Reaction Diffusion Simulation Bottlenecks & Limitations

Natan Sonni Szczepaniak

University of Exeter
College of Engineering, Mathematics and Physical Sciences

Abstract

In this report we find different ways of improving the efficiency of a Reaction Diffusion simulation. This will be done by implementing it in a lower level compiled language, changing the algorithm and trying to render it with an OpenGL framework. The results will then be compared by benchmarking the programs to measure increase in speed.

1 Reaction Diffusion Simulation

Reaction Diffusion Systems are a collection of mathematical models describing various physical processes. The general form of a Reaction Diffusion equation compromises of a reaction term and a diffusion term.

$$\frac{\partial \mathbf{q}(\mathbf{r},t)}{\partial t} = \underline{\underline{\mathbf{D}}}\nabla^2\mathbf{q}(\mathbf{r},t) + \mathbf{R}(\mathbf{q}(\mathbf{r},\mathrm{t})) \tag{1}$$

Where $\mathbf{q}(\mathbf{r},t)$ is a desired function, $\underline{\underline{\mathbf{D}}}$ is a matrix of coefficients for the diffusion term and $\mathbf{R}(\mathbf{q})$ is the reaction term which depends on a given system.

These models can be applied to systems of different dimensions. The one-dimensional case, otherwise known as the Fisher-Kolomogorov-Petrovski-Piskunov (Fisher-KPP) equation takes the following form.

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2} + R(u) \tag{2}$$

Depending on the choice of the reaction term $R(u)$, the equation can be used to describe different systems. Spread of biological populations can be described by the Fisher's equation with $R(u) = u(1-u)$.[7] Temperature in Rayleigh-Bénard convection can be calculated using $R(u) = u(1-u^2)$ and $R(u) = u(1-u)(u-\alpha)$ gives the Zeldovich equation used to study reaction rate in combustion mechanisms.[8][22]

In 1952, Alan Turing wrote about these types of systems in his paper *The chemical basis of morphogenesis* "a system, although it may originally be quite homogeneous, may later develop a pattern or structure due to an instability of the homogeneous equilibrium, which is triggered off by random disturbances" [31]. Due to his focus on living organisms, he predominantly focused on two-dimensional systems comparing them to patterns observed in nature. Having found the relationship among numerous patterns of different organisms at different scales

ranging from patterns of Hydra tentacles to the surface of an embryo during gastrulation, led to the distinct pattern to being known as "Turing Patterns".

Turing suggested a system of two chemicals **A** and **B** that interact to create unique stable patterns providing the basis for the process of morphogenesis. Chemical **B** produces more of chemicals **A** and **B** whereas **A** inhibits the production of **B**. Now if chemical **A** diffuses faster than **B** we will obtain with patches of high concentration of chemical **B**. This leads to patches of different shapes which are determined by the coefficients in the equations.

This 2D system can be described by Equation 1 using u and v as vector functions for chemicals A and B. The partial-differential equation then takes the form of.

$$\begin{aligned} \partial_t u &= D_u \nabla^2 u + R_u(u, v), \\ \partial_t v &= D_v \nabla^2 v + R_v(u, v) \end{aligned} \tag{3}$$

In the equation above, we see a general case for a two-dimensional system of 2 chemicals. Chemical A is represented by u and chemical B is represented by v following the notation conventions in literature.[15] D_u and D_v are the diffusion coefficients the two different chemicals. $R_{u,v}$ describe the reaction between the chemicals. Along with the diffusion coefficients, this term affects the type of patterns generated. There are different models that can be applied here to replicate the behaviour described by Turing including Fitzhugh–Nagumo [33], Lotka-Volterra [21] and Gray-Scott [16].

In order to visualise each one of these systems on a two-dimensional grid, we need to approximate the chemicals as two numbers ranging from 0 to 1 at each cell on the grid. The numbers correspond to the concentration of the specific chemical at each cell.

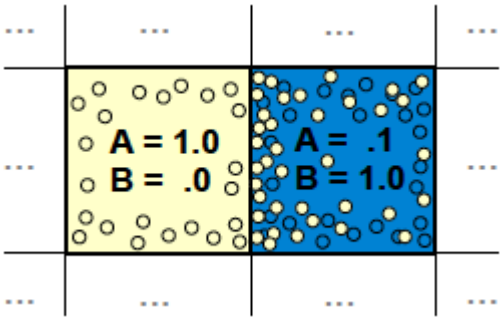

Figure 1: Approximation of system state on a grid based on the concentration of chemicals at each grid cell [28]

For my use case, I will focus on the Gray-Scott model as it provides the most versatility when it comes to generating unique patterns from different parameters. In this model, chemical A is added to the system at a "feed rate" f and Chemical B is removed at a given "kill rate" k. The reaction that takes place between the different chemicals is that when two **B** particles are in vicinity of one **A** particle, they convert the **A** into **B**. This is not too dissimilar from the mechanism behind the cellular automaton "Conway's Game of Life" which has been used to explore ecological and biological systems.[5] In this case however, the rules provide us with the information needed to complete our equation with the correct $R_{u,v}$ terms giving.

$$\begin{aligned} \partial_t u &= D_u \nabla^2 u - uv^2 + f \cdot (1 - u), \\ \partial_t v &= D_v \nabla^2 v + uv^2 + v \cdot (k + f) \end{aligned} \tag{4}$$

The first term of the reaction function $R(u, v)$ describes the interaction between the chemicals. As the concentration of each chemical in a cell ranges from 0 to 1, the chance that two of chemical **B** (v) and one chemical **A** (u) come together is uv^2. As **A** is converted to **B**, this is taken away from the number of chemical **A** and added to the number of chemical **B**. The second term of the reaction function models the feeding and killing of the chemicals as described previously. The feed rate term (which adds A particles) is scaled by $(1-u)$ in order to make sure that the concentration of **A** particles stays below 1 and the kill rate term (which removes B particles) is scaled by v to make sure concentration of **B** doesn't go below 0. In this model, the kill term includes $(k+f)$ to ensure the apparent kill rate doesn't go below the feed rate.

This model results in the patterns described by Alan Turing. The following are some results of a Gray-Scott model simulation.

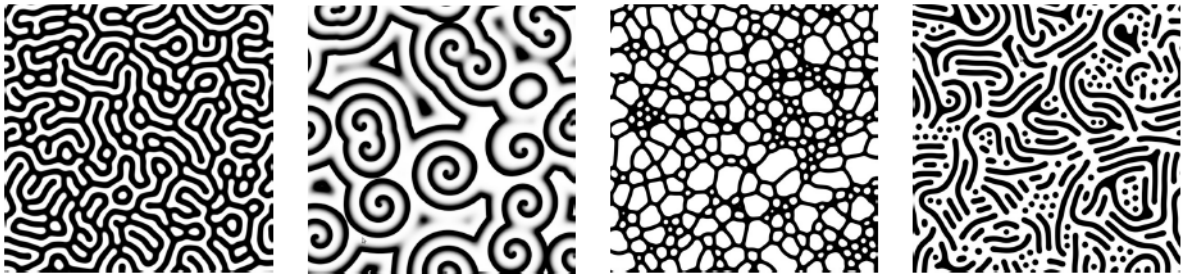

Figure 2: Patterns generated using the Gray-Scott model. Differences in the shapes is dependent on the parameters [28]

As mentioned by Turing, these patterns are often observed in living organisms of all sizes. Here are some examples.

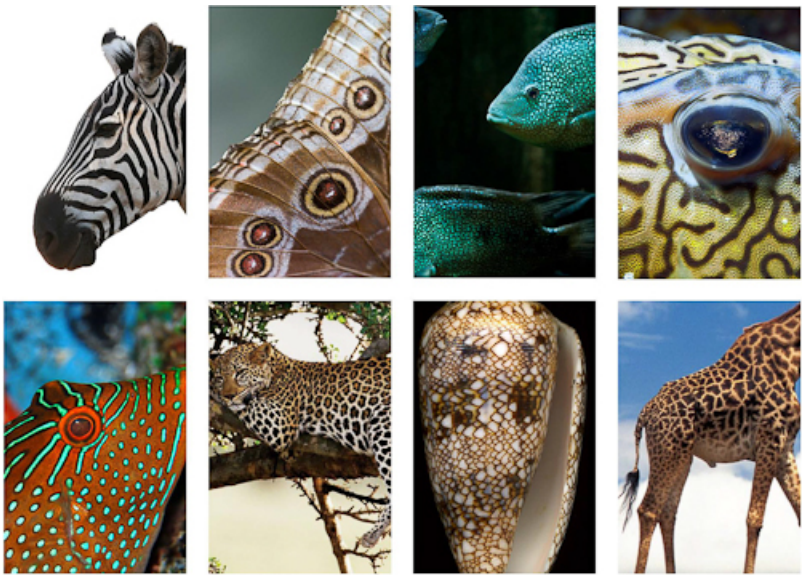

Figure 3: Patterns found in nature on various animal species from different habitats. This is not the only place this behaviour is found however these ones are easily visualised. [31]

This trend is a result of the prevalence of activator-inhibitor systems in the pattern formation stage in living organisms.[20] This is when two substances interact to create gradients and patterns from a seemingly homogeneous starting point. Due to the simplicity of this biological framework, the patterns can be modelled with a relatively simple set of equations as described before. In order to explore all of the possible patterns possible, a plot in parameter space can be investigated. This is where the feed rate and kill rate vary along the x and y axis essentially giving different patterns at different coordinates along with the transitions between them. such a plot can be seen below.

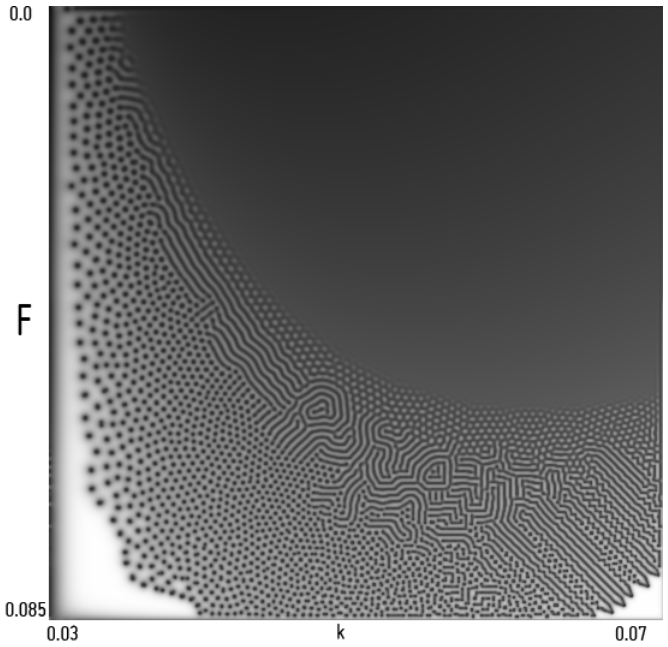

Figure 4: Parameter space of the Gray-Scott model varying feed-rate (F) from 0 to 0.085 and kill rate (k) from 0.03 to 0.07 [28]

Spatial variation in the feed and kill rate gives rise to more unique patterns that depend on the shape of the surface it is formed on. This accounts for the transition in pattern in the puffer fish seen in the bottom left of Figure 3 near the eye area. Spatial variation of diffusion rates also leads to a variation of size of patterns such as the ones seen on the legs of the giraffe compared to its torso.

Additionally to these examples, there also exist more counter intuitive examples worth mentioning. Examples of these unique patterns are everywhere, we don't need to go far to find them as our own fingertips have been hypothesised to originate from this kind of system [10]. This means that hypothetically, given the correct parameters and geometry, we should be able to recreate the unique patterns of our fingerprints.

A 2003 study [13] used this to explore the use of the Reaction Diffusion model to restore low quality fingerprints. Given an image of a low-resolution fingerprint image as a starting point, the researchers were able to apply a "Digital Reaction-Diffusion System" algorithm to enhance the pre-existing patterns. Another area of interest for these two-dimensional systems is by projecting them onto 3D geometries. [17] This method has been used to study the development of embryos as well as the propagation of electrical signals in the heart.[25]

These kinds of systems become more engrossing in higher dimensions. In 3D, this concept was used to study formation of lymphatic vessels (responsible for distributing water around the body), blood clotting, electric discharge in gasses as well as the self-sustaining Belousov–Zhabotinsky reaction. [14], [9], [6]

The reason this set of equations is so versatile is due to its simplicity. This naturally poses a number of limitations however there is still a wealth of insight to be obtained from these models and applications to be found. For this Report, I will focus on solely the 2D case however the improvements in methods suggested here can be applied to any number of dimensions with a substitution of higher dimension equations in the program.

The purpose of this report is to explore the efficiency of implementation of the Gray-Scott algorithm to two-dimensional systems and compare different computational methods for this purpose. I aim to cover the process of creating a piece of simulation software from scratch in order to get an in-depth look to find the most significant bottlenecks and tackle the appropriately.

2 Implementations & Bottlenecks

Before approaching the development process of this type of software some considerations must be taken into account. Namely, the programming language, mathematical engine to carry out calculations and a method for graphical visualisation must all be chosen with care to ensure the best potential for future improvement.

If the wrong tools are chosen, the software may run into possible bottlenecks that would end up being difficult to solve. Listed below are choices available to us when creating a 2D visual simulation program requiring a convolution method.

Possible Choices:

1. Programming Language (Interpreted vs Compiled Languages)
2. Convolution Function Method (Brute force vs FFT Convolution)
3. Graphical Visualisation (CPU vs GPU side rendering)

2.1 Interpreted vs. Compiled

The debate between the intricacy of an interpreted vs compiled languages is convoluted. The biggest point being made is that a language is not inherently either one or the other, but rather it's the implementation. [3]

The main difference lays in the way the software run. A compiled "implementation" is run by translating the entire source code into binary code which is natively executed by the machine. This means there is no middle layer between the instruction and execution which makes it extremely fast. An interpreted "implementation" executes one instruction at a time (compiled line by line) making it more flexible but at the cost of performance. [24]

The intricacy arises in special implementations of pre-existing language. A good example being "Cython" [4], an extension for Python (widely known as an "interpreted" language) which allows the translation into C or C++ code which is then compiled. There are also third party interpreted implementations of the C++ language blurring the line further. [12]

Due to my choice of the standard implementations off Python and C++, for the remainder for this report I will refer to the Python language as interpreted and C++ as compiled.

For this project I wanted to measure the difference in performance of compiled compared to an interpreted language used for the same purpose. I chose Python and C++ respectively to represent the two types. Initially, I was going to use C instead of C++ as I have more experience with the former however C++ is an object-oriented (OOP) language (as is Python) so it would be a more fair comparison if I chose to utilise objects and classes for my code-base.

C is a procedural language meaning it follows a "top-down" design.[24] C++ was invented as an extension mainly in order to add object oriented features to C to organise data easier whilst retaining the speed. Seeing as Python was initially made in C, I believe comparing C++ to Python for the same purpose would be an interesting investigation into the difference between high and low level languages. Having already worked with C in the past, I also wanted to use this opportunity to learn more about C++ and it's potential advantages.

Another difference between Python and C++ worth mentioning that Python is dynamically typed whereas C++ is statically typed. The difference between a dynamically typed and statically typed language is that a dynamically typed language perform type-checking at run time whereas statically typed languages perform type-checking at compiling time. [23]

This essentially means that in a statically typed language such as C++, variable data types must be specified ahead of time which typically results in a more optimised program. Another advantage of a statically typed language is that a lot of bugs can be caught out early on during the compiling stage. This makes it easier the bugs easier to track down and fix as by nature compiling errors are easier to deal with than run-time errors.

Memory allocation is another way in which the two languages differ. Python uses "garbage collection" whereas C++ does not.[19] This means that the memory must be manually managed within C++ code, typically with the use of pointers. If poorly implemented, this can be a cause of memory leaks in the program however with the proper usage it can be used to optimise the software better than Python's automatic management system.

It is important to state that this comparison is not a general comparison of the languages but rather a comparison for this specific use case. There is no doubt that C++ will outperform Python in this case, the question is by how much and is it worth the longer development process. This is a common question asked in industry where the time spent in development is an important factor to consider. Oftentimes, a product would be prototyped using a high-level language such as Python and then implemented in a more reliable language such as C++. I would like to carry out this investigation to learn about the steps needed to carry out performance analysis.

2.2 Brute force vs FFT Convolution

As shown in Equation 4, the solution to the equation requires taking a 2D laplacian which can be calculated using a convolution filter. As this calculation is performed on a matrix of discrete values, we need to perform a discrete convolution with a kernel that approximates the second derivatives of the laplacian.

Essentially, what this operation calculates is the difference between the value of the cell and the average of the cells around it. This term is responsible for modelling the diffusion of the chemicals multiplied by the corresponding diffusion coefficient. We can find the 3x3 laplacian convolution filter via a first principles approach Equation 6.

$$\nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2} \tag{5}$$

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h} \tag{6}$$

Through the method of first principles, assuming the smallest possible value $h = 1$ we find that.

$$\begin{aligned} f'(x) &= f(x+1) - f(x) \\ f'(x+1) &= f(x+2) - f(x+1) \end{aligned} \tag{7}$$

Then to find the second derivative we do the following.

$$\begin{aligned} f"(x) &= f'(x+1) - f'(x) \\ &= f(x+2) - f(x+1) - f(x+1) + f(x) \\ &= f(x+2) - 2f(x+1) + f(x) \end{aligned} \tag{8}$$

As the method introduces a shift of 1 to the equation we can take this out by shifting the output to the left. We then do the same for y giving us our result which can be interpreted as a convolution matrix. These two combined

$$\nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y) \tag{9}$$

This is equivalent to writing this as a matrix below.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

This is a 2D laplacian convolution filter. Filters like these are used very often for image processing. This particular filter or "kernel" is used for edge detection and convolutional neural networks. For our purpose however, we must smooth out this function using a gaussian filter to represent the diffusion correctly. The end result is a Laplacian of Gaussian (LoG) function which takes the form.

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{10}$$

By sampling this function we arrive at a convolution filter used in the Gray-Scott model below.

$$\begin{bmatrix} 0.05 & 0.2 & 0.05 \\ 0.2 & -1 & 0.2 \\ 0.05 & 0.2 & 0.05 \end{bmatrix}$$

Note that for a convolution filter to work, the sum of the filter must be equal to zero in order for the values of the matrix it is applied to don't exceed the maximum value. The center of the filter in negative, this means that when it is applied to a pixel of a high value with low-valued surrounding pixels, the resulting value will end up being negative and the "concentration" will "diffuse" into surrounding pixels.

In this report we will explore two different types of convolution, "Brute Force" and Fast Fourier Transform (FFT) Convolution. The former relies on "scanning" the target image/ matrix with the filter one pixel at a time as shown in Figure 5 below.

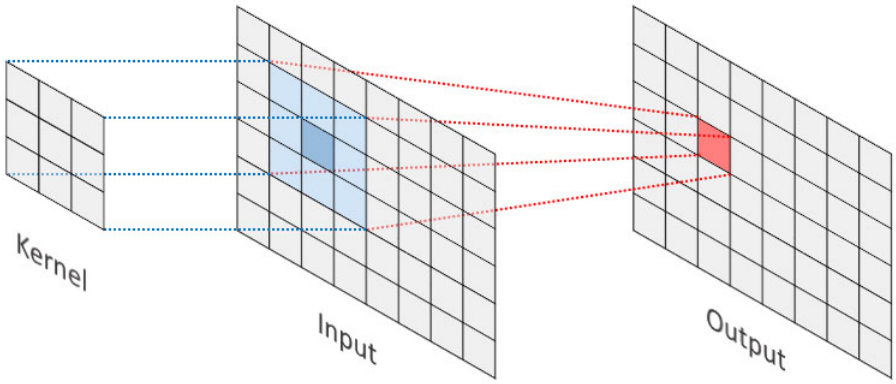

Figure 5: Illustration of how 2D convolution is performed. The values of the kernel are multiplied by the values in the input matrix (shown in blue) the sum of which is given as the output (shown in red).[32]

In order to implement this in code, we must create 4 nested for loops. For each combination of input coordinates and kernel coordinates we multiply the values and find the sum of them. In terms of time complexity using the "Big-O" notation (which assesses the run-time as a function of the input size N) 4 nested loops give us $O(N^4)$. This means that as we scale the "input" matrix, the time to run the program scales proportionally to the power of four. This means that if we were to double the size of the input, the time to run the program would theoretically increase 16.[1]

In reality the "Big-O" notation is not an accurate measure of real run-time speeds. The actual time taken to run the program is measured by T(N) experimentally. O(N) only includes the contribution of the highest growing term in T(N) without any constants. This allows it to be a general description of the algorithm that is system independent. For example, let's say that we have a program which finds the sum of all of the elements in a function. I have written this in python for simplicity.

```
import time

# Initialise array
array = [[5,2,7],
         [9,2,1],
         [1,2,8]]

# Parameters for testing purposes
total_time = 0
n_average = 100

# Function to sum all the elements in the function
def sum_array(array):
    total = 0           # O(1)
```

```
    for row in array:   # N^2
        for i in row:
            total += i  # O(1)
    return array        # O(1)


#Function to run the function n_average times and take an average
for i in range(n_average):
    start = time.time()
    total = sum_array(array)    # total is assigned to variable to avoid caching value
    end = time.time()

    total_time += (end-start)

#Prints average time to run the given function
print(total_time/float(n_average))
```

Looking closer at the function we gather that the operations such as setting, updating and returning a variable is a constant time operation O(1). This just means the operation takes a constant time to run. The double-nested for-loop however, is run N^2 times as there are N rows and N elements in each row. From experiments, using the "time" python library we find that setting a variable and returning a function takes $0.121\mu s$ on average (1.211e-07s). From this we can calculate the time taken to run the program T(N).

$$\begin{aligned} T(N) &= O(1) + N^2 \cdot O(1) + O(1) \\ T(N) &= 0.121(2 + N^2) \end{aligned} \tag{11}$$

Now, if we plug in $N = 3$ we should theoretically get $T_t(3)$=1.331μs. When running the same time utility on the whole function, we get an experimental $T_e(3)$=1.431μs (1.431e-06s). This is higher than the predicted by the T(N) function due to the overhead of creating the for loops in the first place as well as some more lower level operations that were unaccounted for.

Equation 11 gives us the Big-O of $O(N^2)$ as the N^2 is the fastest growing term in the equation and the coefficient is system specific. We can test this by giving the program different sized arrays and testing the performance. The following piece of code shows this in practice.

```
import time
import numpy as np
import matplotlib.pyplot as plt

# Parameters for testing purposes
total_time = 0
n_average = 100

# arrays to hold the size of array and the average time for the corresponding array
sequence = range(1,100)
avg_times = []
```

```
# Function to sum all the elements in the function
def sum_array(array):
    total = 0           # O(1)
    for row in array:   # N^2
        for i in row:
            total += i  # O(1)
    return array        # O(1)

for N in sequence:
    #Function to run the function n_average times and take an average
    for i in range(n_average):
        start = time.time()
        total = sum_array(np.full((N,N),1))    # assignment to avoid caching
        end = time.time()

        total_time += (end-start)

    avg_times.append((total_time/float(n_average)))

x = list(sequence)
y = avg_times

plt.plot(x,y,"r.")
plt.title("Average run-time of function against array size")
plt.xlabel("Size of input array N")
plt.ylabel("Average run time T (s)")
plt.show()
```

This code leads to Figure 6 shown which shows the quadratic nature of the performance of the system with scale. Once again, this is not an big-O plot as the values are system specific.

For this specific purpose, the scalability of the system is important as a larger system lets us explore the parameter space (Figure 4) in more resolution. As it stands, for "brute force" convolution shown in Figure 5 we need 4 nested for loops. Following the previous discussion, this would result in a time complexity of $O(N^4)$. This is extremely inefficient, if we are planning on rendering the simulation as it evolves, this would slow the time at which we could update the image.

To counteract this limitation in the implementation by using the convolution theorem. This will require us taking the two-dimensional Fast Fourier transform of the kernel $\mathbf{K}$ and the input matrix $\mathbf{M}$. Then, we would perform an element-wise multiplication on which we apply an inverse Fourier transform to obtain the convolution of the K and M. This, on the other hand, would lead to a time complexity of $O(N^2 log^2 N)$ which would be a big improvement and allow us to render the simulation at a higher framerate.

To test this on my implementations, I will plot a T(N) plot for both methods, spatial domain convolution "Brute force" as well as frequency domain convolution (FFT).

There has been some exploration of 3D reaction diffusion, this method would also be possible in 3 dimensions, as the time complexity of convolution scales with the dimensions, the Big-O would be $O(N^6)$ for the brute force method and $O(N^3 log^3 N)$ for the FFT.

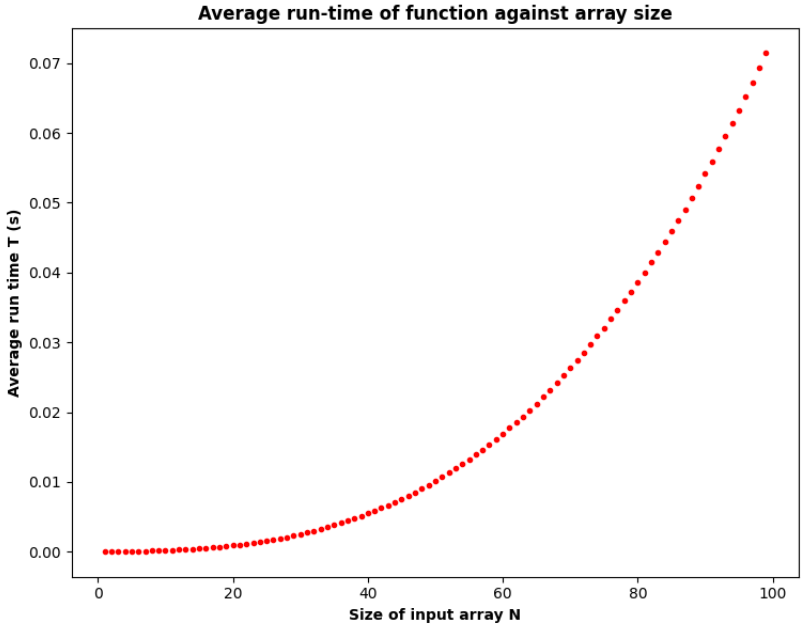

Figure 6: Average run-time T(N) of function summing all elements of an array against size of input array (Averaged over 5 individual runs)

2.3 CPU vs. GPU Side Rendering

After the correct algorithm is implemented that performs the calculation in a satisfactory amount of time, the simulation must be visualised. Technically, we could write the simulation to a file and save it as an animation, to get a more in-depth look and to allow for further features such as real time interaction with the simulation, rendering it real time would be to go. Additionally, if our purpose is to make the software run faster it would only make sense to reap the benefits of it.

As this program is essentially composed of a maths engine and a renderer, it is a constant battle between which one is slowing down which. As our goal is to improve the maths engine, the renderer must be able to catch up with it.

When it comes to rendering graphics, it is better to utilise the GPU as it is its purpose. This is due to the design of the GPU compared to a CPU. CPUs process information sequentially in series whereas GPUs have thousands of small cores and process information in parallel. A diagram of this is shown in the figure below.

This makes operations like drawing 2,073,600 to the screen (Full HD) much easier for a GPU as opposed to a CPU. In order to make the code run on the GPU, we need to send so called "shaders" to the GPU first so that it can parse and preform calculations on the data that we feed it. [27] Most modern GPUs already have a lot of basic shaders pre-installed on them however for more complicated tasks, shaders are written manually by developers. To interact with the GPU, an Application Programming Interface is used. The most commonly used and efficient one is OpenGL in which shaders can be written in the GLSL language which is a shading language based on the syntax of C. [18]

OpenGL sends data to be calculated to the GPU in the form of memory buffers. This information is then processed by a pre-programmed shader. Sometimes a bottleneck arises

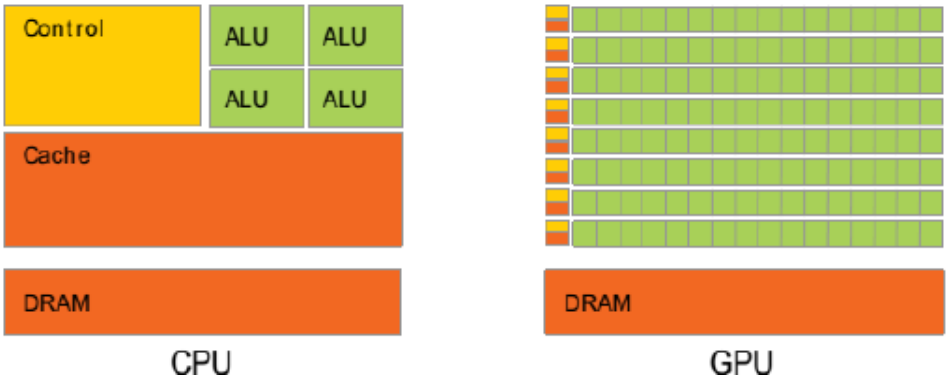

Figure 7: Architecture of CPU vs GPU [30]

between this transfer of data however this is easily addressed with caching (for our purpose this wont be an issue).

The rendering process in the OpenGL API (Also applies to others such as DirectX, Vulkan, etc.) is the following. [26]

1. **Raw Vertices -** Raw vertex points "primitives" in space
2. **Vertex Shader -** Assembles the vertex points and turns them into shapes. Using the vertex shader we can also preform calculations on the vertex memory buffer.
3. **Fragment Shader -** This stage essentially "colours in" the shape generated by the vertex shader by interpolating values following specific instructions in the shader.
4. **Frame Buffer -** This is our final product that is then sent to the screen

In a rendering example where we want to render a simple triangle of a certain colour we must make an array of three vertices, pass this to the vertex shader which is called once per each vertex (3 times). After calculating the shape, we move onto filling it in with a specified number of pixels. Depending on the resolution, the fragment shader will be called thousands of times for each pixel inside the shape. This is where the high number of calculations happen making it much more efficient to carry out this process using a GPU.

If time allows and depending how the rest of the project comes along, I will try to build this on top of the C++ implementation as I believe this one is going to be the most efficient and it will make most sense to spend time essentially creating a 2D renderer for the reaction diffusion simulation.

I will attempt to use the GLFW GLEW OpenGL C++ wrapper to implement this process onto my code by passing the calculated matrix elements from the simulation engine to the vertex buffer. I will compare this to CPU rendering using matplotlib or OpenCV depending on which one has a better implementation in C++. I have no previous OpenGL programming experience but I would like to use this opportunity to learn more about this field to gain more insight into computer graphics as they are so prevalent in our lives.

3 Code

I decided to investigate this algorithm for my project as I previously tried creating it in my spare time. I got as far as successfully simulating the system in python using the brute-force convolution method. Although the program worked, it was limited to matrices of around 200x200 pixels on my laptop. Larger matrices would lead to a significant performance decrease.

Naturally, first thought was to implement the algorithm in a lower level language improving performance for the reasons mentioned in Section 2.1. However, I didn't believe that it was entirely down to the choice of the language, the algorithm was to be revised. After drawing the comparison of the method with my knowledge in image processing (Gaussian Blur, Sharpening Filters) and the 2D convolution theorem learned in the 1st year module PHY1026 "Mathematics for Physicists" I gathered it may be worth experimenting with.

Lastly, I used the matplotlib python library to visualise the results. When implementing this, I knew it is going to be a significant bottleneck as despite its animation feature, matplotlib is a plotting library not a renderer. This meant that after making the algorithm more efficient, the bottleneck would become the rendering engine. From my initial prototype, I was limited to one frame per 400ms by the animation loop of matplotlib as it took that long to render that frame. This was naturally not as significant for smaller matrices however it meant that scaling became a problem. After some preliminary research into rendering frameworks and packages, I came across OpenGL. Even though it would be ambitious of me to try make the algorithm faster on three different levels having little knowledge of the language it would give me a goal to aim for.

This drew me to the conclusion that I will split the project into 3 parts. "Python vs. C++", "FFT vs Brute Force Convolution" and "CPU vs GPU Rendering" giving me enough content to in three different domains. I wanted more variety in my project to give myself the opportunity to focus on areas of interest the most letting me develop in skills I am most passionate about.

In the next section I will describe my results for each one of the sections along with the challenges faced along the way.

The code is available in a repository on my github page seperated into individual sections corresponding to the sections in this report.

github.com/natan-sz/independent-reaction-diffusion

4 Results

4.1 Python vs. C++

As mentioned before, I had a working implementation of this program in python so at first glance, this would require rewriting the program in the other language. Initially i wanted to carry out this in C however i decided upon C++ due to its more extensive libraries and the OOP versatility that I was used to from python. As the "Scientific Programming in C" second year module was my last contact with a C language, I decided to refresh my knowledge by attempting 11 programming languages from `www.hackerrank.com`. These can be seen in the github repository under "4.1 - Python vs. C++/tutorial".

Although these tutorials were useful for me to refamiliarise myself with the syntax they were clearly not enough for this project. I then started learning more about the ways in which I would be able to manipulate the matrix in order to update the values. The way described in the exercises and in the second year module was through arrays and pointers.

Via more independent research, I came across std::vector, a "sequence container that encapsulates dynamic size arrays" [29]. This container is recommended to use when dealing with rendering engines as the values are easier to pass. std::vectors can also be passed by value instead of by reference meaning I will be able to compare passing matrices to the function by reference and by value and draw the comparison to python which only uses passing parameters by value.

At first , I began with passing the 2D std::vector by value, this led to similar performance to the python code hinting that manual memory management may be the big advantage of C++. After switching to passing by reference, I saw a significant boost in speed. This is because passing by value requires the program to create and delete a new array for call of the function in question. After this increase in performance, I researched ways in how to utilise the std::vector class to its full potential.[2] Some rules to abide by when using this container class, is making sure that the size of the vector is determined beforehand, using assignment instead of the "at( )" method and ensuring the free memory after use.

As I decided not to use and IDE for this project to learn more about development using vim with command line utilities on Linux. In order to compile my program, i created a shell script in order to compile my code with g++ and the correct flags for the packages I was using. Although at first I felt very puzzled by the structure of C++ projects seeing how many dependencies there are, I managed to learn more about how header files, dynamic and static linking works. I ended up not using these for this project as the project felt quite overwhelming however it is a good starting point for potential future projects of mine. If i had a chance to do this project again, I would most likely use an IDE so I could focus more on the code and less on the requirements needed to run it.

For the benchmarking of the brute force implementations in C++ against the one in Python, I wrote a script that runs each one of the programs with varying inputs of array sizes N (inputted via "argv[ ]" from standard libraries). Both of the implementations were run and timed 5 times for each array size, this was then plotted and a line of best fit was found. The results of this can be seen in Figure 8.

For this experiment, I set the number of iterations of the simulation to 10 to save on computing time as anything above that would have given the same outcome.

As shown by the plots above, we see that the same algorithm in C++ is significantly faster (50.4 times faster on average) than in Python. This is not quite the same as the 400 times increase in speed often mentioned in discussions however this is most likely to the flaws of my personal code.[11] As previously discussed in Section 2.2, the time taken increases quadratically with the size of the array simulated due to the number of loops used. An order four polynomial has been fitted to each one of these to show the difference.

It is important to mention that the difference also arises from the time it takes to initialise the program. This obviously gives the C++ version an advantage as it was pre-compiled beforehand whereas Python being an interpreted language took longer as it had to execute commands in real time. I did notice however, that in Python, using the numpy library for arrays boosted the speed significantly. This is because a lot of the numpy library operations are implemented in C.

To test the performance difference between the languages I specifically made sure to mea-

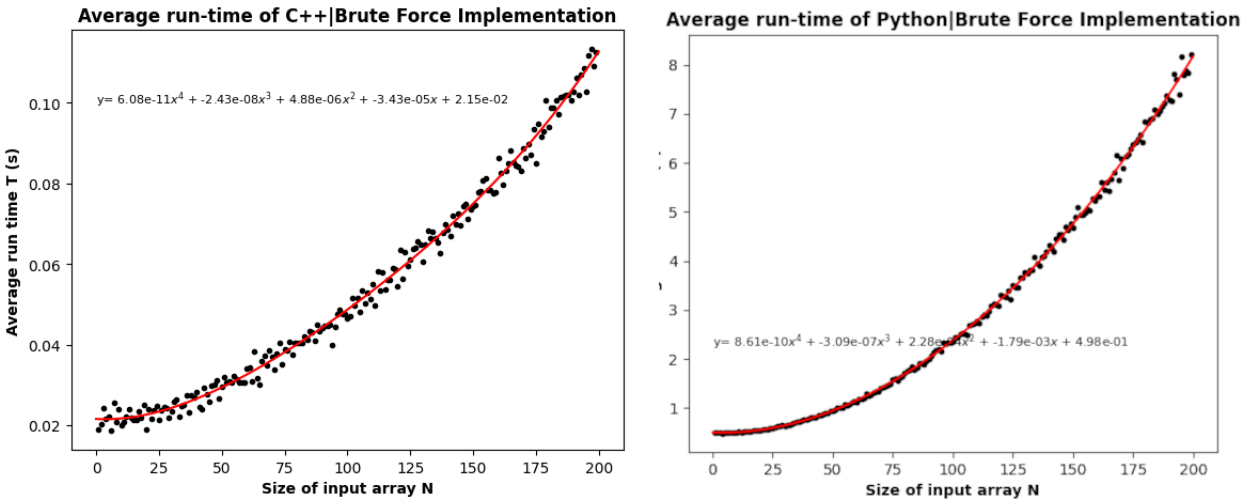

Figure 8: Plots showing the T(N) (average run-time). Both programs were run 10 iterations for a range of matrix sizes (NxN) and averaged over 5 runs. The run-time refers to running the entire program from start to finish.

sure the entire run-time of the program itself and not the functions as this is a more practical comparison.

These tests were carried out on my computer one after the other with as many background processes turned off to reduce noise. Before doing this, there were many more outliers in the data as other background applications were trying to access the memory. As is very visible from the plots, the C++ data is much more spread out than the Python equivalent. This is due to the precision of time measurement being much more apparent in the case where the run times are smaller.

After running this benchmarking script (which runs the program one thousand times) multiple times on the C++ code, I quickly found that the method "clear()" which was meant to free the memory at the end of the code did not do it's job. I knew this when my computer crashed with artifacting on the screen in a heap buffer overflow. I didn't run into that problem before which only shows me how important testing software before using it is. I then promptly fixed this by using the "shrink_to_fit()" method instead which fixed this issue.

Overall, this clearly shows that C++ is much more suitable for brute force calculations. Main factor behind this is the memory management. Despite the optimised numpy arrays, C++ still beats the performance as expected. This is not an exhaustive comparison of the two languages for this purpose as there is still a multitude of improvements that could be done to either one however, the disparity between the same general implementation is significant.

4.2 FFT vs Brute Force Convolution

Next improvement I attempted to make to the code is to implement the Fast-Fourier Transform Convolution method in order to reduce the Big-O notation of the code for it to scale better. After attempting to implement this in order for it to give the right output and be faster than the brute-force counterpart, I decided to opt for a implementation from the library OpenCV often used for video processing and object detection giving us one of the best

possible implementations of this algorithm.

This library is available for both C++ and Python meaning I can test the same implementation for the convolution function in both programming languages without coding it from scratch which the time-frame would not allow me for. OpenCV features a "filter2D()" function which interestingly uses direct convolution (Brute Force) for smaller matrices and FFT convolution for larger ones. This is a very good optimisation which takes into account the difference in Big-O of the two algorithms.

In short, the function switches between these as for smaller N, the function N^2log^2N (FFT) grows faster than N^4 (Brute Force) meaning it would take longer to run. For larger N, N^2log^2N grows slower than N^4 which is why the algorithm switches when it becomes more beneficial to do so.

To compare the python code, I was able to isolate the function away from the variable initialisation process by timing the function within python itself. This was done similarly to the benchmarking code shown in Section 2.2. This let me compare the individual functions giving a fairer comparison.

For this experiment, the update() function was run only once in the program itself but was then run 5 times by the benchmarking script and averaged to obtain the averaged time taken to run the function. The difference in the algorithm lays in the update function which either calls the conv_2D() or the cv2.filter2D() function.

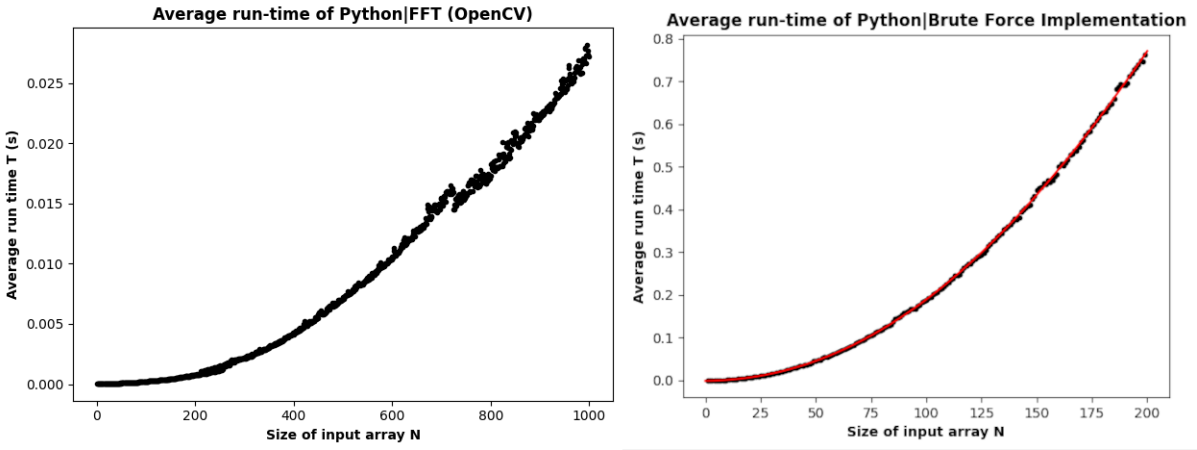

Figure 9: Average run-time of a single call of the update() function in Python which updates the matrix by running convolution and implementing the result in the reaction diffusion equation. (Left:FFT, Right:Brute)

As seen from the left side of Figure 9, we see the discontinuity discussed earlier at a array size of 700x700 with a kernel size of 3x3. This is the point at which it is more efficient to use the FFT algorithm as opposed to the brute force one. I am glad I used the library for this purpose as it led me to finding this out empirically. If I were to implement this myself I would not have obtained this level of efficiency.

Another piece of insight from this comparison we see is how much more efficient the OpenCV algorithm is to my own. If you look closely, the Brute force algorithm on the right was only run between N=1 and N=200. I decided to stop the measurement at that point as it became extremely slow to calculate. This means that the OpenCV algorithm is faster at

performing the calculations for a matrix of 1000x1000 elements than my one is with a matrix of 200x200.

It is also clearly apparent that the Brute force algorithm is a clear $O(N^4)$ algorithm as it follows a very strong correlation similarly to the one in Figure 6.

For the C++ code, I was unable to isolate the function as well as python and decided to run the whole program instead due to time pressure. This comparison will not be fair as the C++ benchmarking is disadvantaged however I am confident that the speed of the implementation will overcome this.

Additionally, when rendering the C++ OpenCV matrices using the inbuilt function cv::imshow(), the output did not show a reaction diffusion pattern. The input that I gave it (square of chemical A in the middle) was blurred gradually meaning the diffusion worked however the reactions did not take place despite them being in the equation. After numerous hours trying to fix this, i decided to focus more on testing the performance than making a pretty picture. From what I gather, the problem laid in the representation of the values calculated in the correct range of values for cv::imshow() to interpret however even normalising the matrix did not help.

The following is the result of benchmarking the two different algorithms (OpenCV and Brute Force) in C++.

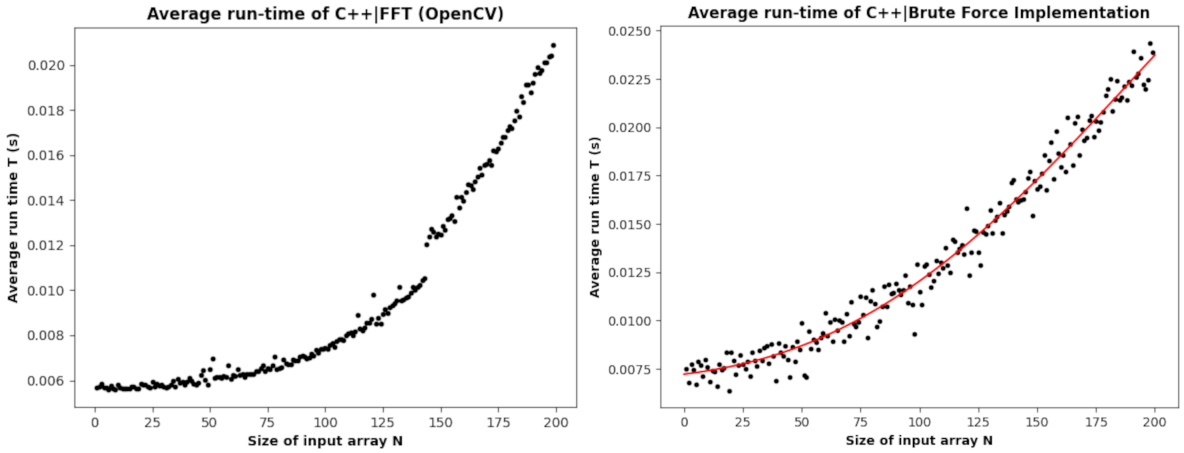

Figure 10: Average run-time of a single call of the update() function in C++ which updates the matrix by running convolution and implementing the result in the reaction diffusion equation. (Left:FFT, Right:Brute)

Unsurprisingly, the OpenCV Fast Fourier Transform algorithm is faster than the brute force algorithm in C++ in the same way as the previous example in Python. The FFT OpenCV graph has the same distinct discontinuity as Figure 9 which is the same algorithm switching concept as before however this time it seems to have happened for a smaller matrix with the same kernel.

From this graph, the run time of C++—FFT is actually slower than the Python—FFT code. This is for two main reasons. As mentioned previously, the C++ benchmarking is based on running the entire program from start to finish including the initialisation stage of setting parameters. This stage is very time consuming in python which is why I decided to omit

this in the previous comparison. Given more time, I would test the time taken to run just the function call in C++, this would require a benchmarking program with plotting abilities. My prediction is that the C++—FFT and Python—FFT implementations would not differ greatly, as they both rely on the same library. The reason the Python—FFT version was so much faster than its counterpart is because the filter2D() function is made in C therefore the source code for the Python and C++ filter2D() is the same leading to similar performance. Additionally, in the C++ code I was forced to use the OpenCV package storage containers cv::Mat which I am sure hindered the performance as well.

4.3 CPU vs GPU Rendering

Having already spent a lot of time on the other sections, I didn't have a lot of time to finish this part of the project. Despite spending long hours trying to grasp the concept of shaders and thousands of abstract GLFW functions after dealing with library dependency issues (GLEW must be imported before GLFW) (I was not using an IDE) I only managed to render a simple triangle in a window essentially parrot coding. My several attempts at this can be seen in the /4.3 - CPU vs GPU Rendering section on the GitHub page.

Here is a snippet of code of what I did to create a vertex buffer with the needed information to make a triangle.

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};

// Assigns VBO
unsigned int VBO;
glGenBuffers(1, &VBO);

// Binds Several buffers together (This creates the buffer)
glBindBuffer(GL_ARRAY_BUFFER, VBO);

// Copy vertex data into the buffer
// Use GL_DYNAMIC_DRAW for frequent access/change
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

First, i create the points at which the vertices need to be drawn, then I create an unsigned integer object named VBO and then pass it to the glGenBuffers() function to assign this as the 1st memory buffer. Next we call glBindBuffer() to create a memory buffer compatible with OpenGL shaders. After this, we copy the "vertices" data into the buffer specifying the type, size and priority of its access ready to be sent to a GPU.

When first attempting this section, I researched into how I could convert the values calculated in the two-dimensional std::vector arrays into pixels on the screen with a window. I found that the way to do this was to use the glGenTextures() command by passing through a texture in the correct format of UNSINGED_FLOAT. This proved to be much harder than expected. This is as far as I got with creating the texture buffer however projecting it on a surface in space proved itself to be very difficult.

```
// Creates OpenGL texture object
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);

// Sets texture wrapping parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// Sets texture filtering parameters
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_FLOAT, 1, 1, 0, GL_FLOAT, GL_UNSIGNED_FLOAT, nullptr);
```

I believe if this was the sole purpose of my project I would be able to take it much further however this part relied on the last part being completed. I will definitely revisit this subject in the future when my C++ skills improve a little bit more as I still find this fascinating and want to get to a smoothly animated reaction diffusion built entirely from very low level up.

5 Conclusion

In conclusion, I found that C++ is a faster programming language when it comes to brute force calculations compared to python due to it being a compiled, statically typed language with manual memory management. On the other hand, Python is a useful tool for prototyping due to its simple syntax and well optimised libraries. I believe that in order to minimise development time, the best way to develop a proof of concept program in python and optimise it as much as possible using all of the libraries possible ensuring you understand the algorithms behind them. Then, when you have a piece of well optimised code, implement it fully in C++.

With the constant development of new and improved python libraries with components made in C, the gap in run-times will keep on decreasing, however, it will never reach the same level due to the overhead Python has in importing modules and passing variables by value. Even if the difference in speed is in the realm of milliseconds, it can quickly add up to hours if ran a million times justifying the time spent in development.

I was not able to delve much into the field of rendering images in real time I made a good start for future development when I have more time on my hands. C++ and OpenGL are much more involved topics of interest than what 13 weeks allows for whilst having many other commitments at hand.

If I had the opportunity to do this project again, I would take a more coordinated approach now that I know how to handle particular aspects of this project. There were several moments in this project when I spent a long time on things that later proved themselves to be trivial. Independent study has its benefits however it also meant that simple obstacles that could have been solved quickly by a brief discussion took much longer than needed. One thing I regret is my lack of communication with the module leader as it would have helped me overcome these challenges much quicker and come up with unintuitive to me ideas. I also ended up not doing error handling or utilising much of the OOP features which would help a lot now that I see their benefit.

That being said, I would consider this project a partial success. Due to my ambitious goals in the time-frame combined with unfamiliarity with C++ and OpenGL I did not manage to

complete all of the parts to my best ability. Despite this, I believe not giving myself a ceiling propelled me into learning more in a shorter period of time. There is so much more to this project than I would possibly be able to write as there are so many different avenues for improvement. Further work in this area can prove to be fruitful as there is still a lot of things to learn from it.

Despite it being difficult, a large quantity of content, combined with the freedom to study what I deem relevant let me explore areas of the field by myself making it very rewarding. I learned about the mechanics behind pattern formation in nature, benefits and drawbacks of low and high-level programming languages, applying two dimensional convolution in practice, benchmarking software, optimisation alongside various other things along the way. I made the project more difficult for myself by not using an IDE and using VIM as my text editor of choice. Although frustrating at times, It helped me learn more key-bindings and shortcuts to use when working with multiple files optimising my workflow further. In the future however, I will most likely opt for an IDE such as Visual Studio which would make package management easier. This would also allow me to port this to other operating systems.

Additionally, this project helped me change the way I code, I now think twice when writing a deep set nested loop when working with high dimensional arrays. I also learned that logging outputs to a file in case of a system crash and to always back up my code to a remote repository such as GitHub in case of data loss. I will use and improve the benchmarking script I made to obtain the time complexity of future projects.

As they say, "Shoot for the moon. Even if you miss, you'll land among the stars." even though I don't believe I landed among the stars, I ended up somewhere in low earth orbit which let me gain a better perspective on the subject.

References

[1] Gayathridevi .S and Dr . "An abstract to calculate big O factors of time and space complexity of machine code". In: July 2011.

[2] *6 Tips to supercharge C++11 vector performance.* URL: https://www.acodersjourney.com/6-tips-supercharge-cpp-11-vector-performance/.

[3] Ernest Ampomah, Ezekiel Mensah, and Abilimi Gilbert. "Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages". In: *Communications on Applied Electronics* 7 (Oct. 2017), pp. 8–13. DOI: 10.5120/cae2017652685.

[4] Stefan Behnel et al. "Cython: The Best of Both Worlds". In: *Computing in Science and Engineering* 13.2 (2011), pp. 31–39. URL: http://www.computer.org/portal/web/csdl/doi/10.1109/MCSE.2010.118.

[5] Lorena Caballero, Bob Hodge, and Sergio Hernandez. "Conway's "Game of Life" and the Epigenetic Principle". In: *Frontiers in Cellular and Infection Microbiology* 6 (June 2016). DOI: 10.3389/fcimb.2016.00057. URL: https://doi.org/10.3389/fcimb.2016.00057.

[6] Sergey Dvinin, Vladimir Bychkov, and Igor Esakov. "DC Discharge Description with Nonlinear Reaction-Diffusion Equation in Air Mixture". In: *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition.* American Institute of Aeronautics and Astronautics, Jan. 2009. DOI: 10.2514/6.2009-1051. URL: https://doi.org/10.2514/6.2009-1051.

[7] R. A. FISHER. "The wave of advance of advantageous genes". In: *Annals of Eugenics* 7.4 (June 1937), pp. 355–369. DOI: 10.1111/j.1469-1809.1937.tb02153.x. URL: https://doi.org/10.1111/j.1469-1809.1937.tb02153.x.

[8] V E Fortov and I V Lomonosov. "Ya B Zeldovich and equation of state problems for matter under extreme conditions". In: *Physics-Uspekhi* 57.3 (Mar. 2014), pp. 219–233. DOI: 10.3367/ufne.0184.201403b.0231. URL: https://doi.org/10.3367/ufne.0184.201403b.0231.

[9] T. Galochkina, M. Marion, and V. Volpert. "Initiation of reaction–diffusion waves of blood coagulation". In: *Physica D: Nonlinear Phenomena* 376-377 (Aug. 2018), pp. 160–170. DOI: 10.1016/j.physd.2017.11.006. URL: https://doi.org/10.1016/j.physd.2017.11.006.

[10] Diego A Garzón-Alvarado and Angelica M Ramırez Martinez. "A biochemical hypothesis on the formation of fingerprints using a turing patterns approach". In: *Theoretical Biology and Medical Modelling* 8.1 (June 2011). DOI: 10.1186/1742-4682-8-24. URL: https://doi.org/10.1186/1742-4682-8-24.

[11] *If you have slow loops in Python, you can fix it... until you can't.* URL: https://www.freecodecamp.org/news/if-you-have-slow-loops-in-python-you-can-fix-it-until-you-cant-3a39e03b6f35/.

[12] *Interpreted C++*. https://www.jmoisio.eu/en/blog/2020/01/05/interpreted-cpp/. (Accessed on 01/12/2021).

[13] K. Ito, Takiko Aoki, and Takuma Higuchi. "Fingerprint Restoration Using Digital Reaction-Diffusion System and Its Evaluation". In: *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences* (Aug. 2003), pp. 1916–1924.

[14] Michael Jeltsch et al. "Genesis and pathogenesis of lymphatic vessels". In: *Cell and Tissue Research* 314.1 (Oct. 2003), pp. 69–84. DOI: 10.1007/s00441-003-0777-2. URL: https://doi.org/10.1007/s00441-003-0777-2.

[15] Aly-Khan Kassam. *Solving reaction-diffusion equations 10 times faster*. Tech. rep. 2003, pg. 5.

[16] Kyoung J. Lee et al. "Pattern Formation by Interacting Chemical Fronts". In: *Science* 261.5118 (1993), pp. 192–194. ISSN: 00368075, 10959203. URL: http://www.jstor.org/stable/2881811.

[17] Colin B. Macdonald, Barry Merriman, and Steven J. Ruuth. "Simple computation of reaction–diffusion processes on point clouds". In: *Proceedings of the National Academy of Sciences* 110.23 (May 2013), pp. 9209–9214. DOI: 10.1073/pnas.1221408110. URL: https://doi.org/10.1073/pnas.1221408110.

[18] Ricardo Marroquim and Andre Maximo. "Introduction to GPU Programming with GLSL". In: vol. 0. Oct. 2009, pp. 3–16. ISBN: 978-0-7695-3815-0. DOI: 10.1109/SIBGRAPI-Tutorials.2009.9.

[19] Alex Martelli and Paula Ferguson. *Python in a Nutshell*. USA: O'Reilly Associates, Inc., 2003. ISBN: 0596001886.

[20] Hans Meinhardt. "Gierer-Meinhardt model". In: *Scholarpedia* 1.12 (2006), p. 1418. DOI: 10.4249/scholarpedia.1418. URL: https://doi.org/10.4249/scholarpedia.1418.

[21] C.V. Pao. "Dynamics of Lotka–Volterra competition reaction–diffusion systems with degenerate diffusion". In: *Journal of Mathematical Analysis and Applications* 421.2 (Jan. 2015), pp. 1721–1742. DOI: 10.1016/j.jmaa.2014.07.070. URL: https://doi.org/10.1016/j.jmaa.2014.07.070.

[22] J. R. A. Pearson. "On convection cells induced by surface tension". In: *Journal of Fluid Mechanics* 4.5 (1958), pp. 489–500. DOI: 10.1017/S0022112058000616.

[23] B.C. Pierce, B. C, and MIT Press. *Types and Programming Languages*. Mit Press. MIT Press, 2002. ISBN: 9780262162098. URL: https://books.google.co.uk/books?id=ti6zoAC9Ph8C.

[24] C. Publishing. *The C Programming Language, 3rd Edition*. C Publishing Series. Independently Published, 2019. ISBN: 9781691352326. URL: https://books.google.co.uk/books?id=9-pEyQEACAAJ.

[25] M. A. Quiroz-Juárez et al. "Generation of ECG signals from a reaction-diffusion model spatially discretized". In: *Scientific Reports* 9.1 (Dec. 2019). DOI: 10.1038/s41598-019-55448-5. URL: https://doi.org/10.1038/s41598-019-55448-5.

[26] *Render To Texture*. URL: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-14-render-to-texture/.

[27] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. 7th. Addison-Wesley Professional, 2009. ISBN: 0321552628.

[28] Karl Sims. *Reaction-Diffusion Tutorial*. 2013. URL: https://www.karlsims.com/rd-banner.jpg.

[29] *std::vector*. URL: https://en.cppreference.com/w/cpp/container/vector.

[30] Vajira Thambawita, Roshan Ragel, and Dhammika Elkaduwe. "To Use or Not to Use: Graphics Processing Units for Pattern Matching Algorithms". In: (Dec. 2014).

[31] "The chemical basis of morphogenesis". In: *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences* 237.641 (Aug. 1952), pp. 37–72. DOI: 10.1098/rstb.1952.0012. URL: https://doi.org/10.1098/rstb.1952.0012.

[32] Hugues Thomas et al. "KPConv: Flexible and Deformable Convolution for Point Clouds". In: *CoRR* abs/1904.08889 (2019). arXiv: 1904.08889. URL: http://arxiv.org/abs/1904.08889.

[33] Qianqian Zheng and Jianwei Shen. "Pattern formation in the FitzHugh–Nagumo model". In: *Computers & Mathematics with Applications* 70.5 (Sept. 2015), pp. 1082–1097. DOI: 10.1016/j.camwa.2015.06.031. URL: https://doi.org/10.1016/j.camwa.2015.06.031.