G., & Hairer, E. (1996). *Solving ordinary differential equations ii* (Vol. 375). Springer Berlin Heidelberg New York.