Implementation and Optimization of a Real-Time Wake Simulation Using Vortex Methods

Sebastian Preston-Jensen

School of Mechanical, Aerospace and Civil Engineering

University of Manchester

Individual Project

2016-2017

Abstract

The use of Discrete Vortex Methods were investigated for a simulation of vortex wakes capable of running in Real-Time. The simulation was split into two components, the convective scheme (responsible for calculating element velocities) and the temporal discretization scheme (responsible for iterating element positions). Three optimization strategies were investigated for the convective scheme, Biasing, Fixed cluster and dynamic cluster scaling. Dynamic Cluster Scaling was found to perform best on both metrics assessed; performance (in terms of computational time) and accuracy (relative to no optimization strategy). The ON^2 complexity was likely reduced to $O\log(N)$ for the dynamic clustering scheme, and element count was increased by an order of magnitude. Two alternatives to Euler's method for the temporal discretization scheme were investigated, Quadratic Position and Quadratic Velocity. Both schemes improved accuracy significantly for the test conditions (2 and 3 orders of magnitude respectively). The quadratic position scheme represented a slight increase in overhead (15%) whilst the quadratic velocity scheme presented a significantly large increase (270%). A Horseshoe Vortex was simulated using the Dynamic Clustering and quadratic velocity schemes (with 2014 elements), the vortex sheet produced the correct looking shape visually. Element Counts have been increased substantially and further study should focus on assessing the accuracy of the code against experimental data.

Acknowledgements

Dr Crowther deserves recognition for his guidance on this project which has been instrumental to the progress achieved and has not only served to aid with the project but fostered a genuine interest in the topic and made the project an enjoyable experience. I would also like to thank my housemates for still talking to me after the numerous unwarranted detailed explanations of the project.

Contents

List of Figures

List of Source Codes

1 Introduction

1.1 Overview

The aim of this project was to investigate the use of Discrete Vortex Methods (DVM) for simulations capable of running in real-time to simulate the vortex wakes produced by an aircraft during flight. The DVM was programmed in Unity using C#. The simulation was split into 2 main functions; the Convective Scheme and the Discretization Scheme. The convective scheme was optimized using an approach to Influence Tree optimizations using a reduced complexity approach to determining the influence tree by predicting the shape of the wake. Two differing order Implicit schemes were investigated for the discretization scheme.

The movement of air in the wake region left behind by an aircraft poses a potential risk to other aircraft. As such, wake regions are implicated as the main contributing factor in many aircraft crashes. The most notable of these crashes is Delta Air Lines Flight 9570, where a McDonald Douglas DC-9, a mid size passenger plane, rotated suddenly around its roll axis during landing, causing a collision with the runway. The crash was fatal for all occupants of the plane. The NTSB (National Transport Safety Bureau) investigation into the crash accounts the sudden rolling of the aircraft to the planes interaction with the wake region caused by a larger DC-10 aircraft that had landed moments before (NTSB 1973).

To avoid such crashes the NTSB mandated stricter wake-separation periods. Wake-Separation periods are waiting times between which subsequent aircraft landing on the same runway must wait. These waiting times represent a significant contribution to limiting airport capacity. Thus vortex wake regions are implicated in safety and efficiency concerns.

1.2 Motivations/Uses

The overview section provided an introduction to vortex wake and their implications to society. This section serves to outline the potential uses of a simulation of vortex wakes capable of running in real-time.

As previously discussed, vortex wakes pose a significant safety concern if a plane enters the wake region. One way to improve the safety in such situations would be to increase the pilots competency at handling such a situation. This is of utmost

important if the pilot is required to fly in conditions where vortex wake interactions are unavoidable, for example air-to-air refuelling. However training pilots in a real-life situation is both impractical and dangerous. Use of a flight simulator remedies both these concerns, however flight simulators employed in current pilot training implement limited and simplistic aerodynamic models (Rolfe and Staples 2010). The realism, and therefore usefulness, of such simulators could be improved via implementation of a simulation of the vortex wakes regions such as the one developed for this project. Flight simulators must operate in real-time to provide an immersive experience for the pilot, hence is it imperative that the simulation be capable of running in real-time.

Vortex wakes are not immediately apparent as they cannot be seen. Safety could therefore be improved if a pilot could visualize vortex wakes. Such a system be devised using the output of a simulation capable of running in real-time.

Wake-Separation periods are a major contribution to limiting airport capacity. Separation periods are mandated in order to ensure the safety of landing aircraft. Separation periods currently consist of imposing a set waiting period, dependent on the size of aircraft previously landed, until the runway may be used again. These separation periods are specified to be applicable to all situations an aircraft controller may experience, hence they often overestimate the time required for the runaway to become safe again. If separation periods can be reduced safely airport capacity can be increased. If seperation periods can be reduced aircraft may be able to spend less time circling the airport and thus burn less fuel, lessening their impact on the environment. If an air traffic controller was equipped with a simulation that predicted the wake of aircraft landing with a high enough degree of accuracy, wake separation times could be decreased.

If the simulation can be run quicker than real-time then the simulation could be used as a predictive tool. For example, air traffic controllers could predict the wake regions behind planes near their airports at points in the future and identify problematic interactions before they arise and adjust the bearings of planes accordingly.

1.3 Research Aims

The research questions to be investigated and answered are listed below

1. To program a Discrete Vortex Method with with no optimizations using a brute-force calculation approach

2. To program a Discrete Vortex Method using a "Long-Range cut off" or "Biasing" optimization and compare its performance and accuracy to the unoptimized case

3. To Pogram a Discrete Vortex Method using an Influence-Tree approach modified from astrophysics calculations and assess its performance and accuracy against the unoptimized case

4. To investigate the temporal discretization schemes used in real-time simulations and determine an ideal scheme to be used

The relevance of these research questions is discussed further in the literature review

2 Literature Review

Computational Fluid Dynamics (CFD) is a broad term encompassing a number of different numerical methods and techniques to solve fluid flow problems. CFD provides a practical alternative to physical models where analytical solutions are impossible or unknown. As such CFD is widely employed by industry and there are an estimated 250,000 purely CFD related jobs worldwide and an estimated potential market value of 2.9 Billion (Hanna and Parry n.d.).

Whilst CFD formally existed as an area of research as early as the 20's, major developments and the formative years of the subject are accountable to increased efforts during the cold war. Large increases in government funding accelerated CFD development motivated by the desire to improve missile technology (ibid.). As such these development mainly focused on improvements in understanding of supersonic aerodynamics and combustion phenomena (*Engineering simulation: past, present and future* 2017). Being subject to large government funding, notable developments in CFD, such as the "Particle-In-Cell" method stem from the work of Harlow and his team, T3, at the Los Alamos National Laboratory, a continuation of the Manhattan Project. These initial developments ran on supercomputers (Harlow 2003), making them prohibitively expensive to most industrial and academic applications, hence work on the subject through the 50's, 60's and to a lesser extent the early 70's is most entirely military based.

As the cost of computing power decreased and hence its accessibility increased, more varied applications and methods were introduced. Breakthroughs in CFD starting in the mid 70's going through the 80's are still in use and recognizable to anyone familiar with the subject, two notable exams are the k-epsilon turblence model described by Launder and Spalding in their 1974 paper "The numerical computation of turbulent flows" (Launder and D.b. Spalding 1974) and the SIMPLE scheme proposed by Spalding and Patankar in their paper "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows" (Patankar and D.b Spalding 1972). Incidentally it is Patankars book on Finite Volume methods "Numerical Heat Transfer and Fluid Flow" that is credited with popularizing the Finite difference and Volumes methods, notably due to their ease to program and wide applicability (Hanna and Parry n.d.). Programs spawned from this are still around, notably FLUENT and STAR-CD.

The finite difference and volumes methods pioneered during this era are still the most widely employed tools used (Hanna and Parry n.d.). They represents mesh based CFD methods where the spatial domain is discretized into grid segments and a series of equations formed at cell faces. This gives rise to two classifications of CFD techniques, mesh and mesh free. Mesh based CFD methods differ from Meshfree methods in that they provide a Eulerian description of the flow field. In a Eulerian description a specific location is observed and the flow at this given location is seen to evolve as time passes. In a Lagrangian description of a flow a certain parcel of fluid is observed and its position is also seen to evolve with time. In practice the difference between the two descriptions are exhibited in the use of either a grid based or particle based simulation. However each poses further specific advantages, particle based simulations tend to exhibit less numerical diffusion and naturally apply continuity (Li and W. K. Liu 2002).

However major differences are exhibited in the way they are calculated. Mesh based simulations often assemble a large matrix of simultaneous linear equations which are solved via an iterative method (Warsi 2006). However, for example, the DVM developed as part of this project, the calculation of the velocity of each particle is independent of each other (the equations are not simultaneous). Hence there are significant differences in their implementation. This leads to some approaches being more ideal for calculations on parallel rather than serial computing architectures.

As the availability of computing power increases, so does the scope of what may be achieved, this has led to increasing interest in real-time simulation. The development of Real-Time CFD has mainly been motivated for non-engineering related applications. Primarily for use in video games and special effects, both applications requiring results that look good, rather than provide accurate results. Coupled with the development of GPU's, an extensively parallel computing architecture, a number of realtime fluid simulations have emerged. The most notable of these being Nvidia Flex, a game physics engine. As the performance and accuracy of these simulations increase their scope as use as engineering tools likewise increases.

Real-Time simulation poses possible innovation in many engineering sectors. For example it streamlines the design process by eliminating the wait time between creating

a CFD model and the results from the calculations. This represents a shift from iterative based design to interactive design. However the uses of a real-time engineering grade simulation posses innovations more far reaching than the engineering process itself. For example work at the University of Leeds has successfully implemented a real-time CFD simulation of the heat generated in a data centre and implemented a predictive cooling system based on this (Khan et al. 2015). This example is notable and unique in that a system utilizing Real-Time CFD has been implemented. There are numerous other areas that development of Real-Time CFD poses advantages in, for example simulation of surgery (O'Connor et al. 2016), contaminant spread and control,

The method of CFD chosen to develop a Real-Time CFD code must achieve a level of performance adequate to obtain it's real time objective. Combined with the increase in potential performance offered by code running on a parallel architecture compared to a serial architecture for the same cost, the popularity of methods utilizing a Lagrangian description of the flow field have become popular. The two most prominent of these being the Lattice-Boltzmann Method and Smoothed Particle Hydrodynamics.

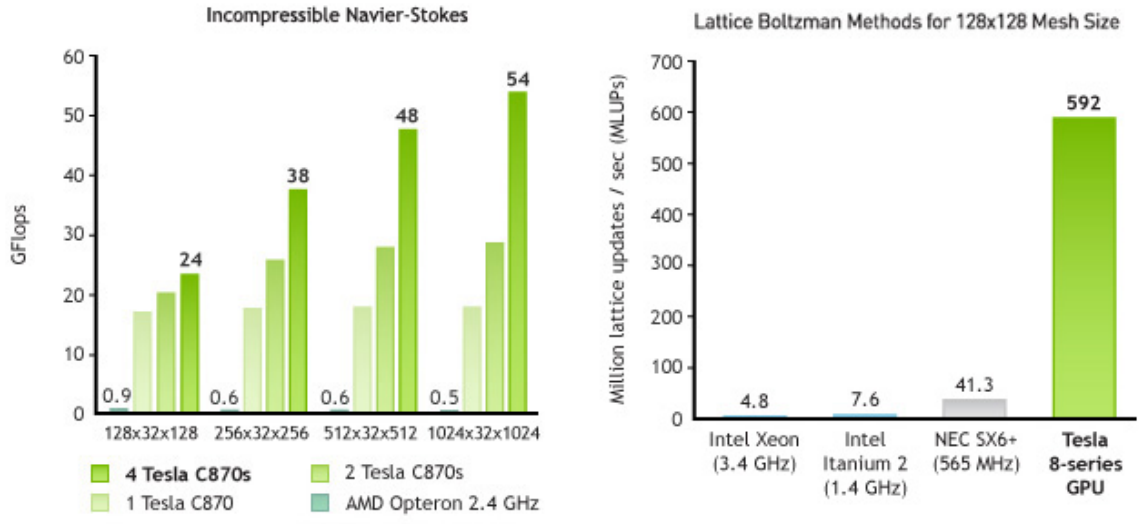

Figure 1: Comparison of Obtainable Computational Speed of Finite Volumes (left) and Lattice Boltzman (right) for a Series of CPUs and GPUs. From Nvidia (*Computational Fluid Dynamics (CFD) - GPU Applications* n.d.)

Figure 1 shows the results of an investigation by Nvidia into the performance of both Finite-Volumes (left) and Lattice-Boltzman (right) running on a selection of

CPU's and GPU's (*Computational Fluid Dynamics (CFD) - GPU Applications* n.d.). It is pertinent to note the author of this study, Nvidia, manafature GPU units and as such have a vested interest in the subject. Further, there is important data omitted from the study, namely the number of cores that each CPU unit is running on is not mentioned. Assuming the worst case scenario and each CPU is running on a single core out of a possible 4, and as such the CPUs tested can actually run four times as fast, the conclusions remain the same. There is a decrease in computational time by performing the simulation on the GPU, however the important comparison is the far larger decrease present in using a GPU for the Lagrangian method (Lattice-Boltzmann) over the Eulerian method (Finite-Volumes).

This does not demonstrate that the use of the Lattice-Boltzmann method necessarily represents a performance increase (the price of the CPU and GPUs are not taken into account), rather its conclusions are limited to demonstrating that the Lattice-Boltzmann method is more suited to performing calculations on a parallel architecture relative to the Finite-Volumes method. Figure 2 shows the performance in FLOPS (Floating Point Operations) of an average CPU versus and average GPU corrected for price, the graph is taken from the article "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware" (Brandvik and Pullan 2008). A substantial difference is seen to develop after 2003 and the trend continues resulting in a notable difference (around a magnitude of difference) between the parallel and serial architectures. This puts into context the advantage of a method of CFD that may be programmed for a parallel naturally poses. This trend also explains the recent heightened interest in Real-Time CFD as the increase in available processing power elevates the scope of a real-time simulation.

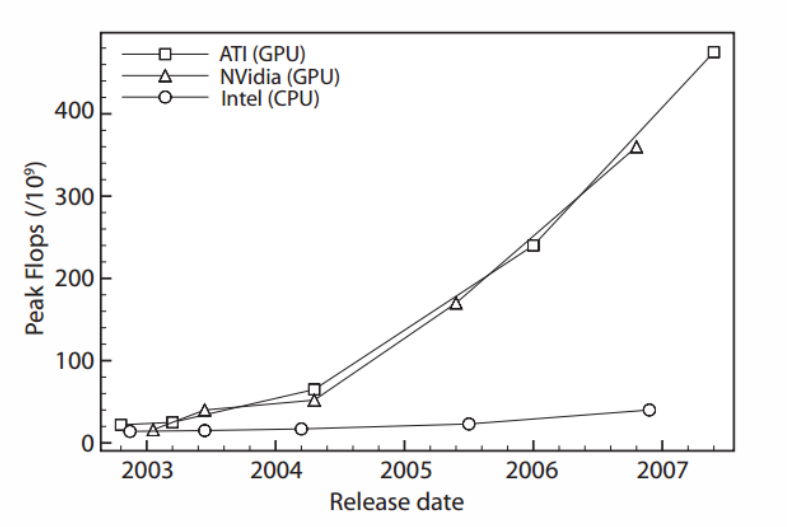

Figure 2: Graph of Increase in Available Computing Power with Respect to Time Given a Fixed Price. (Brandvik and Pullan 2008)

The Lattice-Boltzmann method (LBM) is unique from other methods of CFD discussed here, firstly because it does not solve the Navier-Stokes equations and secondly it is designed specifically to be implemented on parallel computing architectures. The LBM uses a series of particles that represent groupings of molecules and their interactions are modelled (Mohamad 2011). As the LBM simulates molecular interactions it does not rely upon the assumptions that Navier-Stokes based methods (such as Finite-Volumes) makes in order for continuum conditions to apply. Hence it can be applied in situations where continuum assumptions do not apply. This has made the LBM especially important in work on microfluidics (where the flow has a Knudsen Number of less than 0.1) and medical simulations (where multiphase flows are present) (O'Connor et al. 2016)

Discrete Vortex methods are a class of Lagrangian methods. They indirectly solve the Navier-Stokes equations by considering the evolution of vorticity of a series of particles and determining the velocity field induced (Cottet and Koumoutsakos 2008). Their implementation is discussed further in section 4.Being a Lagrangian method they are programmed easily for parallel architectures. Vortex methods however have not seen the same interest as SPH or LBM. No source could be found for this, however it is apparent from the number of published articles and books on each respective subject, Discrete Vortex Methods don't even a Wikipedia page.

Rosenhead was the first to formulate a complete vortex method (Rosenhead 1931) for 2D, inviscid and incompressible flow. However further developments to vortex methods have extended their capability to include compressible flows (Eldredge, Colonius, and Leonard 2002), viscous flows (Baden and Puckett 1990), turbulent flow (Barba 1996) and even combustion (Lakkis and Ghoniem 2003) making them a robust choice for simulation of many phenomena. However their major disadvantage is their inability to model strong boundary layer interactions such as pipe flows. However plenty of examples exist for simpler situations such as the flow around a sphere (Johnson and Patel 1999) or aerofoils (Xu 1999), these flows are characterised by a stationary solid body moving in a large body of fluid, producing a "free wake". This makes them ideal for the wake regions produced by aircraft it flight, further if a simplistic model of lift is used such as a Horseshoe Vortex then solid bodies need not even be taken into account.

In a Discrete Vortex Method a number of particles are assigned initial positions and vorticity, the operating loop of the simulation can then be summarized as follows; the velocity field at every particle is determined, each particle is then iterated to a new position using the value of the velocity field and the new value of the velocity field calculated and the procedure repeated. The calculation of the velocity field requires the consideration of the vorticity of every other particle, and this is required to be considered for every particle, hence the complexity of a vortex method increases proportional to N^2 where N is the number of particles. This is known mathematically as an N-Body problem and occurs predominantly in astrophysics.

Vortex methods have traditionally incorporated a semi-Lagrangian method using Harlows previously mentioned "Particle-In-Cell" method, which was done so that viscous effects may be calculated through vortex stretch (C. H. Liu 2001). However grid free methods with purely Lagrangian formulations of vortex methods have since been developed that take into account viscous effects (Barba 1996). Purely Lagrangian formulations benefit in that they closely emulate the maths of their astrophysics N-Body problem counterparts.

The comparison between the N-Body encountered in astrophysics and Vortex methods is not trivial. In astrophysical problems a series of planets (analogous to vortex particles) are present and the gravitation of every planet effects the velocity of every

other planet, whereas in Vortex methods the velocity of every particle is dependant on the vorticity of every other particle. The gravitational field decays according to an inverse square law and the influence of the vorticity of a particle decays with an inverse cube law. Astrophysical N-Body problems have a wealth of research and established methods of optimization, due to the similarities between the two problem formulations these methods of optimization may pose the possibility of being adapted to vortex methods.

A common algorithm used to optimize the simulation of an N-Body problem is the Barnes-Hut simulation, presented by Dr. Josh Barnes and Dr. Piet Hut in their paper "A hierarchical O(N log N) force-calculation algorithm" (Barnes and Hut 1986). Figure 3 is taken from Barnes and Hut's original paper and shows how the spatial domain is segmented into discrete partitions

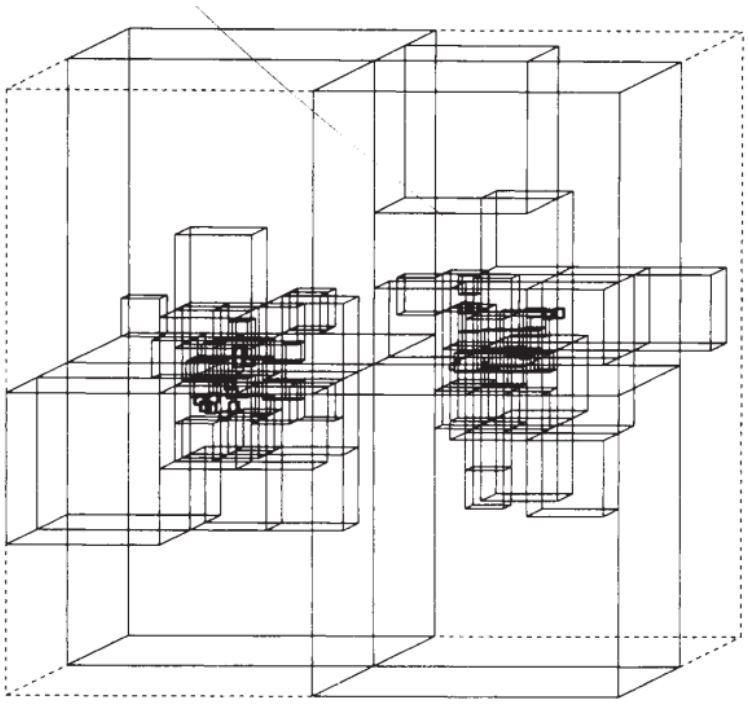

Figure 3: Example of 3D Spatial Domain segmented into Areas of an Influence Tree approach. (Barnes and Hut 1986)

Planets within these partitions are assumed to act as one planet of increased strength and an equivalent position. By assuming a number of planets can be assumed to act as one planet, fewer calculations are performed, and hence calculation

time decreased. Further, these spatial segments of planets can be segmented together numerous times. These methods are known as Influence Tree Methods. The theory behind this is perfectly applicable to Vortex Methods. However a number of modifications can be made to exploit features of Vortex Methods. The discrete particles in a DVM are distributed over a vortex sheet, whilst the sheet itself takes a 3d form there may be potential optimization potential by segmenting the spatial domain based up a 2d representation of this sheet. This report aims to implement a novel modification of the Barnes Hut algorithm applied to vortex methods.

2.1 Concluding Remarks

The early years in the development of CFD led to the development of many varied approaches to simulation, however the Finite Volume method has become the most successful and widely adopted method. However with increased computing power offered from parallel architectures, such as GPUs, there has been renewed interested in the methods developed before the Finite Volumes method took prevalence. The methods that have seen renewed interest employ either a particle or statistical mechanics approach, these approaches can be programmed naturally for parallel architectures whilst the mesh based Finite Volumes method do not.

The Lattice Boltzman and Smoothed Particle Hydrodynamic methods have become the most popular choices for Real-Time CFD simulation, Vortex methods have not seen the same interest and development, however their mathematical implementation represents an N-Body problem. N-Body problems occur in Astrophysics problems and have been subject to research and optimization methodologies have been developed. This report aims to develop a Discrete Vortex Method Simulation optimized using methodologies widely applied to Astrophysical N-Body problems, however implemented in the context of a vortex Wake simulation.

3 Theory - Vortex Methods

3.1 Introduction

This section serves to out line how Vortex Methods work, it provides a description however it does not provide a strict proof. The simulation will be split into two parts, the convective scheme and the discretization scheme, this puts into context the subsequent two theory sections which examine each scheme in greater detail. The description given here is provided for a 2D flow, the vortex method developed in this report is 3D, however only a 2D description is provided for brevity.

3.2 Theory

The Navier-Stokes equations are a set of equations that govern fluid flow, assuming continuum conditions apply. The Navier-Stokes are equations are shown in equations 3.1 through to 3.3. Equation 3.1 is the continuity equation and enforces the incompressibility and conservation of mass of the fluid. Equation 3.2 is the momentum equation and describes the evolution of the fluid velocity ensuring the conservation of linear momentum. Equation 3.3 is the energy equation and describes the evolution of fluid temperature. These equations are often solved by either an iterative method or simplifications are introduced so an analytical solution can be used.

$$div(\vec{V}) = 0 \tag{3.1}$$

$$\frac{\partial V}{\partial t} + V \cdot \nabla V - \nu \nabla^2 V + \frac{1}{\rho} \nabla P = F \tag{3.2}$$

$$\frac{\partial T}{\partial t} + V \cdot T - \alpha \nabla^2 T = \frac{s}{\rho c} \tag{3.3}$$

For the simulation discussed here a number of assumptions are made. Firstly it is assumed the fluid is incompressible, this automatically satisfies equation 3.1 and secondly the fluid is isentropic with constant thermo-physical properties, this automatically satisfies equation 3.3. This leaves us with only equation 3.2, the momentum equation, to consider.

$$\frac{\partial V_X}{\partial t} + v_X \frac{\partial V_X}{\partial x} + V_Y \frac{\partial V_X}{\partial y} - \nu \big(\frac{\partial^2 V_X}{\partial x^2} + \frac{\partial^2 V_Y}{\partial y^2} \big) + \frac{1}{\rho} \frac{\partial P}{\partial x} = F_X \tag{3.4}$$

$$\frac{\partial V_y}{\partial t} + v_X \frac{\partial V_y}{\partial x} + V_Y \frac{\partial V_y}{\partial y} - \nu \big(\frac{\partial^2 V_y}{\partial x^2} + \frac{\partial^2 V_y}{\partial y^2} \big) + \frac{1}{\rho} \frac{\partial P}{\partial y} = F_y \tag{3.5}$$

Equations 3.4 and 3.5 represent the momentum equation in Cartesian coordinates for a 2D flow. These two equations are non-linear PDE's and have no analytical solution. To simplify them further it is assumed that the flow is inviscid and the only external force acting upon it is gravity, in addition to being isentropic and incompressible. This results in equations 3.6 and 3.7

$$\frac{\partial V_x}{\partial t} + v_x \frac{\partial V_x}{\partial x} + V_y \frac{\partial V_x}{\partial y} + \frac{1}{\rho}\frac{\partial P}{\partial x} = g_x \tag{3.6}$$

$$\frac{\partial V_y}{\partial t} + v_x \frac{\partial V_y}{\partial x} + V_y \frac{\partial V_y}{\partial y} + \frac{1}{\rho}\frac{\partial P}{\partial y} = g_y \tag{3.7}$$

Equations 3.6 and 3.7 represent the momentum equation and describes the evolution of the velocity of the fluid in Cartesian coordinates. In order to describe the evolution of the fluid in terms of vorticity, the curl of the momentum equation is taken. In Cartesian coordinates this is performed via equation 3.8. The vorticity of a flow is defined as the curl of the velocity vector, for a 2D flow this is given by equation 3.9 and is simply equation 3.8 applied to a velocity vector with only i and j components.

$$curl(\vec{F}) = (\frac{\partial F_z}{\partial zy} - \frac{\partial F_y}{\partial z})\hat{i} + (\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x})\hat{j} + (\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y})\hat{k} \tag{3.8}$$

$$\omega = \Big(\frac{\partial V_y}{\partial x} - \frac{\partial V_x}{\partial y}\Big) \tag{3.9}$$

In equation 3.8 the function $\vec{F}$ represents the momentum equation. For a 2D flow the vorticity vector has only one component perpendicular to the flow plane. Hence for our momentum equation the definition for curl is calculated by equation 3.10

$$curl(\vec{F}) = (\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y})\hat{k} \tag{3.10}$$

Hence to calculate the the curl of the momentum equation we need to calculate the partial derivatives of equations 3.6 and 3.7 with respect to the y and x directions respectively. This is shown in equations 3.11 and 3.12.

$$\frac{\partial F_x}{\partial y} = \frac{\partial}{\partial y}\Big(\frac{\partial V_x}{\partial t} + v_x \frac{\partial V_x}{\partial x} + V_y \frac{\partial V_x}{\partial y} + \frac{1}{\rho}\frac{\partial P}{\partial x}\Big) \tag{3.11}$$

$$\frac{\partial F_y}{\partial x} = \frac{\partial}{\partial x}\Big(\frac{\partial V_y}{\partial t} + v_x \frac{\partial V_y}{\partial x} + V_y \frac{\partial V_y}{\partial y} + \frac{1}{\rho}\frac{\partial P}{\partial y}\Big) \tag{3.12}$$

Equations 3.11 and 3.12 are first simplified by assuming the pressure variation in the wake is sufficiently small so that it can be neglected. This is not a practical assumption for air travelling over the aerofoil, however it is applicable to the wake region to be modelled where the flow on either side of the aerofoil has merged. This reduces equations 3.11 and 3.12 to equations 3.13 and 3.14 respectively.

$$\frac{\partial F_x}{\partial y} = \frac{\partial}{\partial y}\Big(\frac{\partial V_x}{\partial t} + v_x\frac{\partial V_x}{\partial x} + V_y\frac{\partial V_x}{\partial y}\Big) \tag{3.13}$$

$$\frac{\partial F_y}{\partial x} = \frac{\partial}{\partial x}\Big(\frac{\partial V_y}{\partial t} + v_x\frac{\partial V_y}{\partial x} + V_y\frac{\partial V_y}{\partial y}\Big) \tag{3.14}$$

Performing the differentiations in equations 3.13 and 3.14 yields

$$\frac{\partial F_x}{\partial y} = \frac{\partial}{\partial y}\frac{\partial V_x}{\partial t} + \frac{\partial V_x}{\partial y}\frac{\partial V_x}{\partial x} + V_x\frac{\partial}{\partial y}\frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y}\frac{\partial V_x}{\partial y} + V_y\frac{\partial}{\partial y}\frac{\partial V_x}{\partial y} \tag{3.15}$$

$$\frac{\partial F_y}{\partial x} = \frac{\partial}{\partial x}\frac{\partial V_y}{\partial t} + \frac{\partial V_x}{\partial x}\frac{\partial V_y}{\partial x} + v_x\frac{\partial}{\partial x}\frac{\partial V_y}{\partial x} + \frac{\partial V_y}{\partial x}\frac{\partial V_y}{\partial y} + V_y\frac{\partial}{\partial x}\frac{\partial V_y}{\partial y} \tag{3.16}$$

Substituting equations 3.15 and 3.16 into equation 3.10 and factorising the resultant equation then yields equation 3.17. The equation is equal to 0 as the right hand sides of equations 3.6 and 3.7 are gravitational constant that go to zero when differentiated.

$$\frac{\partial}{\partial t}\Big(\frac{\partial V_y}{\partial x}-\frac{\partial V_x}{\partial y}\Big)+V_x\frac{\partial}{\partial x}\Big(\frac{\partial V_y}{\partial x}-\frac{\partial V_x}{\partial y}\Big)+V_y\frac{\partial}{\partial y}\Big(\frac{\partial V_y}{\partial x}-\frac{\partial V_x}{\partial y}\Big)+\Big(\frac{\partial V_x}{\partial x}+\frac{\partial V_y}{\partial y}\Big)\Big(\frac{\partial V_y}{\partial x}-\frac{\partial V_x}{\partial y}\Big) = 0 \tag{3.17}$$

In equation 3.17 there is a common factor throughout all terms, this is the aforementioned definition for vorticity given in 3.9. Substituting equation 3.9 into equation 3.17 yields equation 3.18

$$\frac{\partial \omega}{\partial t} + V_x\frac{\partial \omega}{\partial x} + V_y\frac{\partial \omega}{\partial y} + \omega\Big(\frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y}\Big) = 0 \tag{3.18}$$

In the last term in equation 3.18 the vorticity is multiplied by two partial differentials. This term is the divergence of the velocity, we previously defined the fluid to be incompressible, hence according to equation 3.1 this term is zero. Hence equation 3.18 becomes 3.19

$$\frac{\partial \omega}{\partial t} + V_x\frac{\partial \omega}{\partial x} + V_y\frac{\partial \omega}{\partial y} = 0 \tag{3.19}$$

Equation 3.19 is a form of the vorticity equation for a 2D inviscid incompressible flow in a Eulerian reference frame, it describes the evolution of vorticity with respect to time. In order to use this for a Lagrangian simulation this equation needs to be written in terms of a Largrangian reference frame. A definition for the Material Derivative is given in equation 3.20 for a arbitrary variable φ travelling in a fluid of velocity $\vec{V}$

$$\frac{D(\varphi)}{Dt} = \frac{\partial(\varphi)}{\partial t} + \vec{V}\nabla\varphi \tag{3.20}$$

Hence substituting φ for ω and expansing yields equation 3.21

$$\frac{D\omega}{Dt} = \frac{\partial\omega}{\partial t} + V_x\frac{\partial\omega}{\partial x} + V_y\frac{\partial\omega}{\partial y} \tag{3.21}$$

Substituting the results of equation 3.19 into equation 3.21 results in equation 3.22

$$\frac{D\omega}{Dt} = 0 \tag{3.22}$$

Equation 3.22 represents Helmholtz's equation. The result of this equation is significant, it governs the evolution of vorticity of a fluid in a Lagrangian reference frame. However, as it can be seen, there is no change in vorticity with respect to time. The vorticity is constant. Hence, a packet of fluid evolving with time and translating has constant vorticity. For a particle based simulation this means that all particles conserve their vorticity.

Hence we know the vorticity of a series of particles of fluid at any point in time, however, to determine their position the velocity field needs to be calculated. The velocity field is recovered from the vorticity through the Biot-Savart law, shown in equation 3.23

$$\vec{V}(x,y,z) = \frac{\Gamma}{4\pi}\int_{+\infty}^{-\infty}\frac{d\vec{\ell}\times\vec{r}}{|\vec{r}|^3} \tag{3.23}$$

The Biot-Savart law allows for the calculation of the induced velocity at any point due to a vortex filament hence the integral across a length (extending to infinity if the fluid has no boundaries due to Helmholtz's second theorem). However, in the present simulation continuous vortex filaments are approximated to a series of discrete points.

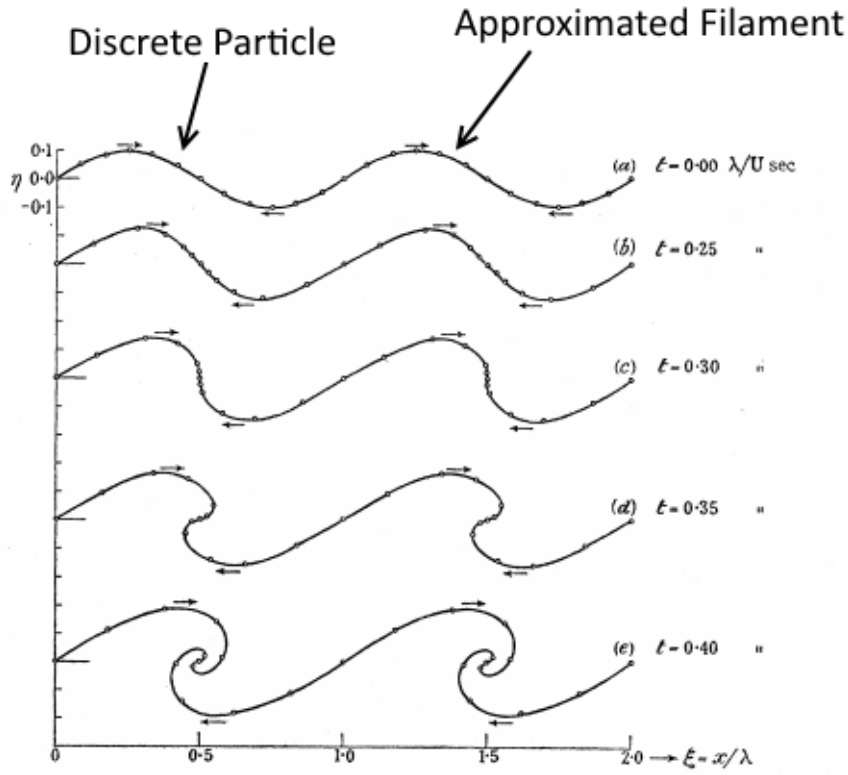

Figure 4: Vorticity Represented with Discrete Particles with Approximated Filament Draw Through Particles. (Rosenhead 1931)

Figure 4 is taken from Rosenhead's original paper and shows a series of discrete vortex particles (the larger dots along the lines) approximating a vortex filament (Rosenhead 1931). The filament is drawn as an interpolating function along the discrete points. The figure also shows Rosenhead's original simulation with every filament drawn being the evolution after a certain time of the filament above it.

The Biot-Savart law given in equation 3.23 assumed a velocity field induced by a vortex filament, if discrete vortex points are used then the equation reduces to equation 3.24

$$\vec{V}(x,y,z) = \frac{\Gamma \times \vec{r}}{4\pi \times |\vec{r}|^3} \tag{3.24}$$

Equation 3.24 gives the velocity field induced by a single discrete vortex. The velocity induced by a series of discrete vortex particles is the superposition of the

velocity fields induced by every individual element. Given a number N of elements, the velocity field at any point is given by equation

$$\vec{V}(x,y,z) = \sum_{i=0}^{N} \frac{\Gamma_i \times \vec{r_i}}{4\pi \times |\vec{r_i}|^3} \tag{3.25}$$

Accordingly a full description of the flow field is obtained. The particles have a initial vorticity, which stays constant throughout the simulation. Their velocities can be extracted from their initial positions and vorticities. Using the velocity field their positions can be incremented for a finite time period δt and the process repeated

3.3 Simulation Algorithm

Figure 5 shows the basic algorithm implemented for any vortex method. The algorithm starts by with the definition of the initial conditions of the simulation. The velocity field is then calculated via the Biot-Savart law, this is knows as the "Convective Scheme". The positions of the elements are then iterated over a finite time step δt, which is known as the "Discretization Scheme". The output is then displayed to the user via the "Rendering Scheme". Both the "Convective Scheme" and the "Discretization" scheme are discussed in further detail in their respective theory sections. The "Rendering Scheme" is not discussed however the code is included in the appendix.

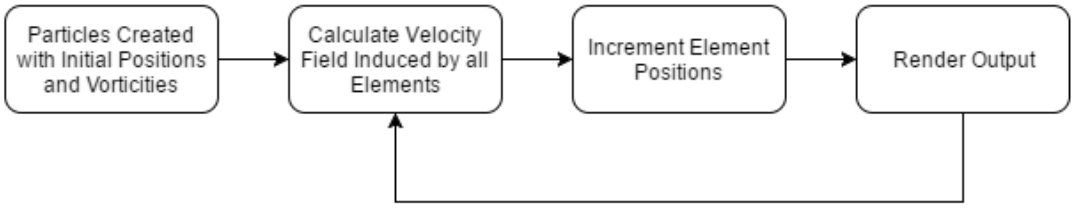

Figure 5: Flow Chart Representation of Basic Components and Main Loop of a Discrete Vortex Method Simulation

4 Theory - Convection Schemes

4.1 Introduction

The convective scheme is responsible for calculating the velocities of all Discrete Vortex Elements given the velocity fields induced by all other elements. The velocity field due to a vortex element at any given point is given by the Biot-Savart law, shown in equation 4.1

$$\vec{V}(x,y,z) = \frac{\Gamma \times \vec{r}}{4\pi \times |\vec{r}|^3} \tag{4.1}$$

To calculate the velocity field at any given point, the influence of all vortex elements must be taken into account, this is found via the summation of the influences of all elements (representing filaments) found through the Biot-Savart law. This is represented in equation 4.2 where N is the total amount of elements, and n is the individual element being considered

$$\vec{V}(x,y,z) = \sum_{n=1}^{N} \frac{\Gamma_n \times \vec{r_n}}{4\pi \times |\vec{r_n}|^3} \tag{4.2}$$

The computational cost of a single evaluation of the Biot-Savart law does not change meaningfully for different variables input into equation 4.1, the maths required is constant but slight variations may arise from larger inputs. Hence, making the assumption that the computational cost of a single evaluation of the Biot-Savart law is constant, we can express the computational cost as in equation 4.3 where $C_{Inv}(x,y,z)$ is the computational cost due to a single element at a given point and A is a constant.

$$C_{Single}(x,y,z) = A \tag{4.3}$$

From the expression in equation 4.3 we can form an expression for the velocity field induced by all elements, this is shown in equation 4.4

$$C_{Total}(x,y,z) = \sum_{n=1}^{N} C_{Inv}(x,y,z) = NA \tag{4.4}$$

From equation 4.4 it can be seen that the complexity of calculating the velocity field is linearly proportional to the amount of elements present. The velocity of an element is found by evaluating the velocity field at that elements position. A single

iteration of the convection scheme must calculate the velocity of all elements present, an expression for the computational cost is given in equation 4.5

$$C_{Iteration} = \sum_{n=1}^{N} C_{Total}(x_n, y_n, z_n) \approx AN^2 \tag{4.5}$$

Equation 4.5 shows that the computational cost of calculating the new velocities of all elements present increases in complexity proportional to N^2. This is known mathematically as an N-Body problem. The computational cost of a single evaluation of the Biot-Savart law, A, is fixed. However modifying the way in which the law is applied constitutes the basis for more computationally efficient convective schemes.

In this section, two modifications to the convective scheme are suggested and evaluated. Both schemes necessarily introduce errors into the numerical solution. The schemes are evaluated based upon a consideration of their performance and accuracy

4.2 Biasing

Biasing is the simplest method of optimization for the convection scheme. Biasing works by "biasing" which elements are taken into account when convecting a given element. Consider again the Biot-Savart law, shown in equation 4.6 for clarity.

$$\vec{V}(x, y, z) = \frac{\Gamma}{4\pi} \int_{+\infty}^{-\infty} \frac{d\vec{\ell} \times \vec{r}}{|\vec{r}|^3} \tag{4.6}$$

From equation 4.6 two things can be noted, the influence of an element on a given element is majorly dependent only on two variables, the convecting elements vorticity Γ and the its distance vector $\vec{r}$. Biasing methods exploit these relationships by ignoring the convective effects of an element if its influence on a certain element can be considered negligible.

Determining whether or not an elements influence is negligible requires an evaluation of either the vorticity Γ or distance vector $\vec{r}$ of the given element, or a coefficient based upon both of these. This evaluation must be performed for all elements every time an element is convected, so the evaluation must be performed N^2 times per iteration, this is an equal amount of times as the Biot-Savart law must be evaluated in the absence of a biasing scheme. It is also important to note that this evaluation process must be performed even for elements whose effects will be taken into account. The evaluation scheme must therefore be kept lightweight and efficient in order to

pose a performance increase of the scheme. If a computationally costly evaluation process is used it may pose a decrease in both overall performance and accuracy of the simulation.

The simplest biasing system would make an evaluation based upon both Vorticity and distance vector, such an evaluation is shown in the condition in equation 4.7 where A is an arbitrary constant.

$$\frac{\Gamma}{\vec{r}} > A \tag{4.7}$$

This approach provides the most accurate results, however it is also the most computationally expensive as it involves both an evaluation and a calculation. Two other criteria may be used for biasing the effect of elements based solely on distance and vorticity respectively. Evaluations based on on either vorticity or distance have the advantage of being more lightweight and representing an almost negligible overhead Whether it is best to consider vorticity or distance depends on the situation that need be modeled. A situation where all elements are close would likely benefit more from an evaluation based on vorticity, however an application where elements are spaced with a larger distance may benefit from an evaluation based on distance.

The simulation considered here is of a wake. This involves initially closely spaced elements seeded with initial values along the wing of an aircraft. As the simulation progresses more elements are spawned in rows, the distance between early rows and later rows increases linearly as more rows are spawned. As there is a large distance between elements in the simulation, a distance based biasing method is appropriate. New rows of elements are created when the distance between the bound elements and the last seeded free elements reaches a limit, from this a grid with near-uniform spacing is created. Thus, the computation cost for convecting an individual element is given by equation by equation 4.8, where B refers to the amount of elements present when the grid is sufficiently large such that elements are spread over a distance where the distance based evaluation is met.

$$C_{Total(x,y,z)} = \begin{cases} NA & N < B \\ BA & N > B \end{cases} \tag{4.8}$$

In equation 4.8 it can be seen that the computational cost for a given element is seen to reach a maximum value and remain constant. If the grid is sufficiently large

such that this condition becomes true, the computational cost is given by equation 4.9

$$C_{Iteration} = \sum_{n=1}^{N} C_{Total}(x_n, y_n, z_n) = NBA \tag{4.9}$$

From equation 4.9 it is apparent that if a well designed clustering scheme is implemented, the N-Body problem reduces in complexity from ON^2 to ON.

4.3 Clustering Schemes

4.3.1 Introduction

Clustering schemes assume that the effect of certain elements on a given element may be "clustered" and approximated to the effect of one element of an equivalent strength and position. By clustering elements together their influence can be taken into account with only one iteration of the Biot-Savart law, which reduces computational overhead. An example of this is shown in Figure 6, where in situation A the leftmost element is convected by a series of distant but closely spaced element. In situation B, the series of disant but closely spaced elements are approximated to a single element.

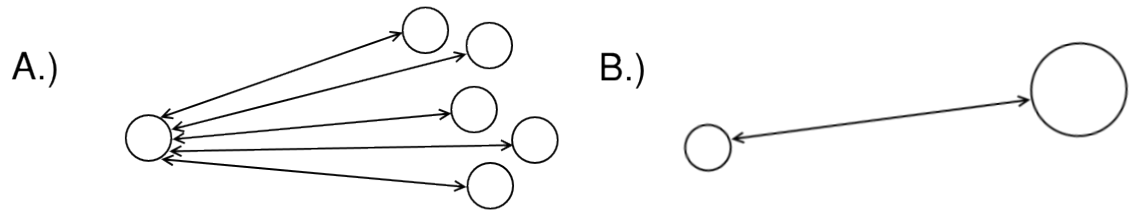

Figure 6: Distant Elements Approximated to a Single Cluster

The series of elements approximated to a single element in Figure 6 is only valid when the convection of the leftmost element is to be performed. For example, the cluster grouping shown could not be used to convect an element inside the cluster group. Hence every element represents a unique situation with its own unique cluster groupings. However, the calculation of these unique cluster groupings represents a large computational overhead, and thus is not beneficial as a method of optimization.

To overcome the large computation overhead associated with calculating unique cluster groupings for individual elements a method is used where the cluster groupings are

selected based upon groupings which are applicable to most elements. For example, consider figure 7. Here 3 sets of closely spaced elements can be seen, clustered into the groups A, B and C.

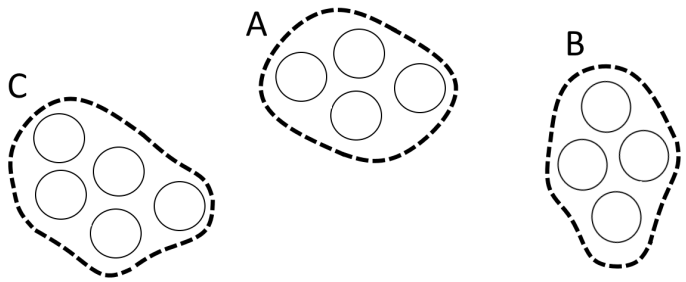

Figure 7: Closely Spaced Elements and their Respective Cluster Groupings

Of course, these cluster groups cannot be used to convect any of the elements, as all elements are in clusters. However, when a single element is to be convected, it could be convected with all elements in its own cluster, and all clusters except its own. This way every element is convected by every other element either directly or through a cluster. Consider for example an element inside cluster B, its convection in this manner is demonstrated in figure 8

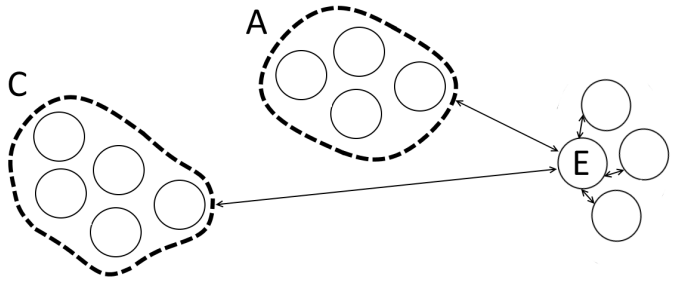

Figure 8: Element 'E' in Cluster 'B' Convecting with Individual Elements and Clusters

In figure 8 an element "E" from cluster B is shown being convected in the previously discussed way. Consider all elements in cluster C, if they are convected in this manner they will be convected by clusters A and B. Hence, the elements in B and A will both convect with cluster C. Likewise elements from clusters C and A will have cluster B in common. In this way clusters may be calculated once for all elements and still be applicable, despite each element requiring a unique situation.

If clusters A and B were closer together it may not be beneficial to convect individual elements in B with the cluster of A but instead with the individual elements of A. Neither is it necessary to convect the elements in cluster B with cluster A, an evaluation based on distance and vorticity could be made to determine whether it would be best to take into account close clusters as a cluster, or their constituent elements.

Approximating a series of elements to a cluster imposes a degree of error as those individual elements are assumed to act at the position of the cluster rather than their actual positions. This error is unavoidable, however its magnitude can be reduced by proper cluster grouping and by more accurate ways to determine cluster position.

4.3.2 Fixed Cluster Scaling

A fixed cluster scaling scheme is the simplest clustering scheme considered in this report. The scale of the cluster is used here not to refer to the amount of elements in the cluster. The amount of elements in a cluster is determined by the spatial positioning of the elements. Rather, the scale of a cluster refers to the way clusters are combined to create new clusters. Up until this point clusters comprised solely of elements have been considered, however a situation could be envisaged where with increased distance from a series of clusters, those clusters could be further clustered together to create a cluster comprised of clusters. This is demonstrated in figure 9 where situation A represents two distant clusters convecting with a given cluster and situation B represents the same situation where the two distance clusters are clustered together.

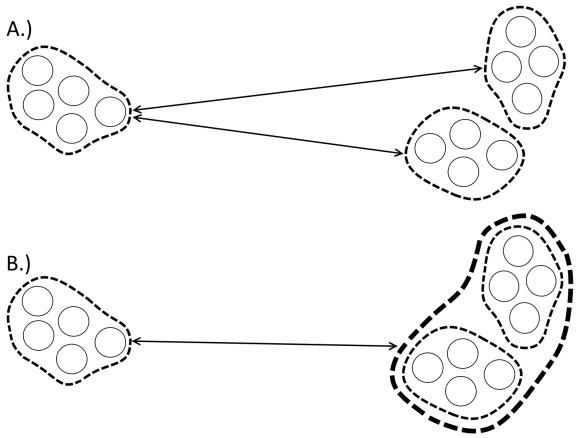

Figure 9: Example of Clustering Clusters Together

In a fixed cluster scaling scheme, clusters are clustered together. Conversely, in a dynamic cluster scaling scheme, clusters are clustered together where appropriate to create clusters of clusters.

In a fixed cluster scale scheme the clusters are determined via the spatial position of the elements. For the cluster groups to be useful they must be comprised of elements clustered together because they are spatially close together, and this gives rise to a physical size of the cluster. In the present simulation, elements are created in a line equidistant to each other and released with uniform velocity perpendicular to the spawn line. Thus the elements form a roughly grid shaped pattern as shown in figure 18.

Because the elements form a grid shaped pattern with roughly equal x and y grid spacing the element count in a cluster is fixed, giving rise to the cluster size $N_{Cluster}$. The Biot-Savart law is required to be iterated for all clusters except the cluster the element in question is contained within, every element from this cluster must be taken into account, the computational cost of convecting a single element is given in equation 4.10

$$C_{Element} = A\frac{N}{N_{Cluster}} + A(N_{Cluster} - 1) \tag{4.10}$$

The first term in equation 4.10 represent the computational cost of iterating the Biot-Savart law for all the clusters. Likewise, the second term represents the computation cost of iterating the Biot-Savart law for the individual elements in the element in questions cluster. The computational cost for convecting all elements present is therefore given by equation 4.11, the term $C_{Cluster}$ is the computational cost of calculating the cluster groupings every loop.

$$C_{All} = NA\Big(\frac{N}{N_{Cluster}} + (N_{Cluster} - 1)\Big) + C_{Cluster} \tag{4.11}$$

Equation 4.11 can be rearranged to equation 4.12

$$C_{All} = \frac{AN^2}{N_{Cluster}} + AN(N_{Cluster} - 1) + C_{Cluster} \tag{4.12}$$

From equation 4.12 it is apparent that the increase in complexity of the scheme still increases proportional to ON^2 as in the unclustered case. However the leading term for the fixed scale cluster scheme is proportional to $AN_{Custer}^{-1}N^2$ compared to the unclustered case where complexity rises with AN^2. This necessarily represents an

optimization, assuming $C_{Cluster}$ is suitable low, as $N_{Cluster} > 1$ and in practice takes a value of around 10.

4.3.3 Dynamic Cluster Scaling

The aforementioned clusters were determined dependant on their spatial locations. This resulted in clusters that were widely applicable to all elements. As distance between an element and a series of given cluster increases, these clusters will still be applicable, however with increased distance the accuracy lost from clustering together these clusters reduces, so it is computationally beneficial to treat the series of clusters as a single cluster, again with an equivalent strength and position.

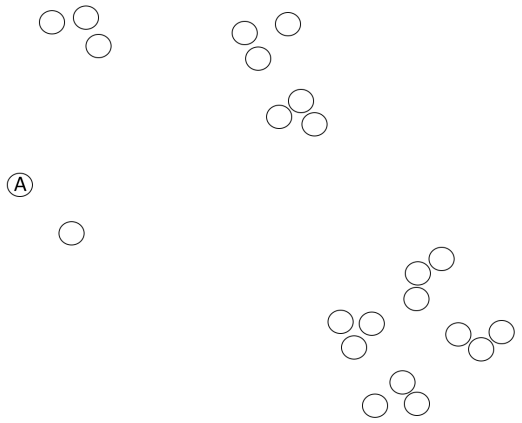

Figure 10: Situation of Unclustered Elements

To demonstrate this, consider element A in figure 11. Element A must currently convect with all other elements in the situation. However, most of the elements in the scene can be seen to occur in groups of 3, using these groups as the basis for clustering elements, the situation reduces to figure 11

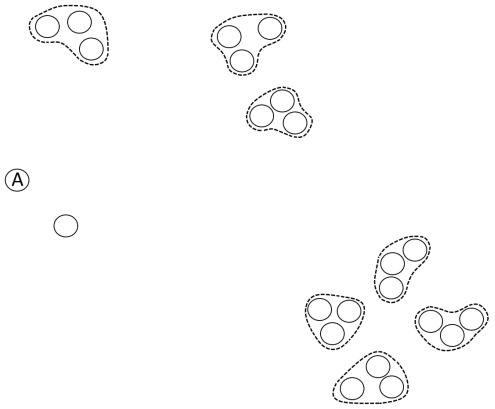

Figure 11: Fixed Cluster Size Applied to Situation

In figure 11 a situation is shown where a fixed cluster scale has been applied. Whereas A originally had to convect with 22 separate elements, it must now convect with 1 element and 7 clusters. This can be reduced further by applying a dynamic cluster scaling system, this is shown in figure 12

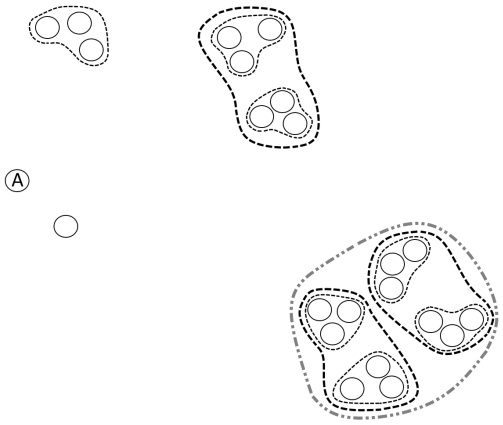

Figure 12: Dynamic Cluster Size Applied to Situation

In figure 12 element A now must convect with a single element, a cluster, a cluster of two clusters and finally, a cluster of two clusters which themselves are clusters of two clusters. Hence It must convect with 1 element and 3 clusters, a large reduction in computation cost over the original 22 elements. This is an example of dynamic cluster scaling

To determine how the computational complexity increases with N, first consider the way elements are clustered in figure 13. This shape of clustering is significant as

it is the shape implemented in the DVM, discussed further in section 7.3. Two assumptions are made, firstly it is assumed that elements always exist in a perfect grid shape. Secondly, it is assumed that there is a constant cluster size, β, and elements are clustered into square clusters of $\beta \times \beta$, and clusters of clusters are also clustered into clusters of $\beta \times \beta$. In figure 13 a series of numbers can be seen be seen next to the grid, this can be thought of as a coordinate system where the unit is iterated every time the end of a cluster is reached spanning a single axis, for brevity this will be referred to as φ.

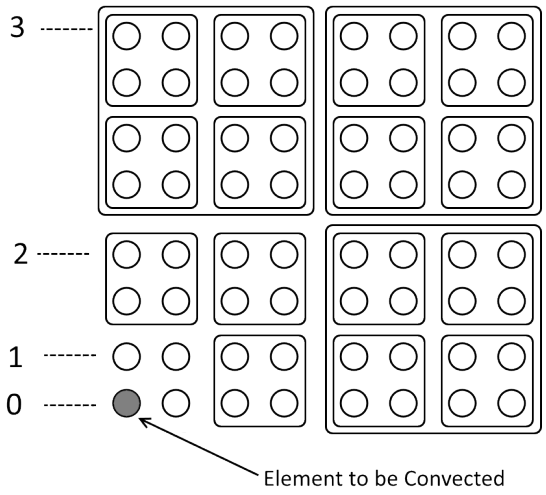

Figure 13: Dynamic Cluster Size Applied to Situation

The first step in deriving the relationship is to consider for every value of φ, how many elements and clusters are considered, and the number of elements and cluster is denoted by N_{Total}. As the assumption that clusters always occur in constant sizes of $\beta \times \beta$ then as φ is incremented by unity then 3 clusters are added. At a value of $\varphi = 1$ there are $\beta \times \beta - 1$ elements to convect with (as an element cannot convect with itself). Hence for all non zero positive integers N_{Total} is given by 4.13

$$N = (\beta^2 - 1) + 3(\varphi - 1) \tag{4.13}$$

Expanding the brackets and combining constants into a single constant C in equation 4.13 results in equation 4.14

$$N = C + 3\varphi \tag{4.14}$$

This represents the Number of times the Biot-Savart law needs to be performed, making the assumption that every iteration takes the same computational time, represented by a constant A, an expression for the computational cost in terms of φ can be found. This is shown in equation 4.15

$$Cost = C + 3A\varphi \tag{4.15}$$

Now the number of elements present for every value of φ needs to be found. Consider the position along a single dimension of the square for each value of φ. In the example shown in figure 13 the values of $\varphi = 1, 2, 3$ refer to positions $2, 4, 8$ along a single dimension of the grid. In general then the position is given by β^{φ}. Hence the amount of elements present as a function of φ is given by equation 4.16

$$N_E(\varphi) = (\beta^{\varphi})^2 = \beta^{2\varphi} \tag{4.16}$$

Equation 4.16 can then be arranged to find φ in terms of the total number of elements. This is shown in equation 4.17

$$\varphi = \frac{1}{2}\log_{\beta}(N_E) \tag{4.17}$$

Substituting the value of φ from equation 4.17 into equation 4.15 yields an equation for the computational cost in terms of the number of elements, this is shown in equation

$$Cost = C + \frac{3}{2}\log_{\beta}(N_E) \tag{4.18}$$

In equation 4.18 it can be seen that the cost increases proportional to $O\log(N)$.

4.3.4 Reactive Cluster Grouping

As previously discussed, predictively grouping clusters may not suffice for all conditions. If this is true clusters must be grouped reactively based upon their current positions. This introduces a necessary overhead associated as groupings must be calculated periodically.

The simplest reactive method of calculating cluster groupings splits the spatial domain into a grid where each grid segment represents a cluster, this is shown in figure 14.

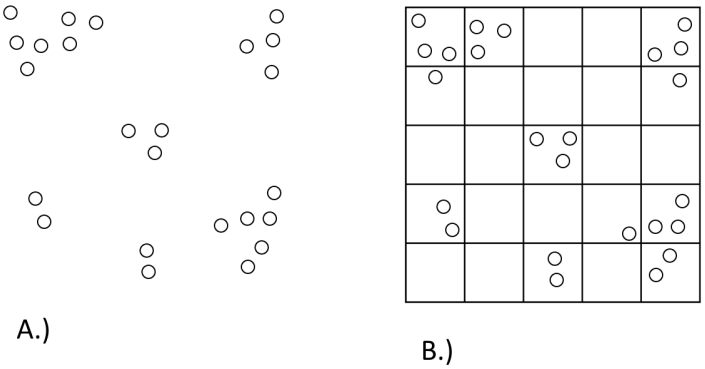

Figure 14: Scattered unclustered elements (A) shown with grid based clustering applied (B)

In figure 14, a grid has simply been overlaid on a series of scattered elements, in essence, this is exactly how the method works, however extended into 3D. Elements inside a given grid segment are assumed to act as part of a cluster.

As every element in a given grid space is assumed to be part of a cluster, calculating cluster groupings is easy as only the the coordinates of the element need be taken into account and not which elements are close to a given element. Each cluster grouping may also be identified by a set of coordinates relative to the grid (referring to the placement on the grid, not actual spatial dimensions), and this is demonstrated in figure 15.

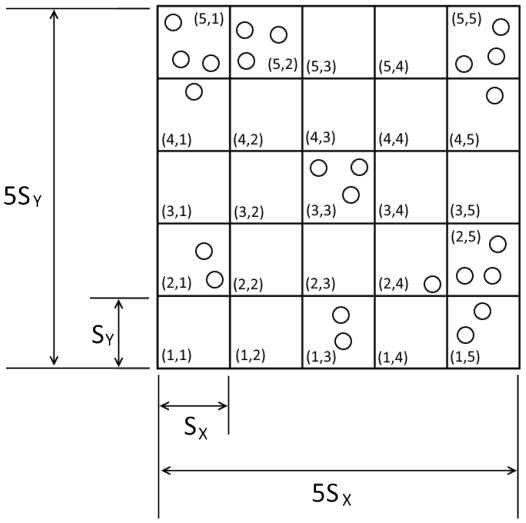

Figure 15: Grid Based Clustering Coordinate System and Grid Spacing

In figure 15 the grid spacings S_X and S_Y are shown. The nomenclature (x,y) is used to specify a given grid cell. Hence determining which cluster and element belongs to in the form (N_x, N_y) is done by evaluating when the conditions in equation 4.19 and 4.20 are true.

$$S_x(N_x - 1) < P_x \leqslant S_x N_x \tag{4.19}$$

$$S_y(N_y - 1) < P_y \leqslant S_y N_y \tag{4.20}$$

Clustering elements into such a grid has a second advantage, dynamic cluster scaling is easy to apply. For example, it is easy to cluster grid cells into clusters of 2x2 or 4x4 cells. Furthermore it is easy to determine in advance which cells should be treated as individual elements, clusters and larger order clusters, as the spacing of cells is constant. A predefined map of how cells should be treated can be defined, such an example is shown in figure 16. In figure 16 the convection of every element in a single cell (shaded in gray) is considered, as the spatial domain is split into discrete cells of constant spacing, a pattern of cell clustering can be overlaid on its position is valid for any cell in the spatial domain. Using such a method to determine how cells are considered (elements, clusters ect..) avoids having to evaluate clusters based on position and strength reactively.

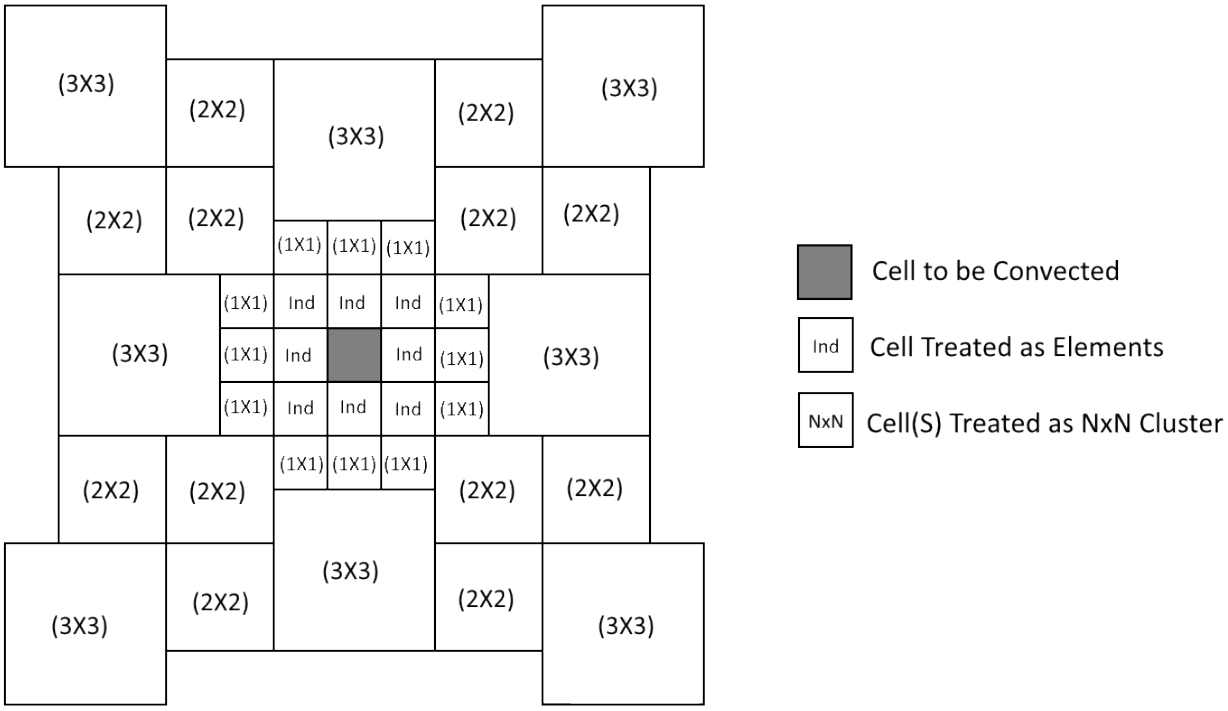

Figure 16: Example of Predetermined Clustering of Grid Cells

4.3.5 Predictive Cluster Grouping

Elements are created with equal spacing along the span of the wing, this is demonstrated in figure 17 A. Elements travel away from the aerofoil with the free stream velocity, this is shown in figure 17 B.

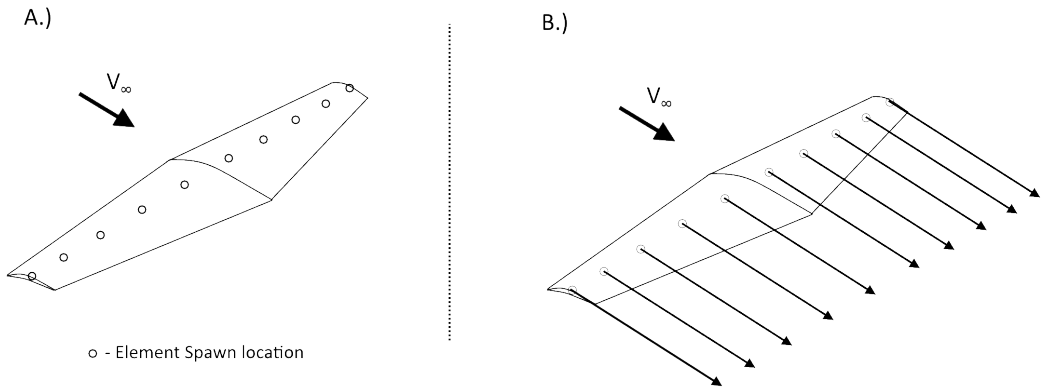

Figure 17: Element Spawn locations Along a Wing Span

After a certain time, a new row of elements is created, the velocity of the elements in both is roughly the same, and this gives rise to a grid shaped pattern of elements, shown in figure 18.

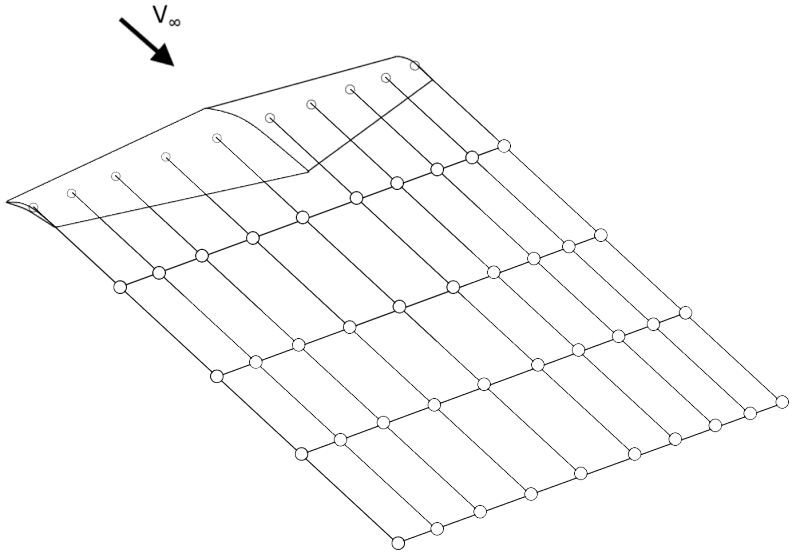

Figure 18: Apparent "Grid" shape Made by Successive Element Rows

This represents a vortex sheet, of course the elements do not stay in a perfectly planar grid pattern, however neighbouring elements do stay close to each other as the

sheet rolls up at the sides. Predictively clustering elements assumes that this grid can be assumed to stay in a shape, though changing, where clusters can be predicted based up on the sheet shape. As a grid shape is present, pre determined clustering "shapes" such as those seen in figure 16 can be used.

This clustering system becomes is when this assumption becomes untrue. When strong wingtip vortices are formed, or lift generation is changed suddenly, are both examples of when this assumption leads to poor accuracy. However, this method may be advantageous over the reactive clustering method as it presents no overhead associated with determining cluster groupings

4.3.6 Cluster Position

An error is introduced into a simulation using a clustering scheme as the position of every individual element in a cluster is lost and every element assumed to act the clusters potion. Determining the position of the cluster will therefore have an effect of the magnitude of this error. The simplest approach to calculate a clusters position would be to average the coordinates of all elements in the cluster. This method weights the position of all elements equally in a cluster. In this section two alternative methods are discussed, a weighted average and a simpler highest influence approach.

Consider a cluster consisting of an amount $N_{Cluster}$ of elements, the position vector of any given element is P_N and its respective vorticity is ω_N. To find a weighted average the total vorticity of the cluster is first found, this is shown in equation 4.21

$$\omega_{Total} = \sum_{N=1}^{N_{Cluster}} \omega_N \tag{4.21}$$

Every element is asigned a relative weighting W_N, for the simple average position case this would be constant for all elements and be equal to $N_{Cluster}^{-1}$. This approach weights the potion of the cluster based on the vorticity of each element, the influence of every element is defined as the ratio of its vorticity to the total vorticity of the cluster. This is shown in equation 4.22

$$W_N = \frac{\omega_N}{\omega_{Total}} \tag{4.22}$$

Hence the weighted average position of the cluster is given by equation 4.23

$$P_{Cluster} = \sum_{N=1}^{N_{Cluster}} \frac{\omega_N}{\omega_{Total}} P_N \tag{4.23}$$

The position of clusters are determined only once for every convective cycle, hence the computation overhead of determining their position is minor in comparison to the cyclic convecting routine. Whilst it is true that determining cluster position should not pose a significant slow down of the simulation, the process can be optimized further by using a Highest Influence approach. In such a method, the cluster is assumed to act at the point of the element whose vorticity is the highest of the cluster. As there is no calculation involved in this method, it is significantly more lightweight than both the average and weighted average approaches.

5 Theory - Temporal Discretization

5.1 Introduction

The convective routine results in the calculation of the velocities of all elements at a given time, given a series of element positions and vorticity, solving analytically results in a large number of interdependent differential equations that are problematic and or inefficient to solve in real time. Instead the time domain is discretized into discrete Time Steps and numerical approximations used to solve for the position of the elements at the end of the time step. Splitting the time domain into discrete time steps is known as Temporal Discretization.

The problem can be stated as given a current position X_1, time t_1 and velocity v_1, find the position x_2 at $t_2 = t_1 + t_{ts}$. This is shown in equation 5.1 where $f(v_1, t_{ts})$ represents the scheme used.

$$x_2 = x_1 + f(v_1, t_{ts}) \tag{5.1}$$

The discretization process necessarily introduces discretization error. By approximating the continuous function to a discrete series of points a truncation error is introduced, this error reduces as more points are used to represent the continuous function. Hence, as a finer time step is used, this error will decrease. The way in which this error scales with the time step is known as the order of accuracy of the scheme. The order of accuracy is determined by how the leading error term scales with the time step. For example, in a first order scheme, the leading error term is proportional to the time step used squared, this is shown in equation 5.2. Likewise, in a second order scheme, the leading error term is proportional the the time step cubed, this is demonstrated in equation 5.3. Note, these definitions are for schemes applied to differential equations, they are not the same as the traditional CFD definitions for order of accuracy used in finite volumes.

$$x_2 = x_2 + ot_{ts}^2 \tag{5.2}$$

$$x_2 = x_2 + ot_{ts}^3 \tag{5.3}$$

It would then seem that a higher order scheme is preferable, as the truncation error is reduced quicker as the time step is refined. However, higher order schemes are also associated with a higher computational overhead, and diminishing returns from even higher order schemes will become apparent. Further, the truncation error

is not the only variable when deciding upon a scheme to use. Two major considerations are made when determining a scheme to use, the convergence and stability of a scheme. As truncation errors sum up as the scheme is iterated, the approximated value of the function is seen to "drift" away from the actual value of the function as the approximation diverges. To increase convergence, a finer time step or higher order discretization scheme can be used.

The rate of convergence of a scheme can be expressed as the rate at which the error norms of the approximation reduces as the time step is refined, a definition of error norms is given in equation 5.4 where $x^{(n)}$ is the numerical approximation to the continuous function and x^{ex} is the exact solution. Convergence of a scheme can be tested by calculating the Error norms of a known continuous function

$$Error^{(n)} = \sum_{n}^{1} |x^n - x^{ex}| \tag{5.4}$$

Higher order schemes often decrease the stability of the approximation. Stability issues arise due to the approximation consistently over or underestimating the value of the function, or the scheme producing non-physical results. Higher order schemes are more susceptible to this error as they may produce more extreme fluctuations based on their inputs if those inputs are based only on current and/or previous values of the function and the independent variable, these are known as explicit schemes. However schemes that utilize future values of the function and independent variables, known as implicit scheme, do not exhibit this behaviour to a significant degree.

It may seem that a high order implicit scheme would be the best choice for a discretization scheme. These are the most common schemes employed by real time physics engines. Whilst PhysX (utilized by unity) and other popular physics engines such as Havok and Newton are closed source and so their schemes are not disclosed (PhysX is know to use a implicit simplectic scheme of unknown order (Boeing and Braunl 2007)) a number of open source engines are available. ODE (Open Dynamics Engine) is a popular open source physics engine, being open source its source code can be freely examined. The Open Dynamics Engine source was examined and comments indicated the use an implicit method described by Stewart and Trinkle in their 1996 paper "An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and Coulomb friction" (Stewart and Trinkle 1996).

Despite both PhysX and ODE using implicit schemes, an explicit scheme is to be used for this simulation. For PhysX and ODE the use of an explicit scheme is reasonable as the largest computational overhead is not associated with the calculation of a velocity of an element (in the the case of PhysX and ODE the element is a Rigidbody, akin to the Discrete Vortex Element in this simulation) and thus it is practical to do this numerous times per time step. Their largest overheads are associated with processes related to collision detection and force summation. However in the present simulation, the single largest overhead is associated with the calculation of the velocity of an element, to such a degree that the discretization scheme represents almost no overhead. Thus using the convection routine to calculate future velocities of elements would represent a decrease in performance. This was noted by Rosenhead in his original paper on vortex methods, however he does not formulate any argument on the use of implicit schemes (Rosenhead 1931).

As the discretization scheme represents a near negligible overhead compared to the convection routine, the motivation behind developing a new scheme should not centre majorly on the performance of the scheme but the increase in accuracy of the scheme. Any increase in performance of the scheme may represent a slight increase in performance of the overall simulation. However, an increase in accuracy of the scheme will allow for a coarser time step to be utilized to achieve the same accuracy as the original scheme. This coarser time step allows for more computation time allocated to the convective routine, given more computational time more elements can be calculated. Thus, improving the accuracy of the scheme represents a possible large increase in accuracy by allowing more elements to be used during the simulation.

In this section, Eulers method of discretization is presented along with 2 novel explicit methods developed for the present simulation based on extrapolations of interpolating polynomials. These schemes are then assessed for their convergence and stability and a suitable scheme selected for the simulation. Eulers scheme is the only scheme that approximates the future value of the function using only current conditions. This is of importance as explicit schemes may only be used once enough time steps have passed such that the previous values they require can be known, thus Eulers scheme is a necessity to be used at the start of the simulation if an explicit scheme is to be used.

5.2 Eulers Method

The convective routine results in the calculation of the velocities of all elements, Eulers scheme makes the assumption that this velocity is constant throughout the entire time step. So it assumes the downwind velocity of the fluid at the next time step is the current fluid velocity. This is shown in equation 5.5 where v_t is the v_x velocity at the current time step and t_{ts} is the time step.

$$x_{new} = x_{current} + \int_{t}^{t+t_{ts}} v_x dt \tag{5.5}$$

Solving equation 5.5 leads to equation 5.6

$$x_{new} = x_{current} + v_x t_{ts} \tag{5.6}$$

This scheme is applied to all 3 spatial dimensions to obtain a new position vector for every element, this is shown in vector form in equation 5.7

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{new} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{old} + t_{ts} \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{5.7}$$

To determine the order of accuracy of the scheme, the scheme in the form given in equation 5.6 is taken. The value for X_{new} is then written as a Taylor series approximation centered on X_{old}, this is given in equation 5.8

$$x_{new} = x_{old} + x'_{old} t_{ts} + (\frac{x''_{old}}{2}) t_{ts}^2 + O t_{ts}^3 \tag{5.8}$$

Taking equations 5.6 from 5.8 yields equation 5.9

$$0 = (x_{old} + x'_{old} t_{ts} + (\frac{x''_{old}}{2}) t_{ts}^2 + O t_{ts}^3 + ...) - (x_{old} + v_x t_{ts}) \tag{5.9}$$

Equation 5.9 further simplifies to equation 5.10

$$0 = (\frac{x''_{old}}{2}) t_{ts}^2 + O t_{ts}^3 + ... \tag{5.10}$$

From equation 5.10 it can be seen that the leading error term is proportional to t_{ts}^2, hence Euler's method is first order accurate.

5.3 Quadratic Position Scheme

This scheme approximates the position at the next time step by use of a polynomial to estimate the position of an element at t+ts. A second second degree polynomial is to be used to interpolate the position of each element. In order to fully define a second degree polynomial 3 values of the polynomial are required. However to ensure the scheme converges on the exact solution, the derivative of the interpolating polynomial should be equal to the derivate of the exact solution. To maintain this condition the polynomial is instead mapped to two known positions, and an extra condition imposed relating its derivative to the velocity at the current time step. These conditions are shown in equations 5.11 to 5.13.

$$p(t) = At^2 + Bt + C \tag{5.11}$$

$$p(t - t_{ts}) = A(t - t_{ts})^2 + B(t - t_{ts}) + C \tag{5.12}$$

$$p'(t) = 2At + B \tag{5.13}$$

If we take the current time as $t = 0$ then these equations further simplify to equations 5.14 to 5.16

$$p(0) = C \tag{5.14}$$

$$p(-t_{ts}) = A(t_{ts})^2 + B(-t_{ts}) + C \tag{5.15}$$

$$p'(0) = B \tag{5.16}$$

Now the only variable not easily obtainable is A. Equation 5.15 can be solved for A, yielding equation 5.17

$$A = \frac{p'(0)}{t_{ts}} + \frac{p(-t_{ts}) - p(0)}{t_{ts}^2} \tag{5.17}$$

Substituting the coefficients A, B and C into equation 5.11 for $t = t_{ts}$ yields equation

$$p(t_{ts}) = \big(\frac{p'(0)}{t_{ts}} + \frac{p(-t_{ts} - p(0)}{t_{ts}^2}\big)t_{ts}^2 + p'(0)t_{ts} + p(0) \tag{5.18}$$

Expanding and smplifying equation 5.18 yields equation 5.19

$$p(t_{ts}) = p(-t_{ts}) + 2p'(0) \tag{5.19}$$

The future position of the element is then determined by $p(t_{ts})$, this scheme applied to all 3 spatial dimensions is shown in matrix form in equation 5.20.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{new} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{Previous-Time-Step} + 2t_{ts} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{5.20}$$

5.4 Quadratic Velocity Scheme

This scheme uses a LaGrange Interpolating polynomial to extrapolate the future velocity trend as a function of time which is then integrated over the time step period for all 3 spatial dimensions to estimate the position at the text time step. The general form of the LaGrange interpolating polynomial is given is equations 5.21 and 5.22

$$p(x) = \sum_{j=1}^{n} P_j(x) \tag{5.21}$$

$$p_j(x) = f(x) \prod_{k=i,k\neq j}^{n} \frac{x - x_k}{x_j - x_k} \tag{5.22}$$

Given 3 know previous know velocities v1, v2, v3 at t1, t2 and t3 respectively, the velocity-time curve interpolating polynomial is given by equation (3)

$$p(x) = \frac{(t-t_2)(t-t_3)}{(t_1-t_2)(t_1-t_3)}v_1 + \frac{(t-t_1)(t-t_3)}{(t_2-t_1)(t2-t_3)}v_2 + \frac{(t-t_1)(t-t_2)}{(t_3-t_1)(t_3-t_2)}v_3 \tag{5.23}$$

The time-step between iterations of the convection routine is fixed during the simulation, so the three know previous velocities occur at equally spaced time steps, given a fixed time step of t_{ts}, the values of t for which velocities are known are given in equation 5.24 where t is is the current time.

$$v_1 = v(t - 2t_{ts}), v_2 = v(t - t_{ts}), v_3 = v(t) \tag{5.24}$$

For each time step a new interpolating polynomial is calculated, and the function integrated over the range t to t+tts, to simplify the calculation the polynomial is instead centred around 0, for this case the known velocities reduces to equation 5.25, this polynomial is integrated over the range 0 to tts.

$$v_1 = v(2t_{ts}), v_2 = v(t_{ts}), v_3 = v(0) \tag{5.25}$$

Centring the polynomial on 0 reduces the complexity of equation 5.23 by removing the term x_1, equation 5.26 shows equation 5.24 with substituted values for t_1, t_2 and t_3

$$p(x) = \frac{(t + t_{ts})(t + 2t_{ts})}{2t_{ts}^2} v_1 - \frac{t(t + 2t_{ts})}{t_{ts}^2} v_2 + \frac{t(t + t_{ts})}{2t_{ts}^2} v_3 \tag{5.26}$$

Equation 5.26 further simplifies to equation 5.27

$$p(x) = \frac{v_1}{2t_{ts}^2}(t^2 + 3t * t_{ts} + 2t_{ts}^2) - \frac{v_2}{t_{ts}^2}(t^2 + 2t * t_{ts}) + \frac{v_3}{2t_{ts}^2}(t^2 + t * t_{ts}) \tag{5.27}$$

This can be expressed as a second degree polynomial of the form of equation 5.28.

$$p(x) = Ax^2 + Bx + c \tag{5.28}$$

Where the coefficients A, B and C are given by equations 5.29, 5.30 and 5.31 respectively

$$A = \frac{v_1}{2t_{ts}^2} - \frac{v_2}{t_{ts}^2} + \frac{v_3}{2t_{ts}^2} \tag{5.29}$$

$$B = \frac{3v_1}{2t_{ts}} - \frac{2v_2}{t_{ts}} + \frac{v_3}{2t_{ts}} \tag{5.30}$$

$$C = V_1 \tag{5.31}$$

The change in position of all the elements is calculated by the integration of this function over the range 0 to t_{ts}, this is shown in equation 5.32

$$\delta x = \int_0^{t_{ts}} v(t)dt = \int_0^{t_{ts}} (Ax^2 + Bx + c) = \frac{At_{ts}^3}{3} + \frac{Bt_{ts}^2}{2} + Ct_{ts} \tag{5.32}$$

Expanding equation 5.32 by replacing the coefficients of the polynomial a, b and c with substitutions from equations 5.29 through to 5.31 results in equation 5.33.

$$\Delta x = t_{ts} \left[v_1(\frac{5}{12}) + v_2(\frac{4}{3}) + v_3(\frac{23}{12}) \right] \tag{5.33}$$

This scheme is applied for all three spatial dimensions, this is shown in matrix form in equation 5.34

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{new} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{old} + t_{ts} \begin{bmatrix} v_{x1} & v_{x2} & v_{x3} \\ v_{y1} & v_{y2} & v_{y3} \\ v_{z1} & v_{z2} & v_{z3} \end{bmatrix} \begin{pmatrix} \frac{5}{12} \\ \frac{4}{3} \\ \frac{23}{12} \end{pmatrix} \tag{5.34}$$

6 Methods - General

6.1 Unity Implementation

The simulation is programmed in Unity using C-Sharp. C-Sharp is an object orientated programming language developed by Microsoft as part of its .NET program. However Unity implements the language through the Mono platform, this allows for Multi platform development, so the simulation can run on all x86 operating systems and some ARM based systems such as Android and IOS. The language is designed to be inherently similar to C or C++ to aid portability of code between the two languages, and the performance of these languages is also comparable. The use of C-Sharp was selected over other languages, such as Matlab, as it poses far high performance and features such as multi-threading, however this is at the cost of a less user friendly languages not orientated for scientific use.

Unity utilizes the open source OpenGL API as its rendering engine, as such the visual aspects of the simulation are handled quite naturally by Unity's internal functions. These functions form all user-interface and visualization aspects of the simulation.

6.2 Experimental Aparatus

All experiments were carried out on the same computer, the specifications are shown in table 1. The processor clock speed was limited to 17Ghz by disabling Intels Turbo Boost feature to ensure a fair test. All non essential background tasks were stopped.

Processor	Dual Core Intel Core I5-4210u 1.7GHz
Memory	8192MB DDR3
GPU	Intel(R) Internal HD Graphics
Operating System	Windows 8.1 64-bit

Table 1: Experimental Computer Specifications

6.3 Determining Real-Time

Throughout this section Real-Time is referred to, a time-step that refers to real-time is not easily defined. However throughout this report 60 frames per second (a time step of 0.02s) is referred to as the threshold for real-time. This time-step was used as a definition as it is what unity utilizes as its frame-rate. Unity, being a game engine, is designed to respond to inputs in real times. Hence 60 frames per second is a sensible definition of real time.

7 Methods - Convection Schemes

7.1 Testing Methodology

First a simple convection scheme will be tested. This scheme will implement none of the optimization methods discussed in this report and represents the full N-Body problem. This scheme is implemented and tested so the optimizations can be compared to to a result where no optimizations are present. Schemes will be assessed based on two parameters, their performance (computational cost) and accuracy

The performance of the schemes will be assessed by finding the time taken to calculate a set number of increments with a range of element counts. Hence, all variables are kept constant except element count, and the effect of this on computational cost is measured. From this data the performance of the schemes can be compared against the optimized case. However, by taking data over a range of element counts, the trend of how computational cost increases with element count can be found. The use of this is twofold. Firstly a smooth consist trend is expected, so noise or errors present in the data can be assessed. Secondly the theoretical trends presented in the theory section can be assessed for accuracy (I.E the biasing scheme should reduce to a ON problem).

As the schemes run through a set range of iterations, error will compound on the position of each element as a result of approximations made by the schemes. This will lead to a discrepancy between the positions calculated by a given scheme and the unoptimized case. The distance between these two positions is represented by the vector $\vec{D}$. To quantify the positional error, the magnitude of the vector from the unoptimized case to the optimized case is found, this is done via Pythagoras shown in equation 7.1, where subscripts N and S represent the position given by the scheme in question and the simple unoptimized case respectively.

$$|\vec{D}| = \sqrt{(x_N - x_s)^2 + (y_N - y_S)^2 + (z_N - z_S)^2} \tag{7.1}$$

As shown in equation 7.2, once the vector $\vec{D}$ is found it is divided by the magnitude of the vector from the initial point of a given element to its ending position during the simulation, this is denoted as the vector $\vec{M}$

$$\vec{E_{Element}} = \frac{\vec{D}}{\vec{M}} \tag{7.2}$$

This results in a measure where a value of 0 represents no impact on accuracy, and a value of 1 represents a case where the elements still inhabit their starting positions. This is still a measure for a single element, to quantify the error for a given situation the sum of these measures for all elements is found. This sum is then divided by the amount of elements, this is done so that a value of 1 still maintains its meaning of no movement from the starting position, this is shown in equation 7.3.

$$\vec{E} = \frac{1}{N} \sum \vec{E_{Element}} \tag{7.3}$$

Total accumulated error (error norms) was not selected as with increased element count the error may be seen to increase due to the increased fidelity even when the positional error on a given element decreases. By taking an error value relative to how far the element should have moved, the measure becomes less sensitive to the initial conditions used, as the error is proportional to where the element should have moved to. Hence, for an element whose position did not change significantly for the unoptimized case, but did change significantly for an optimized case, may give a low error vector, but using this measure would result in a large error value.

The initial conditions influences how the elements move during the iterations, hence selection of initial conditions is not arbitrary and should yield non-trivial results. The initial conditions chosen for this simulation were designed to emulate the shape of the wake the simulation is being developed for. This consisted of a planar grid with vorticity seeded at two parallel edges of the grid.

This poses two advantages. Firstly, the situation is close to the situation that is to be simulated. Secondly, it provides an ideal way to increase element count linearly. This is done by extending the planar grid in a single dimensions. For example going from a 10x10 grid to a 10x11 grid.

Each convective scheme has separate parameters that can be tuned to achieve a different balance of accuracy and performance, these are discussed in their respective sections.

7.1.1 Simple Convection - Unity Implementation

The simple convection code represents the unoptimized N-Body problem. It is the simplest implementation and provides a benchmark for the other schemes to be compared against.

The convective section of the vortex method is shown in listing 1, this section of code is responsible for implementing the Biot-Savart law. Elements are created in a planar grid shape, any element can be specified by its position on this grid. The position and vorticity of elements are stored in the 2d arrays $elementPositions[x, y]$ and $elementVorticity[x, y]$ respectively, the index of these arrays refer to the position of the given element on the initial grid.

The code works by cycling through the entire grid, this is done via the two for loops on lines 2 and 4. This procedure allows for every element to be selected. Once every element is selected, every other element must be selected, hence the grid must be cycled through again, this is performed by the for loops on lines 10 and 12. This results in a procedure that cycles through all element, for every element, hence it is necessary to check that for a given element, the routine has not selected itself for consideration. This is done via the evaluation on line 16.

Before the Biot-Savart law can be implemented the radius between the elements needs to be calculated, this is done on line 20. The Biot-Savart law is then implemented on line 21.

```
//for loop to cycle through all elements
        for (int xN = 0; xN < xSize; xN++)
        {
            for (int yN = 0; yN < ySize; yN++)
            {
                //Reset the velocity field
                elementVelocity[xN, yN] = Vector3.zero;

                //Now that every element is going to be cycled through, the elements need
                 ↪  to be cycled through again
                for (int xC = 0; xC < xSize; xC++)
                {
                    for (int yC = 0; yC < ySize; yC++)
                    {

                        //Check an element cannot convect with itself
                        if (xN != xC && yN != yC)
                        {

                                //Calcuate Influence
                                Vector3 r = elementPosition[xN, yN] - elementPosition[xC,
                                 ↪  yC];
                                elementVelocity[xN, yN] += (1f / (4f * Mathf.PI)) *
                                 ↪  (Vector3.Cross(elementVorticity[xC, yC], r) /
                                 ↪  (Mathf.Pow(r.magnitude, 2)));

                        }
                    }
                }

            }
        }
```

Listing 1: Unoptimized Convection Scheme Implemented in C#

7.1.2 Biasing - Tuning Parameters

Both distance based and vorticity based biasing schemes are to be tested. Firstly the computational time for a range of element counts will be found for the same range as the simple unoptimized case so that the data may be compared. The error will then be calculated in the previously mentioned way

To asses the associated increase or decrease in overhead from the schemes the computational time for the maximum grid size will be recorded for a range of biasing factors. The range of biasing factors are determined by taking 120% of the maximum value (maximum radius between two elements, or magnitude of vorticity of the convecting element). A range is specified with biasing factors more than the maximum value so that in the last section of the range the biasing value is not used. By doing this the increased overhead accountable solely due to the evaluation process can be quantified.

7.1.3 Biasing - Unity Implementation

To implement a biasing scheme the convective scheme presented in listing 1 was modified to include an additional evaluation before the Biot-Savart law is implemented. Both distance based and vorticity based biasing was implemented.

The distance based biasing implementation is shown in listing 2. The radius is calculated on line 2, this is used for the evaluation made on line 5

```
1 //Calcuate Influence
2  Vector3 r = elementPosition[xN, yN] - elementPosition[xC, yC];
3
4 //Check if the radius is small enough to be considered
5  if (r.magnitude < radiusBias)
6  {
7     elementVelocity[xN, yN] += (1f / (4f * Mathf.PI)) *
       ↪   (Vector3.Cross(elementVorticities[xC, yC], r) / (Mathf.Pow(r.magnitude, 2)));
8  }
```

Listing 2: Distance based biasing system implemented in C#

The implementation of a vorticity based biasing system is shown in listing 3. The evaluation is made on line 2. The implementation differs here in that the evaluation procedure relies upon a variable, $elementVorticity.magnitude$, that is not required for the implementation of the Biot-Savart law. Conversely, the distance based biasing scheme used the radius between the two elements, which was used in the implementation of the Biot-Savart law. Hence, it is expected that the vorticity based biasing system represents a larger overhead than the distance based biasing system.

```
1 //Bias based on vorticity
2 if (elementVorticity[xC,yC].magnitude < radiusBias)
3 {
4
5         //Calculate Radius
6         Vector3 r = elementPosition[xN, yN] - elementPosition[xC, yC];
7
8         //Implement Biot-Savart Law
9         elementVelocity[xN, yN] += (1f / (4f * Mathf.PI)) *
           ↪   (Vector3.Cross(elementVorticies[xC, yC], r) / (Mathf.Pow(r.magnitude, 2)));
10
11 }
```

Listing 3: Vorticity based biasing system implemented in C#

7.2 Fixed Cluster Size - Tuning Parameters

The fixed scale clustering system will be tested for a range of cluster sizes. The initial conditions and number of iterations tested are the same as for the simple convection

case, this allows for compariosn of performance and accuracy against the data sets. Computational time and accuracy will be assessed against the simple case. Three cluster sizes will be tested, $1x1$, $3x3$ and $5x5$.

A cluster size of $1x1$ was selected as this refers to a situation where no clustering is taking place, however the infrastructure is still in place to handle the clustering system. Therefore the increase in time taken over the unoptimized case is accountable to this additional infrastructure. Hence the discrepancy in these times can be used as a measure of how cumbersome the infrastructure is.

The sizes $3x3$ and $5x5$ were selected as they were the only sizes that gave a suitable resolution of data points over the range tested. The grid sizing used must be a multiple of the cluster size. For example, for a $5x5$ cluster 5 different grid sizes were used, $15x5$, $15x10$... to $15x25$. However for a $3x3$ cluster there are 7 possible grid sizes in the range.

As the grid size must be a multiple of the cluster size the accuracies cannot be assessed at a grid size of $15x20$ like the biasing scheme were assessed as the the $3x3$ cluster size cannot be used for this grid size. Instead the accuracies will be assessed at a grid sizing of $15x15$. Hence the values of error found for the biasing schemes are not perfectly comparable to those for this data set. However, as number of iterations are constant (which represents the major increase in error through the summation of truncation errors) and the influence of an element decreases with distance (which is increased with a larger grid) these error values should still be a good indicator.

7.2.1 Fixed Cluster Size - Unity Implementation

Listing 4 shows a fixed size cluster scheme implemented in Unity. The code has been compacted in order to be shown here (though no lines have been excluded), the full version is included in the appendix. Elements are specified in the same way as the simple convection scheme, by their initial grid position. This scheme implements square clusters based upon their grid position. The function was defined to be applicable to any situation, hence the cluster size or grid dimensions are not explicitly defined in the script, instead they can be defined and changed in the program with no need for the script to be modified.

The script works by first cycling through the clusters, this is done via the for loops on lines 2 and 4. This results in a procedure where all clusters are cycled through, the individual elements in every cluster need to be cycled through. This is achieved via an additional two for loops on lines 7 and 9. This results in a procedure where all elements are cycled through.

Now all elements are cycled through, they must be convected by either elements or clusters. Elements and clusters are handled separately. The current element is convected with every other element in its cluster, this is done between lines 14 and 27. This section of code is identical to the the simple convection script, however it is applied for only the elements in current cluster. The current element is then convected by every cluster except its own, this is done between lines 30 and 40. This code again is very similar to the simple convection script, however the only difference being $clusterPosition$ and $clusterVorticity$ are used instead of $elementPosition$ and $elementVorticity$.

```
//First Cycle through all clusters
for (int xC = 0; xC < (xSize/clusterSize); xC++)
{
for (int yC = 0; yC < (ySize/clusterSize); yC++)
{
//Now we need to consider every element in the clsuter
for (int xI = 0; xI < clusterSize; xI++)
{
for (int yI = 0; yI < clusterSize; yI++)
{
//We need to reset their velocities so they dont add up
elementVelocity[(xC * clusterSize) + xI, (yC * clusterSize) + yI] = Vector3.zero;
//Now we need to cycle through the individual elements in that cluster
for (int xIC = 0; xIC < clusterSize; xIC++)
{
for (int yIC = 0; yIC < clusterSize; yIC++)
{
//Now we need to check that we're not convecting the element with itself
if (xI != xIC && yI != yIC)
{
//now we need to find the radius
Vector3 Radius = elementPosition[(xC * clusterSize) + xI, (yC * clusterSize) + yI] -
 ↪  elementPosition[(xC * clusterSize) + xIC, (yC * clusterSize) + yIC];
//Now we can implement the biot-savart law
elementVelocity[(xC * clusterSize) + xI, (yC * clusterSize) + yI] += (1.0f / (4.0f *
 ↪  Mathf.PI)) * (Vector3.Cross(vorT[(xC * clusterSize) + xIC, (yC * clusterSize) + yIC],
 ↪  Radius) / Mathf.Pow(Radius.magnitude, 2));
}
}
}
//Now we need to convect the given element with the clusters
//First we cycle through all the clusters
for (int xCC = 0; xCC < (xSize/clusterSize); xCC++)
{for (int yCC = 0; yCC < (ySize/gridSpacing); yCC++)
{
//Now we need to check that we're not going to convect with the clsuter the element is
 ↪  inside of
if (xC != xCC && yC != yCC){
//Now we need to calculate the radius
Vector3 Radius = elementPosition[(xC * clusterSize) + xI, (yC * clusterSize) + yI] -
 ↪  clusterPosition[xCC, yCC];//Now we can implement the biot-savart law
elementVelocity[(xC * clusterSize) + xI, (yC * clusterSize) + yI] += (1.0f / (4.0f *
 ↪  Mathf.PI)) * (Vector3.Cross(clusterVorticity[xCC, yCC], Radius) /
 ↪  Mathf.Pow(Radius.magnitude, 2));
}
}
}
}
}
}
}
```

Listing 4: Fixed Cluster Size Convection Scheme Implemented in C#

7.3 Dynamic Clustering Scheme

The dynamic clustering scheme developed implements a routine where clusters may be clustered together to an infinite degree (limtied by memory constraints in reality). It makes use of a fractal recurring square division pattern to cluster elements

together, this is discussed further in the implementation section. The dynamic clustering scheme is unique in that unlike the fixed size clustering scheme, the ON^2 relation should be reduced to $Olog(N)$. To test this the simulation time over 500 iterations (as with the previous schemes) is recorded for a series of element counts. However unlike the simple and fixed size clustering schemes, the element count is increased by varying both grid dimensions instead of a single dimension. This is done so that the dynamic clusters, which are squares, may be exploited. Vorticity will be created in an identical manner as the previous schemes and accuracy will be asses for the closest possible situation (a grid size of 16x20 rather than 15x20).

7.3.1 Dynamic Clustering Scheme - Unity Implementation

The dynamic clustering schemes implementation is considerably more complex than the Simple and Fixed cluster size schemes, as the scaling cluster sizes necessitates the use of non-procedural programming in favour of recursive subroutines. Hence instead of presenting the code and discussing its function, the architecture of the implementation is presented and its sub functions are discussed in detail.

To explain the workings of the architecture, it is first necessary to explain the way in which elements are clustered and their respective nomenclature. The "degree" of clustering refers to how many abstractions are applied to the base system of elements. A degree of 0 represents no clustering, a degree of 1 represents a single abstraction of the elements into clusters of elements, and likewise an abstraction of 2 represents the use of a second abstraction where the clusers of elements form the first level of abstraction are clustered into clusters of clusters. This is shown in figure 19.

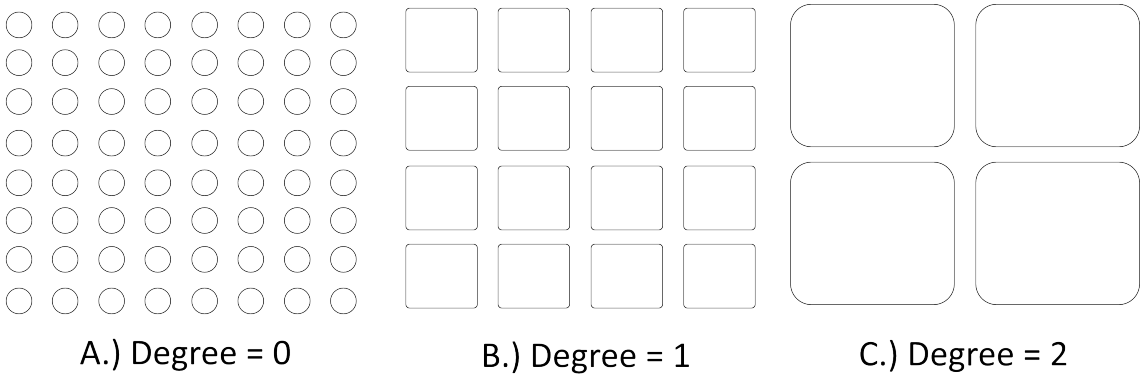

Figure 19: Demonstration of meaning of "degree" of cluster, a degree of 0 indicated only elements, a degree of 1 represtend cluster of 2x2 elements and a degree of 2 represents clusters of 2x2 clusters of degree 1

It is pertinent to note that all levels of abstraction under, and including the maximum degree of abstraction used exist. For example, if a degree of 2 is used, the abstraction layer for a degree of 1 is also calculated. Higher abstractions are always created using the same routine, hence a fractal like recurring pattern is seen if elements from higher degrees are "zoomed" in upon. This is shown in figure 20

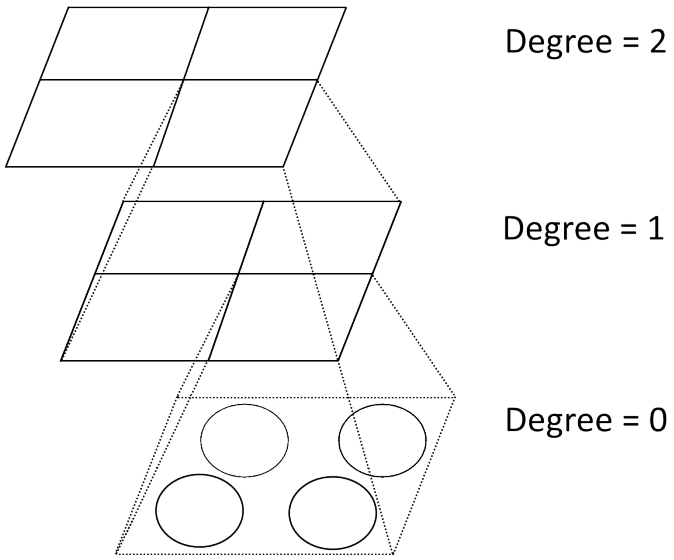

Figure 20: Fractal like pattern generated by numerous abstractions applied using the same procedure

Elements are convected by other elements and clusters of every level of abstraction. The maximum degree of abstraction is found by considering the largest degree of abstraction that can be used that results in more than one quadrant being present at the maximum degree. For example, the maximum degree of abstraction that can be used for a $8x8$ grid of elements is 2, this is because $2^2 = 4$, resulting in 4 quadrants present on the grid, however $2^3 = 8$, which would result in 1 quadrant which is not allowed.

The algorithm convects a single element by considering the maximum degree of abstraction and finding which quadrant the element belongs to. The element is then convected with every quadrant it is not part of, and the the quadrant it is part of is "zoomed" in on and the procedure repeated for that level of abstraction. This procedure recurs until all levels of abstraction up to and including 0 have been considered, and so all elements have been taken into account. This gives rise to a pattern of clusters for a given element as demonstrated in figure 21

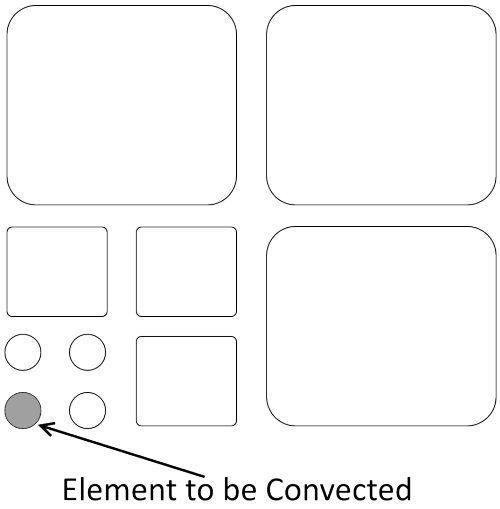

Figure 21: Clustering pattern for a given element

Because the implementation of such an algorithm is more complex than the previous schemes the separate steps in the algorithms were segmented into independent functions, called from a central function, "convectionDispatcher", which dispatches tasks to other functions. This gives rise to a single "black box" function that need only be called in the main loop. This is demonstrated in figure 22

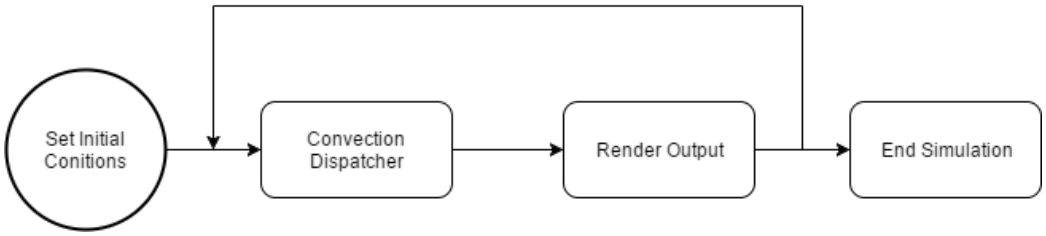

Figure 22: Example of "Black Box" nature of the dispatch function

The convection dispatcher function is shown in list 5. Upon inspection it is apparent that the function only calls other functions. The functions findVorticities() and updatePositions() are shared across all schemes and are included in the appendix. The function determineMaximumDegree() determines the maximum abstraction degree that results in more than one quadrant, this function is included in the main loop so that the element count can be varied during the simulation. The function determineCoefficients() uses the maximum abstraction degree to determine the vorticities and positions of all abstraction levels.

```
1  //Function that ties all the parts together
2      void convectionDispatcher()
3      {
4
5          //Perform Subroutines to find necessary values
6          findVorticities();
7          determineMaximumDegree()
8          determineCoefficients();
9          resetAllVelocities();
10
11
12         //Now we cycle through all elements and run the convection routine
13         for (int x = 0; x < xSize; x++)
14         {
15             for (int y = 0; y < ySize; y++)
16             {
17                 Convect(x, y, maxDegree, 0, xSize, 0, ySize, 0, 0);
18             }
19         }
20
21         //Now implement the discretizaton scheme
22         updatePositions();
23
24     }
```

Listing 5: Entire Convection Dispatcher Function

On lines 13 to 19, two for loops are present that cycle through every coordinate on the grid, the function Convect() is then called. The Convect() takes the the coordinate of the element and convects it with the appropriate elements and clusters of varying abstraction levels. The entire function is provided in the appendix, however its operation is described here in segments.

```
1      //This function finds the appropriate elements and abstracted clusters for the given
        ↪  element
2      void Convect(int xg, int yg, int degree, int xs, int xe, int ys, int ye, int xBias, int
        ↪  yBias)
3      {
```

Listing 6: Convect Function Declaration showing all input arguments.

The Convect() function takes 9 arguments, these are shown in list 6. The variables xg and yg specify the elements position on the grid to convect. The variable degree specifies the level of abstraction to be considered, in list 5 this is always set to the maximum level of abstraction possible when calling the function. The relevance of this will be discussed later. The variables xs, xe, ys, ye specify the area of the grid of elements to be considered, xs and xe being the x coordinate at the start and end of the grid segment and likewise ys and ye being the start and end of the y coordinates

of the grid segment. In reference to list 5, again it is seen that these are set so that they specify the entire grid. This is shown in figure 23. The variables $xBias$ and $yBias$ are discussed later when their relevance becomes apparent.

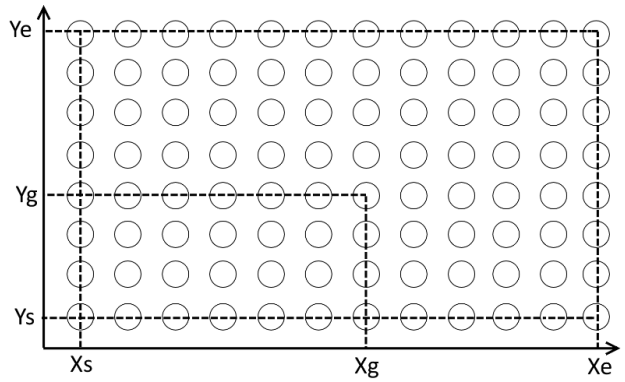

Figure 23: Physical representation of variables used in the Convect() function

The first step for the convective function is to determine the size of the grid it is considering, given the specified level of abstraction. This section of the code is shown in list 7. On line 2, the size of clusters is found in terms of elements, for example a 2x2 cluster of elements has a spacing of 2, though a 2x2 cluster of clusters of degree 1 has a spacing of 4 ($2x2x2$). Once the size of the clusters in terms of elements is known, the size of the grid to be considered is then dived by this spacing to yield the number of segments present, this shown on lines 10 and 11. The evaluation on line 8 handles an anomoly at the boundary (abstraction = 0) where the grid size calcultion returns unity.

```
1         //determine grid spacing
2         int spacing = (int) Mathf.Pow(clusterSize, degree);
3
4
5         //first step is to determine the current cluster size
6         int xGridSize = 0;
7         int yGridSize = 0;
8         if (degree != 0)
9         {
10            xGridSize = (xe - xs + 1) / spacing;
11            yGridSize = (ye - ys + 1) / spacing;
12        }
13        else
14        {
15            xGridSize = 2;
16            yGridSize = 2;
17        }
```

Listing 7: Section of the Convect() function that determines the current grid size given the level of abstraction

At this point the Convect function has taken a segment of the grid and divided it up into quadrants given the current. Hence the grid representation shown in figure 23 has transformed to figure 24.

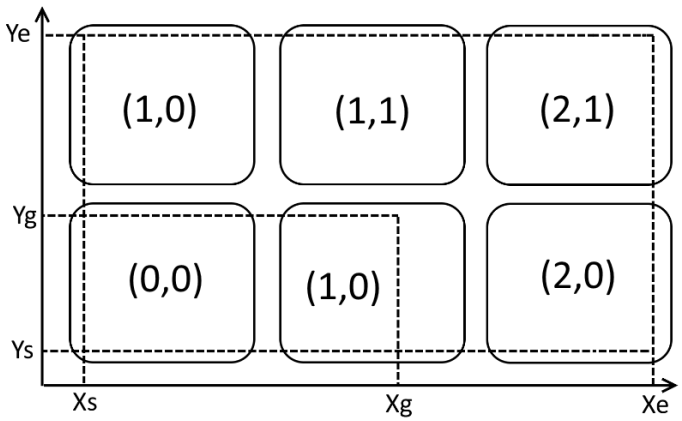

Figure 24: Result of the Convect() function splitting the considered grid segment into clusters

The next step of the Convect function is to determine which of these quadrants the element to be convected (at location xg,yg) lies within. This is shown in list 8

```
1  //determine current position on the grid
2          int xgp = 0;     //start by declaring variables for our position on the "local" grid
3          int ygp = 0;     //And againt for the y position
4          //for for loops to determine their values
5          for (int x = 0; x < xGridSize; x++)
6          {
7              if ( xg >= (xs + x*spacing) && xg < (xs + (x + 1) * spacing))
8              {
9                  xgp = x;
10             }
11         }
12         //now determine the y coordinate
13         for (int y = 0; y < yGridSize; y++)
14         {
15             if ( yg >= (ys + y * spacing) && yg < (ys + (y + 1) * spacing))
16             {
17                 ygp = y;
18             }
19         }
```

Listing 8: Section of the Convect() function that determines the grid segment the element to be convected lies within

Now the quadrant that the element lies in is known, so the element in question can be convected with the segments it is not in, and the segment it is in should be considered at the next lowest abstraction level. This is shown in list 9

```
1  //Now cycle through clsuters
2          for ( int x = 0; x <  xGridSize; x++)
3          {
4              for ( int y = 0; y < yGridSize; y++)
5              {
6                  if ( x == xgp && y == ygp)
7                  {
8                      //Check if we're at the base level abstraction
9                      if (degree > 0)
10                     {
11                         Convect(xg, yg, degree - 1, xs + (xgp * spacing), xs +
                            ↪  (xgp+1)*spacing, ys + (ygp * spacing), ys + (ygp + 1) *
                            ↪  spacing, (xBias + x) * clusterSize, (yBias + y) *
                            ↪  clusterSize);
12                     }
13                 }
14                 else
15                 {
16                     biotSavart(xg, yg, x + xBias, y + yBias, degree);
17                 }
18             }
19         }
```

Listing 9: Section of the Convect() function that either calls the Biot-Savart function or calls itself to consider a lower abstraction level

In list 9 two for loops (on lines 2 and 4) are used to cycle through the coordinated of all grid segments in figure 24. Inside these for loops an evaluation is made to determine whether the current cluster in question contains the element to be convected. If the element is not in that cluster, the function biotSavart() is called and the element is convected with these clusters.

If the element is in the cluster, the situation is more complex. First another evaluation is made, this is shown on line 9. This evaluation determines whether or not the lowest (0) abstraction layer is being considered. If the lowest abstraction level is being considered then nothing needs to be done. However if the element that is to be convected is still in a cluster that cluster then needs to be considered further. This is done by calling the Convect() function from within the Convect() function.

The second call to the Convect() function can be seen on line 11. The arguments passed to this instance of the COnvect() function are different from the initial instance of the function. The values of xg and yg passed to the new function are constant. However the values for xs, xe, ys and ye now reflect the coordinates of the segment the element was present in. This is demonstrated in figure 25.

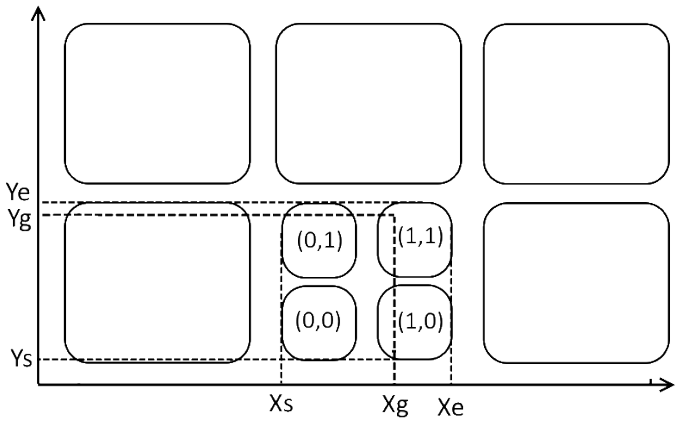

Figure 25: Result of the Convect() function splitting the considered grid segment into clusters

Values for $xBias$ and $yBias$ are also passed to the new instance of the Convect() function. These values represents the x and y coordinate of the current cluster multiplied by the cluster spacing. Physically this represents the coordinates of the starting point of the new set of clusters. In the example shown in 25 this would refer to the coordinate of the cluster $(0, 0)$ however on a local global, the the coordinate $(2, 0)$ would be given (see figure 26).

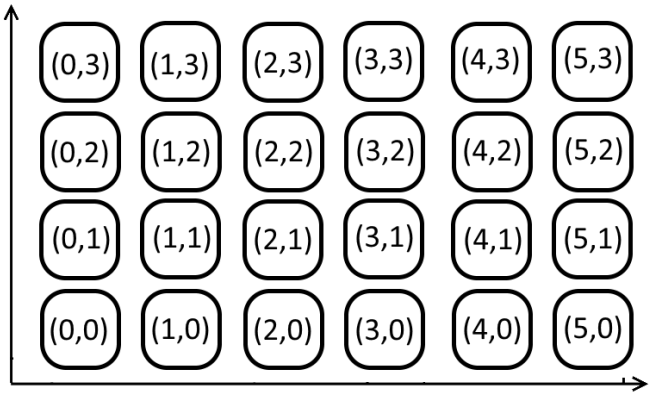

Figure 26: Global coordinates of clusters for a single level lower abstraction

These global coordinates are passed to the biotSavart() function when a element is to be convected by a cluster. The biotSavart() function is shown in listing 10. Its implementation is identical to the Simple Convections scheme, with the except of the added dimension to the position, and vorticity arrays to account for the degree of clustering.

```
//This function implements the biot-savart law maybe
    void biotSavart(int ex, int ey, int cx, int cy, int degree)
    {

        //This calculate the influence
        //First the radius,
        Vector3 radius = elementPosition[ex, ey, 0] - elementPosition[cx, cy, degree];
        //Now the Biot-Savart law itself
        elementVelocity[ex, ey] += (1f / (4f * Mathf.PI)) * (Vector3.Cross(vorT[cx, cy,
         ↪  degree], radius) / (Mathf.Pow(radius.magnitude, 2)));

    }
```

Listing 10: Implementation of the Biot-Savart law for the dynamic clustering scheme

8 Methods - Discretization Schemes

8.1 Unity Implementation

All schemes were programmed as self contained functions, the only input allowed to every function was the ID of an element. Likewise, the output of every function was a 3 dimensional vector for the new position of the element. This turns the schemes into a series of "black boxes" which have identical inputs and outputs. As the input and outputs are identical no modification need be made to any other part of the code in order to use a different scheme, as they are perfectly modular. All schemes have access to a series of shared variables and their own unique variables required. The variables shared for all schemes are shown in listing 11. To keep the schemes modular each scheme uses the same data types for inputs and output. Unity's Vector3 variable class is used for all positions and velocities and a floating point variable is used for the time step. The full script is provided in the appendix however snippets of important code and shown and explained here.

```
1  //Shared Variables for all Schemes
2  private Vector3[] elementPositions = new Vector3[255];
3  private Vector3[] elementVelocities = new Vector3[255];
4  public float timeIncrement = 2.0f;
```

Listing 11: Variable Declarations for all Discretization Schemes

Eulers method is implemented via the function shown in listing 12. Euler's method is implemented on line 10, lines 6 and 7 obtain the current position and velocity from the shared variables.

```
1  //Function to Iterate Eulers Method
2  public Vector3 EulerIterate(int elementID)
3  {
4
5      //Get Required Information from the Element ID
6      Vector3 currentPosition = elementPositions[elementID];
7      Vector3 currentVelocity = elementVelocities[elementID];
8
9      //Apply Eulers Methods
10     Vector3 newPosition = currentPosition + (timeIncrement * currentVelocity);
11
12     //Return calcualted new position
13     return newPosition;
14
15 }
```

Listing 12: Eulers Discretization Method Implemented in Unity

The implementation of the Quadratic Velocity scheme requires additional infrastructure compared to the implementation of Euler's method as the previous position needs to be known. The additional variable declaration are shown in list 13. Two new variables are shown here, the $elementPastPositions$ array holds the position of an element at the last time step. The boolean array $eulerMode$ is used to determine whether Eulers scheme should be used for a given element so that previous position data can be generated.

```
1  //Variables Unique to the Quadratic Position Scheme
2  private Vector3[] elementPastPositions = new Vector3[255];
3  public bool[] eulerMode = new bool[255];
```

Listing 13: Variable Declarations for the Quadratic Position Scheme

The function to implement the quadratic position scheme is shown in list 14. The Inclusion of an "if" statement can seen on lines 8 to 17 to determine whether to use Euler's method or the Quadratic Position Scheme. The Quadratic Position scheme can be seen implemented on line 14, this is in contrast to line 10 in list 12. Comparison of both implementations reveals very similar computation, with the Quadratic Position scheme likely exhibiting lower performance due to the additional multiplication in the solution, the added evaluation step and the need to define the variable newPos with an initialization value before the evaluation (a requirement of C# to set the scope of the local variable).

```
1  //Function to Iterate using the Quadratic Position Scheme
2  public Vector3 QPIterate(int eID)
3  {
4
5      Vector3 newPos = new Vector3(0.0f, 0.0f, 0.0f);
6
7      //Determine whether Euler's method should be used
8      if (eulerMode[eID] == true)
9      {
10          newPos = EulerIterate(eID);
11          eulerMode[eID] = false;
12     }
13     else
14     {
15         newPos = elementPastPositions[eID] + (2f * timeIncrement * elementVelocities[eID]);
16     }
17
18     return newPos;
19 }
```

Listing 14: Quadratic Position Discretization Scheme Function Implemented in Unity

The Quadratic Velocity scheme required its own set of unique variables to be implemented. As the Quadratic Position scheme required a past position, the Quadratic

Velocity scheme required 2 previous velocities for every element. Similiarly, the Quadratic Velocity scheme also needs a mechanism to determine whether Euler's method should be initiated. The variable declarations unique to the Quadratic Position Scheme are show in list 15.

```
1  //Variables Unique to the Quadratic Velocity Scheme
2  private Vector3[,] elementPastVelocities = new Vector3[255, 2];
3  public int[] eulerCount = new int[255];
```

Listing 15: Variable Declarations for the Quadratic Velocity Scheme

The implementation of the Quadratic Velocity scheme is shown in list 16. The new position is calculated on line 21, otherwise the code is mostly identical to the Quadratic Position scheme. However it differs in that Euler's Method must be called twice, not once.

```
1   //Function to Iterate using the Quadratic Velocity Scheme
2      public Vector3 QVIterate(int eID)
3      {
4
5          //Declare variable with null value to define local scope in function
6          Vector3 newPos = new Vector3(0.0f, 0.0f, 0.0f);
7
8          if (eulerCount[eID] != 0)
9          {
10
11             //Call Function to Iterate via Eulers Method
12             newPos = EulerIterate(eID);
13
14             //reduce the ammont eulers method needs to be used
15             eulerCount[eID] -= 1;
16
17         }
18         else
19         {
20             //Use the QV Scheme to find new position
21             newPos = elementPositions[eID] + timeIncrement * ((5f / 12f) *
               ↪   elementPastVelocities[eID, 0]-(4f/3f)* elementPastVelocities[eID,
               ↪   1]+(23f/12f)*elementVelocities[eID]);
22         }
23
24         //shift past velocities array
25         elementPastVelocities[eID, 0] = elementPastVelocities[eID, 1];
26         elementPastVelocities[eID, 1] = elementVelocities[eID];
27
28         return newPos;
29
30     }
```

Listing 16: Quadratic Velocity Discretization Scheme Function Implemented in Unity

The entire script used to test the computational time of the dsicretization schemes is included in the appendix. An example showing the important mechanics is shown in

list 17. On lines 2 and 13, the current time is taken by using Unity's Time.realtimeSinceStartup class which returns the current time since the program first executes in seconds. Between these two time readings is a "for" loop containing only a call to the method of a scheme to be tested. Hence the difference in these two readings is accountable to the time taken to calculate the given number of iterations, the difference is calculated on line 8.

The scheme is used to set the value of the Vector3 "testPosition", this is not the position used by the schemes themselves so the position the scheme "sees" does not change throughout the iterations. The velocity vectors are also not changed throughout the iterations, so the inputs and outputs of the schemes are constant throughout the iterations. This ensures no change in measured performance are accountable to different values for position and velocity being used.

```
1  //Take initial time reading
2              var initialTime = Time.realtimeSinceStartup;
3
4              for (int itNo = 0; itNo < iterationCount; itNo++)
5              {
6
7                  //Use QP Scheme
8                  testPosition = QPIterate(eID);
9
10             }
11
12             //Take end time
13             var endTime = Time.realtimeSinceStartup;
14
15             //return time take
16             iterationTime = (endTime - initialTime);
```

Listing 17: Section of Script to Time Discretization Schemes

8.2 Testing Methodology

To asses the accuracy of the proposed schemes, approximations were made to a known function. The selection of a known function to approximate is problematic as the different schemes have different propensities to model different functions. For example, the Quadratic Position Scheme models the position-time relationship as a quadratic function, thus it will perfectly model such a function. Likewise, the Quadratic Position Scheme models the position-time relationship as a third degree polynomial, and thus would model such a function quite naturally.

The function chosen to asses the schemes was $f(x) = e^x - 1$, this function was chosen

as none of the schemes can naturally model this relationship. A cyclic function such as $f(x) = sin(x)$ was not chosen so that the accumulation of truncation errors could more easily be assessed.

To model the function $f(x) = e^x - 1$ using the schemes an initial value of the function at $x = 0$ was given, resulting in $f(x) = 0$. A value for the derivate at $f(0)$ was given and is analogous to the velocity at the current time step. Using this information (and previous information for the implicit schemes) the schemes are used to make a prediction for the value of $f(t_{ts})$. The value of $f(t_t)$ approximated by the schemes refers to an actual value of $f(x)$, the x value this value of $f(x)$ refers to is then taken and used to find the value of $f'(x)$ at this point, and this is used as the current velocity, and the process is then repeated over a set number of iterations. These calculations were performed in Matlab

As previously discussed, the implicit schemes require information about the previous states of the function. Practically this is resolved by using Euler's Method for the required amount of initial iterations. However for evaluation purposes the schemes are seeded with real values of previous states of the known function. This is done so that comparisons can be made on data accountable solely to the individual schemes.

Approximations were made to $f(x) = e^x - 1$ over the range $0 \leqslant x \leqslant 5$, over this range the schemes were evaluated for 5 different time steps of $t_{ts} = 0.02$, 0.12, 0.22, 0.32 and 0.42. These time steps were used as they have equal increments and the lowest of the range is 0.02s, Unity's default time-step. Hence convergence for a practical scheme should be obtained by this minimum time step. The range over which the function was evaluated gave sufficient points (250 for $T_{ts} = 0.02s$) to evaluate the accumulated truncation error in a time range similar to what an element may exist for in the simulation.

The convergence of the scheme was tested purely mathematically, however, to test the performance of the schemes they need to be tested running in unity. To do this a script was written to record the time taken for a given amount of iterations. The scheme is required to run many times a second, so the time taken to calculate should be in the order of milliseconds. Recording the time for a single iteration is therefore not preferable, as on this time scale measurements are highly susceptible to noise due to background processes running on the computer.

To minimize these sources of error only Unity was run at the time of testing and all but essential background processes were closed via windows task manager. A range of data points were taken for constant inputs into the schemes but with a differing number of iterations. The range over which iterations were taken was determined by testing the amount of iterations of Eulers Method where the time taken to complete was around 10 seconds, this time range was then split into 50 intervals and tested at each interval.

A magnitude of around 10 seconds was selected as it is sufficiently large enough that the aforementioned noise shouldn't be problematic, and the computational time is not excessive such that to gather all data points becomes cumbersome. A range of 250 increments was selected so that the trend could be accurately captured. The resolution of this range was also fine enough such that any background noise, due to a notification or likewise, was captured and the test could be re run.

As the maths performed is constant, the calculation time should increase linearly with the amount of iterations, from the closeness of agreement of the data points to a linear trend the noise in the data can be assessed. The schemes may then be compared to relative to each other by comparison of the slopes of these linear trends.

9 Results & Discussion - Convection Schemes

9.1 Simple Convection

The simple convective scheme was run for 500 iterations at a time step of 0.02s, Eulers method was used. The number of iterations were selected at this point as sufficient "curling up" of the vortex sheet was present, this is demonstrated in figure 32

Figure27 shows the rise in computational cost with increased element count. A trend is clear throughout all data points visually. Further the data points show very high agreement to a quadratic trend, with a correlation of $R^2 = 0.999$. This confirms that the simple unoptimized vortex method does represent an N-Body problem.

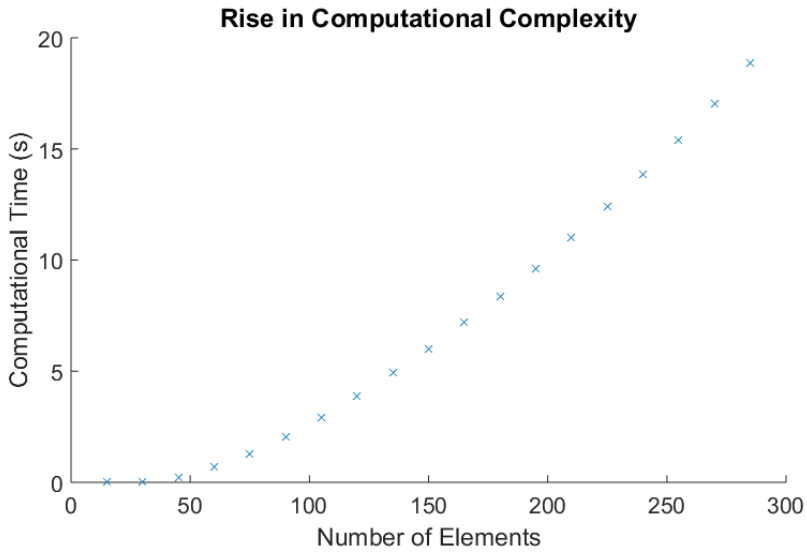

Figure 27: ON^2 Increase in Complexity Demonstrated for an Unoptimized Case

For the maximum grid enlargement, 15 by 30 elements, the positions of all elements were recorded for the final increment so that the results form other schemes could be compared against this. The computational time for the maximum grid size was 21 seconds

9.1.1 Biasing Schemes

Figure 28 show the results of increasing the biasing radius on computational time, values were found for the maximum grid size (15 by 30). A constant time of $T = 21s$ is plotted on the same axis, this refers to the time taken without the biasing scheme implemented. The radius is dimensionless (the simulation has no inherent units),

though the maximum radius encountered for the grid is 25 and the range tested is 0-30, representing 120% of the maximum radius, hence any trends apparent are captured in the data.

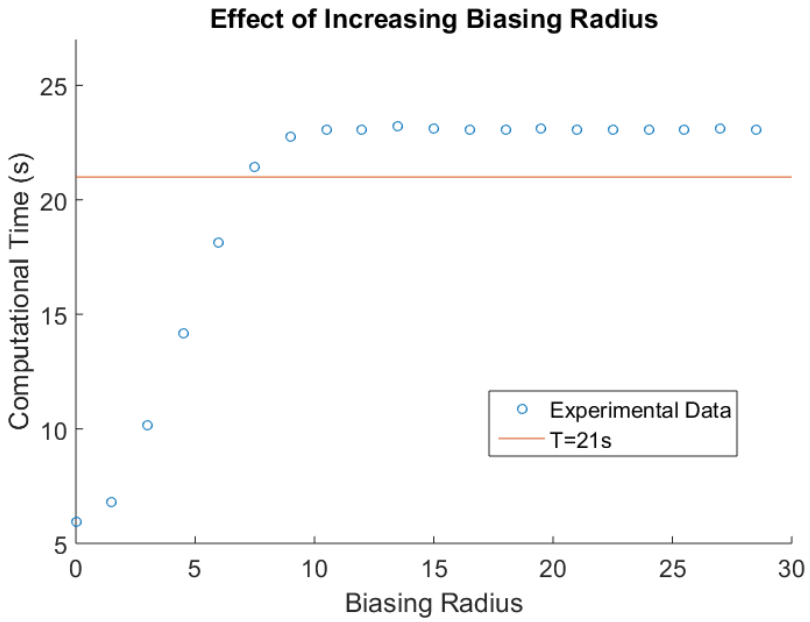

Figure 28: Increase in computational cost with increased biasing radius

The computational time is initially far lower with the biasing than without, however the computational time increases quickly as the biasing radius increases, and the two are equal around $r = 7$. Whn the computational time is below around 21s represents an optimization, however the computational cost is seen to increase past the cost of implementing no biasing past this radius. Hence use of a biasing scheme past this point represents a loss in accuracy and decrease in performance. The data appears to settle on a constant value (from around 15 to 30) of around 23, the difference between this and the $T = 21s$ represents the additional overhead from implementing the evaluation procedure of the biasing scheme.

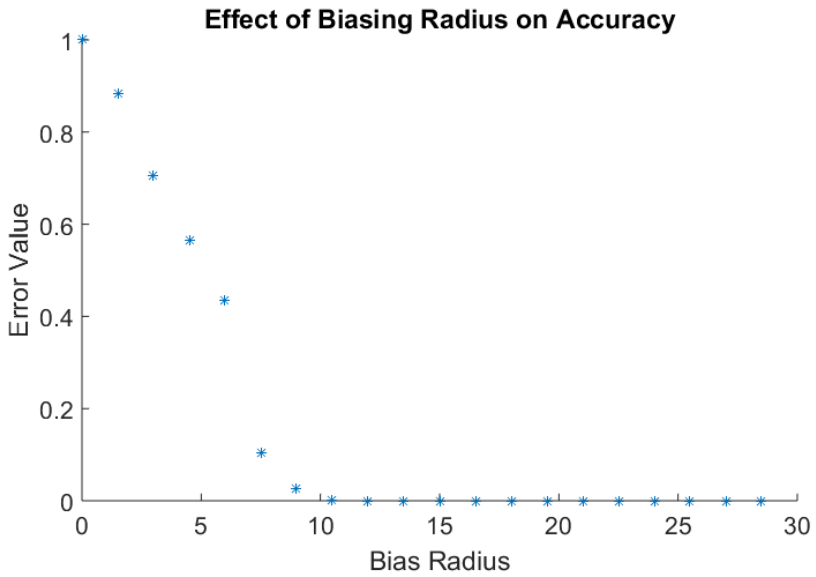

Figure 29: Effect of increasing biasing radius on accuracy relative to Unoptimized case

Figure 29 shows the effect of increasing the biasing radius on the accuracy (as defined in section) of the simulation. A steep decrease in error is seen immediately as radius is increased. This decrease appears to occur linearly (and shows a correlation of $R^2 = 0.98$), however this is purely observational and further analysis is required to explain this. The error is seen to decrease to negligible levels before the maximum range is reached. Hence the error is reduced to negligible levels when all elements are not taken into account.

For the range of values of the Bias Radius which represents an optimization (roughly 0-7), the error values seen to range from about 0.9 to 0.1. The lowest of this range, 0.1, still represents an incurred error of about 10% of the positions of all elements. Hence Biasing methods may be used to optimize a simulation however their effect on accuracy make their use limited for this evaluation criteria.

An interesting phenomena was noted during the data processing. The error reduced to zero before the biasing radius had reached the end of its range. An error of exactly zero indicates no difference between the biased case and an unoptimized case. This of course is mathematically impossible, instead this represents an error so small its influence is not captured by the 6 decimal point precision of C#'s $float$ type variable. This outlines how small of an influence some elements have on others, with their influence being so small it cannot accurately be numerically captured without

use of a double precise floating point variable. This indicates there is certainly a good argument for the use of biasing, however more computationally efficient biasing schemes must be employed to see any benefit.

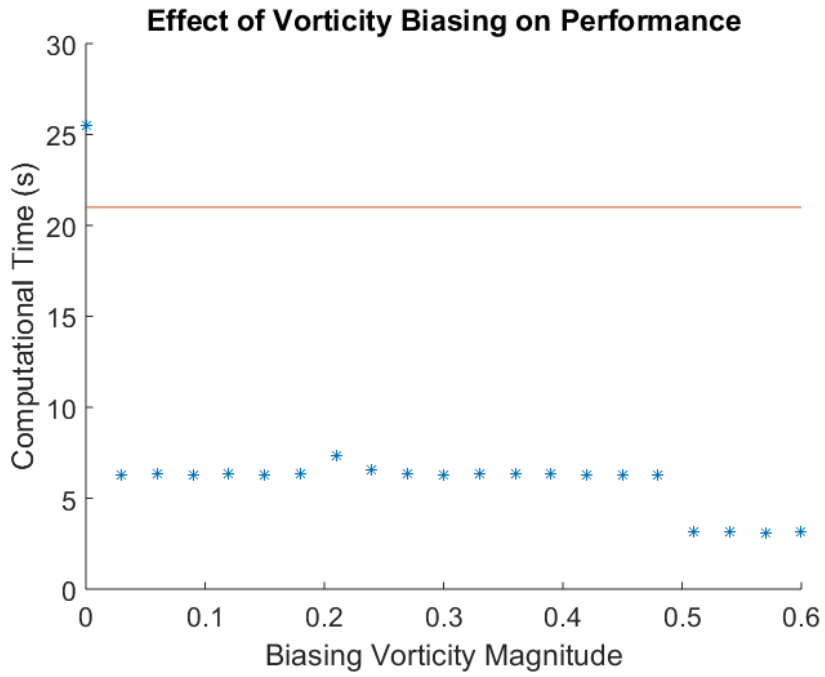

Figure 30: Effect on performance of using a vorticity biasing scheme

Figure 30 shows how the computational time varies with increasing the vorticity bias. The graph is characterised by two regions of constant computational time separated by a discontinuity. This continuity occurs at $|\omega| = 0$. The discontinuity is a result of the initial conditions used, the magnitude of the vorticity of elements, which is either $|\omega| = 0$ or 0.5. Hence the first constant section (around $T = 5s$) refers to a situation where only non zero vorticity elements are taken into account. Likewise, the situation after the discontinuity (around $T = 3s$) represents a situation where no elements are taken into account.

At $|\omega| = 0$, all elements are taken into account. The computational overhead given by this point is significantly increased over the unoptimized case an is higher than the equivalent case for the radius biasing scheme. This is expected, for the vorticity biasing scheme the magnitude of the vorticity must be found for the evaluation, which represents an increased overhead. Likewise the distance based biasing requires the

magnitude of the distance vector to be found, however this value is then used in the Biot-Savart law, so its computation does not result in an increased overhead.

Whilst the results may seem promising in figure 30, it is important to interpret them in the context of the initial conditions used. The initial conditions used were designed to emulate the simulation running in a steady state, so there is no vorticity seeded on the inner rows of elements. However when lift is suddenly generated there is vorticity on these inner elements. Given such initial conditions, a continuous trend would be exhibited for the computational times and in retrospect may have been more appropriate. However, from the current data it can be concluded that the evaluation procedure for a vorticity based biasing method represents a larger computational overhead than distance biasing.

Vorticity based biasing schemes may also become inappropriate in situations where low vorticity elements become close to each other and thus become very influential. This, combined with the increased overhead compared to distancing biasing makes them inappropriate, in their current form, for use in the simulation.

9.2 Predictive Fixed Cluster Scale

Figure 31 shows the computational times of the fixed size cluster scheme for the same conditions as previously tested. The data from the unoptimized case is included for comparison. At a grid spacing of $15x20$ (referring to 300 elements) the computation time is seen to roughly double for a cluster size of $1x1$ compared to the unoptimized case. This represents the increased overhead from the addition of calculating cluster groupings and cycling through individual elements In an inefficient way. However, this should become more efficient as the the cluster size increases.

This is definitely exhibited in the data, as the data points referring to cluster sizes of $3x3$ and $5x5$ are significantly lower than the $1x1$ cluster, being around a fourth at 300 elements. They are also lower than the unoptimized case notably. At a grid size of 225 (the only number of elements where data exist for all data sets) the two clustering schemes have times of $8.1s$ and $4.6s$ for the $3x3$ and $5x5$ respectively. Compared to the time from the unoptimized case for this element count of $11.6s$ an optimization it terms of performance has definitely been achieved.

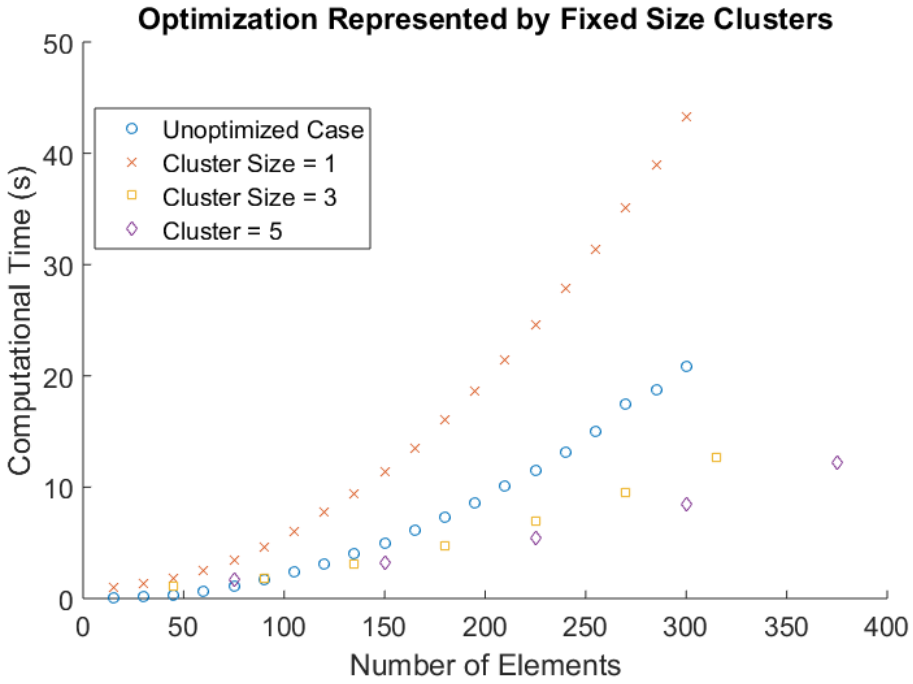

Figure 31: Relationship between cluster size and computational time

To determine whether this optimization is useful, its effect on accuracy must be considered. The errors for the for the three different cluster sizes are shown in table 9.2. The error for a cluster size of $1x1$ was determined in order to confirm that the it gave results equal to that of the unoptimized case. As the error is zero this has been confirmed and thus the implementation of the clustering scheme is giving the expected mathematical results

Cluster Size	1x1	3x3	5x5
Error	0	0.088	0.110

Table 2: Calculated Errors for a 15x15 grid for three different cluster sizes.

The errors for the $3x3$ and $5x5$ cluster sizes are remarkable, an optimization has been achieved in terms of performance, and the trade off for accuracy is notably improved over the biasing methods. Further analysis of the element positions visually revealed that the errors were largely caused by elements on cluster boundaries. Figure 32 shows the position of the elements predicted by the unoptimized scheme and cluster sizes of $3x3$ and $5x5$ for 500 iterations.

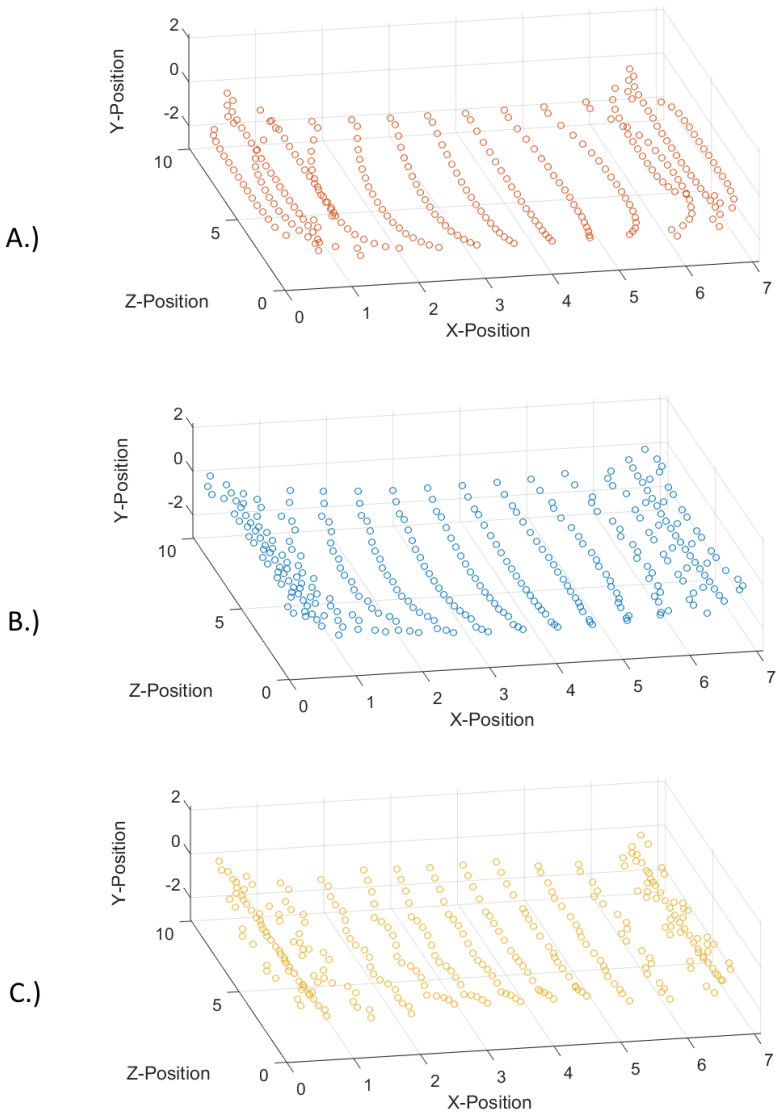

Figure 32: Final positions of elements for the unoptimized case (A), cluster size of $3x3$ (B) and $5x5$ (C)

It can be be seen qualitatively that the jump from no clustering to a $3x3$ cluster imposes inaccuracies. This can be seen exhibited in the curvature of the central rows of elements running along the z direction. The difference is notable between the unoptimized case and the $3x3$ cluster sizing. However the for the $5x5$ cluster size large deviations from the unoptimized case are seen and the cluster boundaries are clearly visible along the central elements spanning across the Z direction. This is likely caused by elements inside of a cluster being on that clusters boundary. This would cause the most influential elements to that element, the neighbouring elements, being treated as a mix of elements (neighbours in its own cluster) and clusters (neighbours

in a neighbouring cluster).Hence this problem could be resolved via treating the the neighbouring clusters as individual elements as well.

The edges of the data points, where elements have vorticity, are seen to roll up. There is significant deviation between the unoptimized case the and the clustered data sets. This is likely due to the vorticity being approximated to act at a different location for each case. The use of a weighted average approach to determining cluster position would likely reduce this error.

9.3 Dynamic Cluster Size

Figure 38 shows the computational times of the dynamic cluster size scheme on the same axes as the unoptimized case. A remarkable decrease in computational time is shown represented by the dynamic cluster scheme compared to unoptimized case. Given a computational time of $5.5s$, the dynamic clustering scheme processed 1024 elements, whilst the unoptimized case handled 139 elements in $4.9s$. Hence it is clear that the dynamic cluster scheme represents a definite increase in performance.

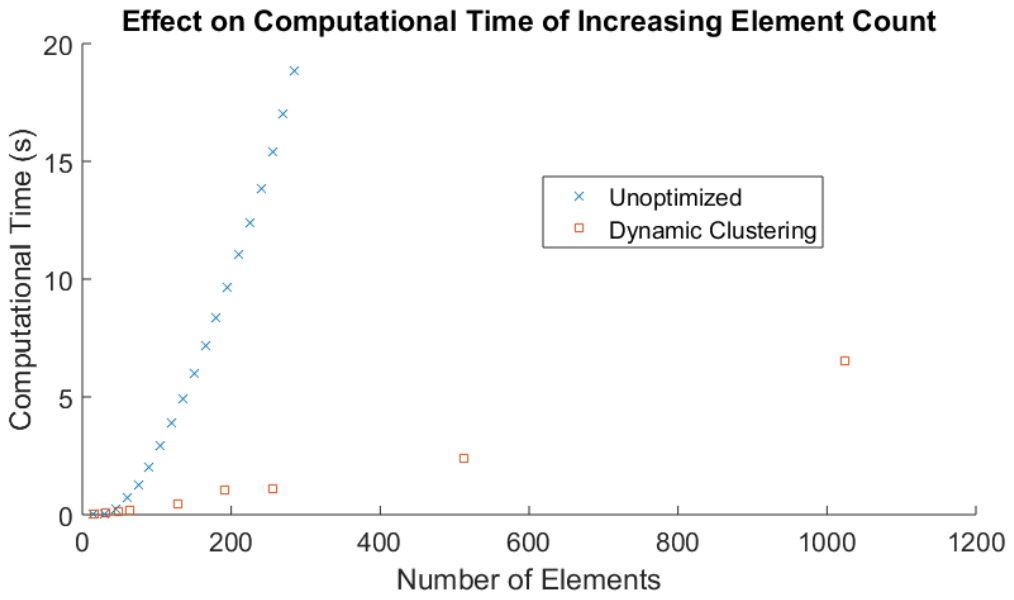

Figure 33: Increase in computational time given increased element count for the unoptimized case and dynamic cluster scaling

The performance increase from the dynamic clustering scheme is larger than that of the fixed sized cluster scheme significantly. Further the ON^2 relation appears to have been reduced as well. Whether the predicted relation of $Olog(N)$ has been

exhibited is hard to asses however a least squares regression line of the data points shows a correlation coefficient of 0.99.

The accuracy for the dynamic clustering scheme was 0.051 after 500 iterations with a grid size of 16x20. This is lower than both fixed cluster sizes and the computational time was significantly lower. Hence the Dynamic clustering scheme exhibits better performance than the fixed size clustering scheme, and results is less error.

10 Results & Discussion - Temporal Discretization

Eulers method of discretization is considered first, figure 34 shows Eulers schemes approximation to the function $f(x) = e^x - 1$ for 4 different time steps; in the range $0 \leqslant x \leqslant 5$

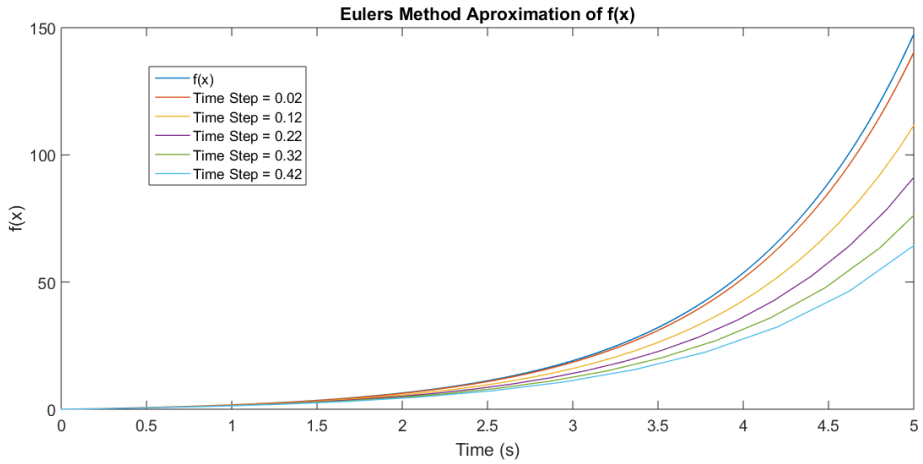

Figure 34: Eulers method used to approximate $f(x) = e^x - 1$ for 5 different time steps

From figure 34 it is apparent that refining the time step increases the accuracy of the solution from the scheme. Hence Eulers method shows convergence upon $f(x)$ and would provide an accurate solution for the simulation for the current time step ($t_{ts} = 0.02s$). The percent error from each time step are shown in table 10.

Time Step (s)	0.42	0.32	0.22	0.12	0.02
Percent Error %	54.4	42.4	34.3	20.8	4.8

Table 3: Error at $x = 5$ for Eulers Method at 5 Time Steps

At the default time step used by Unity, $t_{ts} = 0.02s$ the error returned by Eulers method is around 5% for a 5 second range. The time an element is likely to exist for in the simulation is likely in the same magnitude as this, but considerably larger, hence the positional accuracy approximated by Eulers scheme will be larger than 5%. This error would further propagate through the simulation as it affects the velocities calculated for all other elements.An error of around %5 may be tolerated for the simulation to be able to run in real time, however it is far from ideal.

The next scheme to be considered is the quadratic position scheme, figure 35 shows the approximations to $f(x) = e^x - 1$ for the same time steps and range as Eulers method was examined.

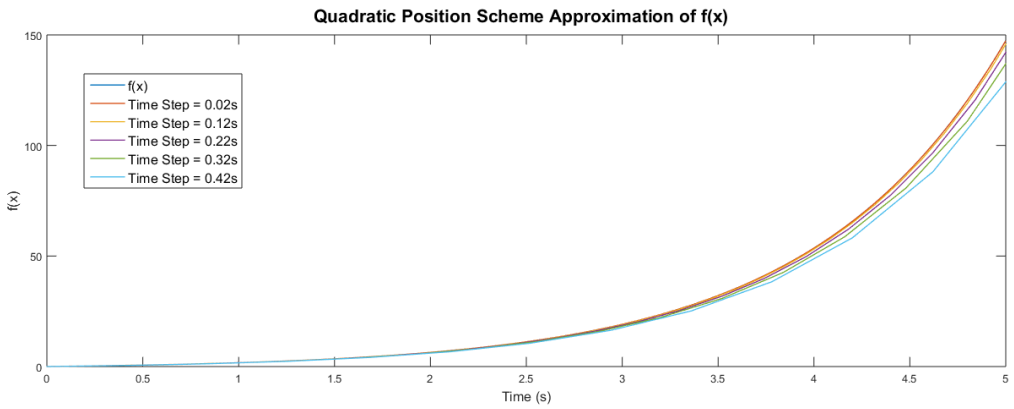

Figure 35: Quadratic position scheme used to approximate $f(x) = e^x - 1$ for 5 different time steps

From figure 35, it is apparent that the quadratic position scheme converges upon f(x), further, in contrast to figure 34 the convergence is seen to happen at a quicker rate than Eulers method. The convergence exhibited can be seen by the closely spaced lines of the approximation relative to Eulers method. The lines representing the approximations are so close distinguishing between them becomes difficult. The errors for the solution of $f(x)$ at $x = 5$ is shown in table 10.

Time Step (s)	0.42	0.32	0.22	0.12	0.02
Percent Error %	9.8	3.4	1.9	2.8	0.04

Table 4: Error at $x = 5$ for the Quadratic Position scheme at 5 Time Steps

From these error values it is evident that the Quadratic Position scheme converges quicker than Eulers method, at $t = 5$ the error is 0.04% compared to %4.8 for Eulers method. Note the scheme appears to diverge between time steps of $t_{ts} = 0.22$ and $t_{ts} = 0.12$ as the error increases, however the error then decreases from $t_{ts} = 0.12$ to $t_{ts} = 0.02$

To obtain a similiar positional error as Eulers method at a time step of $t_{ts} = 0.002s$, the Qudratic Positional scheme could be used with a time step of $t_{ts} = 0.34$. The

Quadratic Velocity scheme therefore is a more ideal scheme to use for the simulation, provided the added computational overhead from the complexity of the scheme does not outweigh the advantage of the quicker convergence.

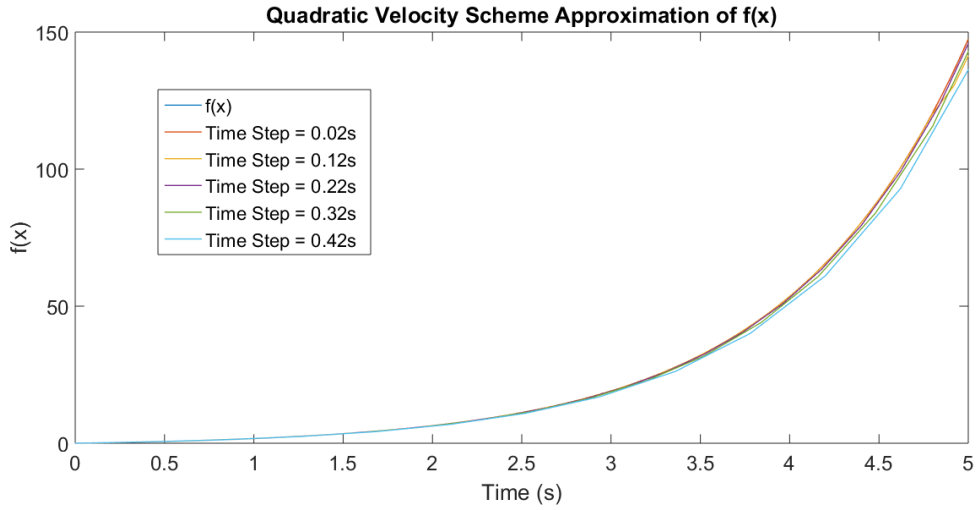

Figure 36: Quadratic Velocity scheme used to approximate $f(x) = e^x - 1$ for 5 different time steps

Figure 36 shows the Quadratic Velocity schemes approximation to the function, again for the same range and time steps. The lines representing different times steps are almost indistinguishable from each other, indicating yet a quicker rate of convergence than the Quadratic Position scheme. To quantitatively asses this the errors at $t = 5$ are tabulated in table 5. For the smallest time step, $t_{ts} = 0.02s$ the Quadratic Velocity scheme has an almost negligible error of around 0.02%. This is far lower than both the Quadratic Position Scheme and EUler's Method. Throughout the rest of the range of time steps the Quadratic Velocity scheme results in error values similar to the Quadratic Position scheme.

Time Step (s)	0.42	0.32	0.22	0.12	0.02
Percent Error %	7.6	3.1	1.1	4.2	0.002

Table 5: Error at $x = 5$ for the Quadratic Velocity scheme at 5 Time Steps

Figure 37 shows a comparison of the computational time of the scheme for a range of iterations. The schemes were tested up to 25 million iterations, this was found to refer to a computational time of Eulers method of around 10 seconds. The data shows

a very high degree of agreement for all 3 schemes with linear trends being apparent. Noise or other errors are apparent in some data points as they clearly deviate from the trend. However, even with this apparent the data is highly consistent with R^2 values of at least $R^2 = 0.99$ for all three data series.

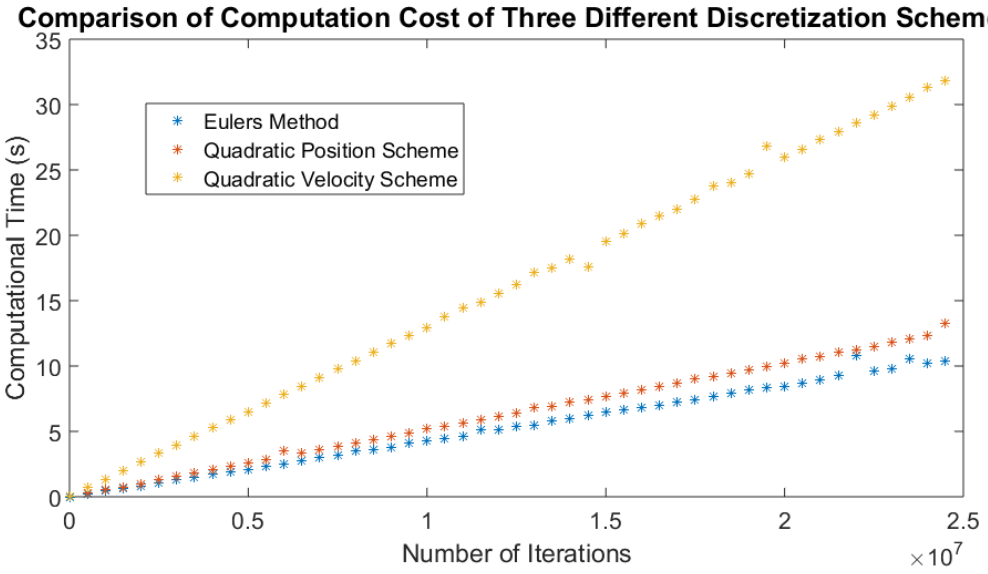

Figure 37: Comparison of the Computational cost of Euler's Method, Quadratic Position scheme and Quadratic Velocity Scheme

Linear trends were fitted to the three data sets using the least squares method (see appendix), and the slopes of these trends are shown in table 10. Comparison of the slopes reveals that the computational cost of the quadratic position scheme is around 20% more than Euler's methods ($5.1/4.3 = 1.19$) and the Quadratic Velocity scheme is around 3 times as expensive as Euler's methods ($13/4.3 = 3.02$). The computational cost of the schemes are of a magnitude of 10^-7 to 10^-6 seconds, the current time step of $t_{ts} = 0.02s$ is of magnitude 10^-2, hence the discretization scheme poses a near negligible overhead.

Scheme	Euler's Methods	Quadratic Position	Quadratic Velocity
Slope (x10^7)	4.3	5.1	13.0

Table 6: Gradients of line of best fit for computational cost of the three schemes

Whilst the scheme may present a near negligible overhead, its effect on accuracy is not trivial. Accuracy is seen to increase by orders of magnitude by use of the quadratic schemes over Eulers method. These increases in accuracy likewise represent increased computational overhead, however this is minor in comparison to the overhead of the

convective scheme. Hence, as a minimum the Quadratic Position scheme could be used for the final simulation, and possibly the more accurate Quadratic Velocity scheme.

11 Results & Discussion - Qualitative Analysis

Figure 38 shows three different view of the same wake predicted by a simple Horseshoe Vortex. It is a screen shot taken from the simulation running in Real-Time using 1024 elements arranged in a grid of 32x32 elements. The convective scheme used was the Dynamic Clustering scheme and the discretization method was the Quadratic Velocity scheme. The second to last row of elements on either side in the x-direction were given initial vorticity. Rows spanning the x-direction were spawned every $0.3s$. The spacing in the x-direction between consecutive elements was 0.5. The free-stream velocity of the flow was 0.5.

From figure 38 a number of features are notable. Firstly wingtip vortices are present and the vortex sheet can be seen to roll up tightly at either side. Secondly the central portion of the sheet is seen to move downward in the vertical direction (a consequence of lift generation) under the influence of the horseshoe vortex. This is the shape that is expected to results from a horseshoe vortex. However, bumpiness along the vortex sheet can be seen in the central section, this is an error likely introduced by the way the location of clusters are calculated and was outlined in the discussion of the fixed cluster size scheme. However unique to this example is the presence of a bump with two bumps of relatively lower magnitude. This is likely caused by the addition of extra clustering abstractions.

However, even with the presence of errors, the vortex sheet is well formed and visually looks correct. However this says nothing of the simulations accuracy relative to experimental data. With the promising results obtained from the simulation, though there is a definitely a case for pursuing further investigations into the accuracy of the simulation further.

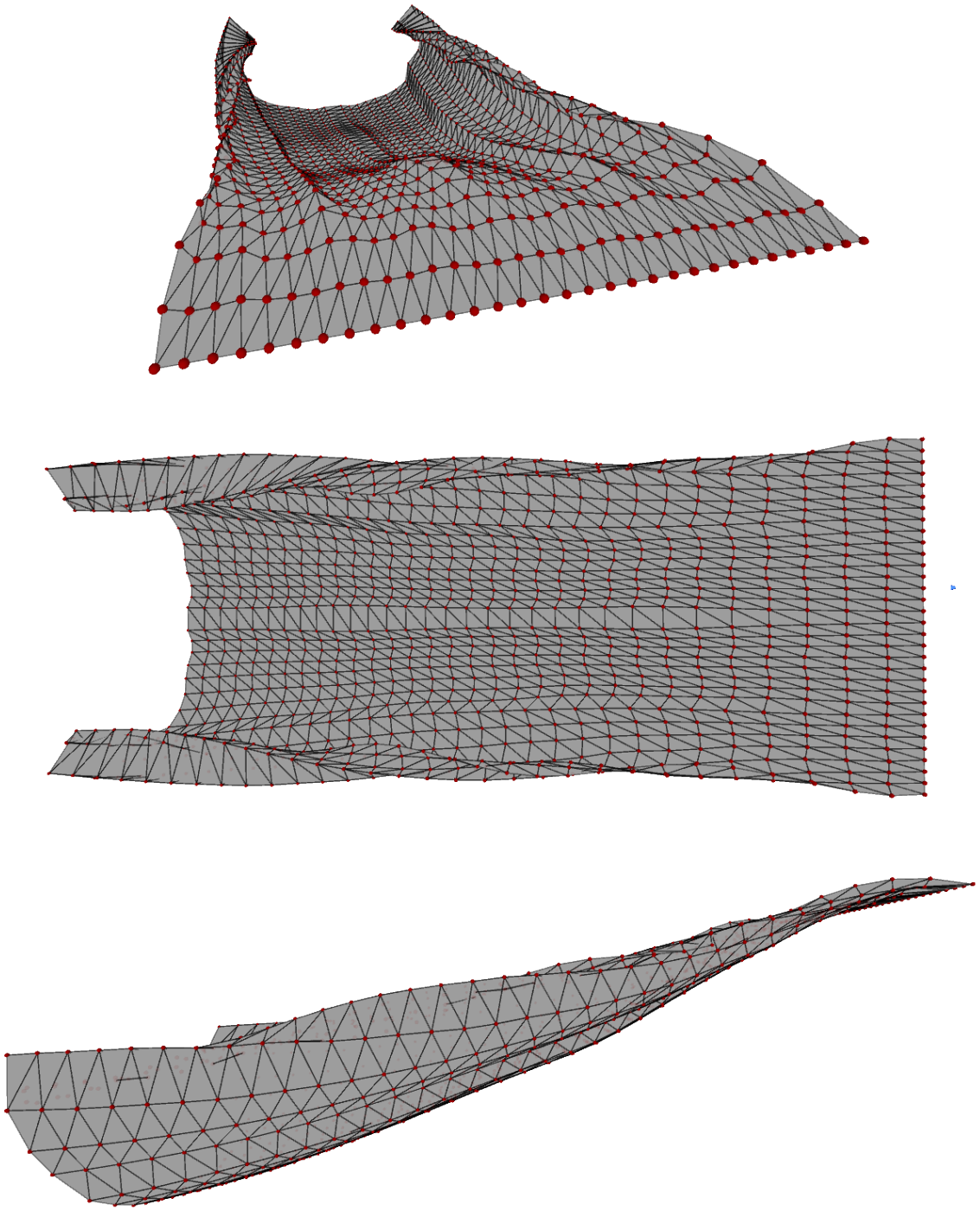

Figure 38: Screen Shot of the Discrete Vortex Method Developed Running at 60fps Using Dynamic Clustering and the Quadratic Velocity Scheme

12 Conclusions

12.1 General Conclusions

A series of investigations were performed into optimizations for the convective scheme and discretization scheme for a Discrete Vortex Method simulation. The findings from these investigations were then used for a qualitative analysis of the shape of vortex sheet produced by the given combination.

For the convective scheme, 3 different optimizations were investigated, a "biasing" scheme, a fixed size clustering scheme and a dynamic clustering scheme. The biasing scheme represented the simplest modification to the convective scheme, it was used as a benchmark, compared to the more complex schemes it was found to be ineffective for both radius and vorticity biasing. Fixed size clusters and dynamics scaling clusters both represented significant performance increases compared to biasing for the same level of accuracy. The dynamic cluster scaling scheme however produced less error than all fixed cluster sizes considered whilst performing calculations quicker, hence there was no advantage found for using fixed cluster sizes over dynamics clustering.

Two simplistic implicit differing order temporal discretization schemes were compared to Euler's Method. Both showed a significant increase in accuracy. The Quadratic Position scheme represented an almost negligible increase in overhead compared to Euler's method, whilst the Quadratic Velocity scheme represented a roughly threefold increase in computational overhead. For the number of iterations and lowest time step considered the accumulated truncation error for the Quadratic position scheme was two orders of magnitude lower than Euler's methods, whilst the Quadratic Velocity scheme was three orders of magnitude lower. Hence the quadratic position scheme as a minimum is likely a better alternative than Euler's method.

12.2 Specific Conclusions

12.2.1 Convective Scheme

Biasing was used as a benchmark as it is the simplest method of optimization both conceptually and in terms of implementation. Optimizations were achieved using both radius and vorticity based biasing. Vorticity based biasing yielded the largest optimization in terms of performance. However this was likely down the the initial conditions used during the simulation and such results would not be seen if more

complex initial conditions were used. Further investigations should be performed to assess this. Radius based biasing represented an optimization, however the gain in performance was not as significant as the clustering schemes and the accuracy over the clustering schemes reduced. Hence biasing posed no advantage over clustering other than its simper implementation. Further radius biasing did not reduce the N-Body problem to a ON complexity due to the additional overhead of the evaluation.

Both clustering schemes, fixed and dynamic, were superior to biasing in terms of both performance and accuracy. For the fixed clustering scheme performance was seen to increase with cluster size, however the accuracy was seen to decrease with cluster size. The dynamic cluster scheme used a base cluster abstraction size of 2x2 and was more accurate than both fixed cluster sizes tested. This is accountable to the aforementioned trend noted in the fixed clusters where larger cluster sizes lead to less accuracy, however the dynamic clustering scheme used larger clusters further away for the less influential elements. The dynamic clustering scheme posed both better performance and accuracy than fixed cluster size, hence it was used for the simulation in the Qualitative Analysis.

The positional errors from the fixed cluster size scheme were concentrated mainly on the elements that lay on cluster boundaries. This was likely caused by the vorticity of the clusters being approximated to act at the average position of all the elements in the cluster. This effect was exhibited in the dynamic clustering scheme where ridges of varying magnitude, referring to differing cluster abstractions, were visible on the vortex sheet.

12.2.2 Discretization Scheme

Three discretization schemes were tested, Eulers Methods and the quadratic position and velocity schemes. The schemes were tested by their approximation to the function $f(x) = e^x - 1$ over the range $x = 0$ to 5, for the smallest time step $t = 0.02$ (referring to 60fps) the quadratic position scheme was two magnitudes of order more accurate than Eulers methods (4.8% compared to \$0.04) whilst the quadratic scheme was three order of magnitude lower (4.8% compared to 0.002%). The quadratic position scheme posed a roughly 15% increase in computational cost over Eulers method whilst the quadratic position scheme was around 270% increase in computational cost. However the computational times were all far lower than the convective scheme. Hence the Quadratic position scheme was found to always be a better alternative than Eulers

Methods, with the Quadratic velocity scheme likely being better if there is a large number of elements (so that the computational cost of the convective scheme become so large such that the cost of the discretization scheme is negligble).

12.3 Recommendations

The qualitative analysis section demonstrated that the use of the optimizations were feasible and produced visually correct looking results for a simple Horseshoe Vortex. However all investigations relating to accuracy were relative to the unoptimized case. Further study is required to determine whether results accurate relative to experimental data can be obtained with an element count capable of running in real-time.

All code developed during this project was single-threaded (ran on a single CPU core). However as discussed in the literature review, Real-Time CFD efforts are almost entirely implemented on parallel architectures. The dynamic clustering scheme was programmed taking this into consideration. To implement a multi threaded simulation the Convect() function can be called from independent cores. Hence multi threading is easily realised by splitting the Convect() calls between multiple cores. This is easily achieved in C# by defining a "thread" method type. Hence no new infrastructure is required. If investigations reveal an element count capable of simulation realistic results cannot be obtained using a single threaded code the addition of multi threading needs to be considered.

If a clustering scheme is to be implemented for further study the method by which cluster position is determined should be examined in further detail. Errors for both clustering schemes were concentrated on cluster boundaries, as a first order approach the weighted average approach discussed in the theory section of this report could be investigated.

Euler's method was found to be significantly inefficient compared to higher order scheme with an accuracy increase of around two orders of magnitude for a 15% increase in computational cost. Hence if the same accuracy is required then a higher order scheme can be used and the time step increased. If a larger time step is used the convective scheme is given more time to perform its calculations. Given more time to perform its calculations means more element may be used. Hence Further investigation is required to determine the optimum parameters in regard to time-step and element count should be conducted. If the time-step is larger than required for

the simulation to appear to run in real-time interpolations could be used. Both the the quadratic position and velocity scheme mapped polynomials to position, these polynomial could be used for multiple frames whilst the convective scheme calculates the new velocity field.

13 Time Managment

Semester 1

Activity	Week Number											
	1	2	3	4	5	6	7	8	9	10	11	12
Familiarization of Unity												
Background Reading On Vortex Methods												
Investigation into Current Method												
Experiments conducted on existing method												
Development of Independent Testing Environment												
Development of New Convection Schemes												
Comparison on Initial and New Convection Scheme												
Writing up Report												

Semester 2

Activity	Week Number									
	1	2	3	4	5	6	7	8	9	10
Report Write Up										
Develop Biasing Scheme										
Develop Fixed Cluster Scheme										
Develop Dynamic Clustering										
Perform Experiments										
Combine Optimal Parameters										

Bibliography

Baden, Scott B and Elbridge Gerry Puckett (1990). "A fast vortex method for computing 2D viscous flow". In: *Journal of Computational Physics* 91.2, pp. 278–297. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1016/0021-9991(90)90038-3. URL: http://www.sciencedirect.com/science/article/pii/0021999190900383.

Barba, L (1996). *Computing high-Reynolds number vortical flows: A highly accurate method with a fully meshless formulation*. DOI: 10.1016/b978-044482322-9/50039-6.

Barnes, Josh and Piet Hut (1986). *A hierarchical O(N log N) force-calculation algorithm*. DOI: 10.1038/324446a0.

Boeing, Adrian and Thomas Braunl (2007). *Evaluation of real-time physics simulation systems*. DOI: 10.1145/1321261.1321312.

Brandvik, Tobias and Graham Pullan (2008). *Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware*. DOI: 10.2514/6.2008-607.

Computational Fluid Dynamics (CFD) - GPU Applications. URL: http://www.nvidia.com/object/computational_fluid_dynamics.html.

Cottet, Georges-Henri and Petros D. Koumoutsakos (2008). *Vortex methods: theory and practice*. Cambridge University Press.

Eldredge, Jeff D., Tim Colonius, and Anthony Leonard (2002). "A Vortex Particle Method for Two-Dimensional Compressible Flow". In: *Journal of Computational Physics* 179.2, pp. 371–399. ISSN: 0021-9991. DOI: http://dx.doi.org/10.1006/jcph.2002.7060. URL: http://www.sciencedirect.com/science/article/pii/S0021999102970609.

Engineering simulation: past, present and future (2017). URL: https://www.scientific-computing.com/feature/engineering-simulation-past-present-and-future.

Hanna, Keith and John Parry. "NAFEMS World Congress". In: *Back to the Future: Trends in Commercial CFD*. Mentor Graphics Corporation, Mechanical Analysis Division. URL: http://s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_90428.pdf.

Harlow, Francis H (2003). *Fluid dynamics in Group T-3 Los Alamos National Laboratory q (LA-UR-03-3852)*.

Johnson, T. A. and V. C. Patel (1999). *Flow past a sphere up to a Reynolds number of 300*. DOI: 10.1017/s0022112098003206.

Khan, M. Amirul Islam et al. (2015). *Real-time flow simulation of indoor environments using lattice Boltzmann method*. DOI: 10.1007/s12273-015-0232-9.

Lakkis, Issam and Ahmed F. Ghoniem (2003). "Axisymmetric vortex method for low-Mach number, diffusion-controlled combustion". In: *Journal of Computational Physics* 184.2, pp. 435–475. ISSN: 0021-9991. DOI: https://doi.org/10.1016/S0021-9991(02)00030-X. URL: http://www.sciencedirect.com/science/article/pii/S002199910200030X.

Launder, B.e. and D.b. Spalding (1974). *The Numerical Computation Of Turbulent Flows*. DOI: 10.1016/b978-0-08-030937-8.50016-7.

Li, Shaofan and Wing Kam Liu (2002). *Meshfree and particle methods and their applications*. DOI: 10.1115/1.1431547.

Liu, Chung Ho (2001). *A three-dimensional vortex particle-in-cell method for vortex motions in the vicinity of a wall*. DOI: 10.1002/fld.180.

Mohamad, A. A. (2011). *Lattice Boltzmann Method: Fundamentals and Engineering Applications with Computer Codes*.

NTSB (1973). *AIRCRAFT ACCIDENT REPORT DELTA AIR LINES, INC. MCDONNELL DOUGLAS DC-9-14, N3305L GREATER SOUTHWEST INTERNATIONAL AIRPORT FORT WORTH, TEXAS MAY 30, 1972*. URL: https://www.ntsb.gov/investigations/AccidentReports/Reports/AAR7315.pdf.

O'Connor, Joseph et al. (2016). "Computational fluid dynamics in the microcirculation and microfluidics: What role can the lattice Boltzmann method play?" In: *Integrative Biology* 8.5, pp. 589–602. ISSN: 1757-9694. DOI: 10.1039/c6ib00009f.

Patankar, S.v and D.b Spalding (1972). *A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows*. DOI: 10.1016/0017-9310(72)90054-3.

Rolfe, J. M. and K. J. Staples (2010). *Flight simulation*. Cambridge University Press.

Rosenhead, L. (1931). *The Formation of Vortices from a Surface of Discontinuity*. DOI: 10.1098/rspa.1931.0189.

Stewart, D E and J C Trinkle (1996). *An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and Coulomb friction*. DOI: 10.1002/(SICI)1097-0207(19960815)39:15<2673::AID-NME972>3.0.CO;2-I.

Warsi, Z. U. A. (2006). *Fluid dynamics: theoretical and computational approaches*.

Xu, Cheng (1999). *A vortex method for separated flow around an airfoil with a detached spoiler*. DOI: 10.1007/s004660050408.

14 Appendix: Simple Convection Script

```
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;
using UnityEngine;

public class SimpleConvection : MonoBehaviour {

    //variables for simulation
    private int xSize;
    private int ySize;
    private float gridSpacing;
    private Vector3[,] elementPosition;
    private Vector3[,] elementVelocity;
    private Vector2[,] elementVorticity;
    public Vector3[,] vorT;
    public Vector3 freeStreamVelocity = new Vector3(0.1f, 0.0f, 0.0f);
    //private Vector2[,] elementVorticityVector;

    //For testing and data collection purposes
    private bool testFlipFLop;
    private bool biasFlipFlop;
    private float[] timesTaken;
    private float initialTime;
    private float endTime;
    private float vorticityBias;
    private float radiusBias;
    private float[] biasTimes;
    private float[] biasParameters;
    private int iterationCount;
    private bool runFlipFlop;

        // Use this for initialization
        void Start () {

        //define grid size and spacing
        xSize = 16;
        ySize = 20;
        gridSpacing = 0.5f;
        elementPosition = new Vector3[255,255];
        elementVelocity = new Vector3[255, 255];
        elementVorticity = new Vector2[255, 255];
        vorT = new Vector3[255, 255];

        //for testing and data collection purposes
        testFlipFLop = false;
        biasFlipFlop = false;
        runFlipFlop = true;
        timesTaken = new float[255];
        biasTimes = new float[255];
        biasParameters = new float[255];
        iterationCount = 0;


        setInitialConditions();

        savePositionCSV();

    }

```

```
61          // Update is called once per frame
62          void Update () {
63
64          }
65
66      //physics sensitive things
67      private void FixedUpdate()
68      {
69
70
71          //testing routing
72          if (testFlipFLop == true)
73          {
74
75              for (int gridSize = 1; gridSize < 21; gridSize++)
76              {
77
78                  //Counter to show user progress
79
80                  //Make sure initial conditions are kept constant
81                  xSize = 15;
82                  ySize = gridSize;
83                  setInitialConditions();
84
85                  //take initial time
86                  initialTime = Time.realtimeSinceStartup;
87
88                  //this for loop replicates the fixed update loop, but doesnt require
                     ↪  rendering
89                  for (int loopcount = 0; loopcount < 500; loopcount++)
90                  {
91
92                      updatePositions();
93                      findVorticities();
94                      convectElements();
95
96                  }
97
98                  //take initial time
99                  endTime = Time.realtimeSinceStartup;
100
101                 //reporttime
102                 timesTaken[gridSize] = endTime - initialTime;
103                 saveTimeCSV();
104                 //savePositionCSV();
105
106             }
107             testFlipFLop = false;
108
109         }
110
111         //biasing factors test
112         if (biasFlipFlop == true)
113         {
114
115
116             for (float biasFactor = 0; biasFactor < 0.6; biasFactor = biasFactor +
                 ↪  0.03f, iterationCount++)
117             {
118
119                 //Make sure initial conditions are kept constant
120                 xSize = 15;
121                 ySize = 20;
```

```
122                 setInitialConditions();
123 
124                 //take initial time
125                 initialTime = Time.realtimeSinceStartup;
126 
127                 //set bias
128                 radiusBias = biasFactor;
129 
130                 //this for loop replicates the fixed update loop, but doesnt require
                        ↪  rendering
131                 for (int loopcount = 0; loopcount < 500; loopcount++)
132                 {
133 
134                     updatePositions();
135                     findVorticities();
136                     convectElements();
137 
138                 }
139 
140 
141             //take initial time
142             endTime = Time.realtimeSinceStartup;
143 
144             //record values
145             biasTimes[iterationCount] = (endTime - initialTime);
146             biasParameters[iterationCount] = biasFactor;
147                 //saveBiasPositionCSV(biasFactor);
148                 //savePositionCSV();
149 
150 
151             }
152 
153             //savePositionCSV();
154             saveBiasCSV();
155             Debug.Log("Done!");
156 
157             //reset so it doesnt run in the loop
158             biasFlipFlop = false;
159         }
160 
161         //this is just for normal running
162         if (runFlipFlop == true)
163         {
164 
165             for (int iterations = 0; iterations < 500; iterations++)
166             {
167                 updatePositions();
168                 findVorticities();
169                 convectElements();
170             }
171             savePositionCSV();
172             runFlipFlop = false;
173         }
174 
175 
176     }
177 
178     //function to set initial conditions
179     void setInitialConditions()
180     {
181         //set initial positiions and velocities
182         for (int x = 0; x < xSize; x++)
183         {
```

```
184             for (int y = 0; y < ySize; y++)
185             {
186                 elementPosition[x, y] = new Vector3(x * gridSpacing, 0, y *
                    ↪  gridSpacing);
187                 elementVelocity[x, y] = new Vector3(0.0f, 0.0f, 0.0f);
188                 elementVorticity[x, y] = new Vector2(0.0f, 0.0f);
189 
190                 if (x == 1)
191                 {
192                     elementVorticity[x, y] = new Vector2(0.0f, -0.5f);
193 
194                 }
195 
196                 if (x == xSize - 2)
197                 {
198                     elementVorticity[x, y] = new Vector2(0.0f, 0.5f);
199                 }
200             }
201         }
202     }
203 
204     //update positions
205     void updatePositions()
206     {
207 
208         //for loops to cycle through grid
209         for (int x = 0; x < xSize; x++)
210         {
211             for (int y = 0; y< ySize; y++)
212             {
213                 elementPosition[x, y] = elementPosition[x, y] + 0.02f *
                    ↪  (elementVelocity[x, y] + freeStreamVelocity);
214             }
215         }
216 
217     }
218 
219     //This function is written awfuly but it works, if you're reading this, redo this!
220     //function to find vorticities
221     void findVorticities()
222     {
223 
224         //find initial vorticity vectors
225         for (int x = 1; x < xSize - 1; x++)
226         {
227             for (int y = 1; y < ySize - 1; y++)
228             {
229 
230                 //find vorticity in x and y
231                 Vector3 vorX = elementVorticity[x, y].x * (elementPosition[x + 1, y] -
                    ↪  elementPosition[x - 1, y]).normalized;
232                 Vector3 vorY = elementVorticity[x, y].y * (elementPosition[x, y + 1] -
                    ↪  elementPosition[x, y - 1]).normalized;
233                 vorT[x, y] = vorX + vorY;
234 
235             }
236         }
237 
238     }
239 
240     //This function is written awfuly but it works, if you're reading this, redo this!
241     //function to find vorticities
242     void findVorticities2()
```

```
243     {
244
245         //find initial vorticity vectors
246         for (int x = 1; x < xSize - 2; x++)
247         {
248             for (int y = 1; y < ySize - 2; y++)
249             {
250
251                 //find vorticity in x and y
252                 Vector3 vorX = elementVorticity[x, y].x * (elementPosition[x + 1, y] -
                     ↪   elementPosition[x - 1, y]).normalized;
253                 Vector3 vorY = elementVorticity[x, y].y * (elementPosition[x, y + 1] -
                     ↪   elementPosition[x, y - 1]).normalized;
254                 vorT[x, y] = vorX + vorY;
255
256             }
257         }
258
259         //determine vorticities at grid extremes
260         //At 0,0
261         Vector3 vortX = elementVorticity[0, 0].x * (elementPosition[1, 0] -
             ↪   elementPosition[0, 0]).normalized;
262         Vector3 vortY = elementVorticity[0, 0].y * (elementPosition[0,0] -
             ↪   elementPosition[0, 1]).normalized;
263         vorT[0, 0] = vortX + vortY;
264         //at xMax,0
265         vortX = elementVorticity[xSize - 1, 0].x * (elementPosition[xSize - 1, 0] -
             ↪   elementPosition[xSize - 2, 0]).normalized;
266         vortY = elementVorticity[xSize - 1, 0].y * (elementPosition[xSize - 1, 0] -
             ↪   elementPosition[xSize - 1, 1]).normalized;
267         vorT[xSize-1, 0] = vortX + vortY;
268         //at 0,yMax
269         vortX = elementVorticity[0, ySize-1].x * (elementPosition[1, ySize - 1] -
             ↪   elementPosition[0, ySize - 1]).normalized;
270         vortY = elementVorticity[0, ySize - 1].y * (elementPosition[0, ySize - 2] -
             ↪   elementPosition[0, ySize - 1]).normalized;
271         vorT[0, ySize - 1] = vortX + vortY;
272         //at xMax,yMax
273         vortX = elementVorticity[xSize - 1, ySize - 1].x * (elementPosition[xSize - 2,
             ↪   ySize - 1] - elementPosition[xSize - 1, ySize - 1]).normalized;
274         vortY = elementVorticity[xSize - 1, ySize - 1].y * (elementPosition[xSize - 1,
             ↪   ySize - 2] - elementPosition[xSize - 1, ySize - 1]).normalized;
275         vorT[xSize - 1, ySize - 1] = vortX + vortY;
276
277         //now lets figure them out for the rows along the sides! (this is very
             ↪   inefficient but time constraints)
278         //first the row at x=0
279         for (int y = 1; y< ySize -2; y++)
280         {
281             vortX = elementVorticity[0, y].x * (elementPosition[1, y] -
                 ↪   elementPosition[0, y]).normalized;
282             vortY = elementVorticity[0, y].y * (elementPosition[0, y] -
                 ↪   elementPosition[0, y + 1]).normalized;
283             vorT[0, 0] = vortX + vortY;
284         }
285         //and the row at y=0
286         for (int x = 1; x < xSize - 2; x++)
287         {
288
289         }
290
291     }
292
```

```
293     //function to convect all elements
294     void convectElements()
295     {
296
297
298
299         //for loop to cycle through all elements
300         for (int xN = 0; xN < xSize; xN++)
301         {
302             for (int yN = 0; yN < ySize; yN++)
303             {
304                 //Reset the velocity field
305                 elementVelocity[xN, yN] = Vector3.zero;
306
307                 //Now that every element is going to be cycled through, the elements
                     ↪  need to be cycled through again
308                 for (int xC = 0; xC < xSize; xC++)
309                 {
310                     for (int yC = 0; yC < ySize; yC++)
311                     {
312
313                         if (xN != xC && yN != yC)
314                         {
315
316
317
318                             if (elementVorticity[xC,yC].magnitude >= radiusBias)
319                             {
320
321                                 //Calcuate Influence
322                                 Vector3 r = elementPosition[xN, yN] -
                                     ↪  elementPosition[xC, yC];
323                                 elementVelocity[xN, yN] = elementVelocity[xN, yN] + (1f
                                     ↪  / (4f * Mathf.PI)) * (Vector3.Cross(vorT[xC, yC],
                                     ↪  r) / (Mathf.Pow(r.magnitude, 2)));
324
325                             }
326                         }
327                     }
328                 }
329
330             }
331         }
332
333     }
334
335     //function to convect all elements
336     void convectClusteredElements()
337     {
338
339
340
341         //for loop to cycle through all elements
342         for (int xN = 0; xN < xSize; xN++)
343         {
344             for (int yN = 0; yN < ySize; yN++)
345             {
346                 //Reset the velocity field
347                 elementVelocity[xN, yN] = Vector3.zero;
348
349                 //Now that every element is going to be cycled through, the elements
                     ↪  need to be cycled through again
350                 for (int xC = 1; xC < xSize - 1; xC++)
```

```
351                 {
352                     for (int yC = 1; yC < ySize - 1; yC++)
353                     {
354 
355                         if (xN != xC && yN != yC)
356                         {
357 
358 
359 
360                             if (elementVorticity[xC, yC].magnitude >= radiusBias)
361                             {
362 
363                                 //Calcuate Influence
364                                 Vector3 r = elementPosition[xN, yN] -
                                     ↪  elementPosition[xC, yC];
365                                 elementVelocity[xN, yN] = elementVelocity[xN, yN] + (1f
                                     ↪  / (4f * Mathf.PI)) * (Vector3.Cross(vorT[xC, yC],
                                     ↪  r) / (Mathf.Pow(r.magnitude, 2)));
366 
367                             }
368                         }
369                     }
370                 }
371 
372             }
373         }
374 
375     }
376 
377     //Function to write positions to  csv file
378     void savePositionCSV()
379     {
380 
381 
382         //file to write data into
383         using (StreamWriter writetext = new StreamWriter("Output.csv"))
384         {
385 
386             //Write Initial headings
387             writetext.WriteLine("X Grid Position, Y  Grid Position, VecX, VecY, VecZ");
388 
389             for (int x = 0; x < xSize; x++)
390             {
391                 for (int y = 0; y < ySize; y++)
392                 {
393                     writetext.WriteLine(x + "," + y + "," + elementPosition[x,y].x +
                         ↪  "," + elementPosition[x, y].y + "," + elementPosition[x,
                         ↪  y].z);
394                 }
395             }
396         }
397 
398 
399     }
400 
401     //Function to write positions to  csv file
402     void saveBiasPositionCSV(float BiasFactor)
403     {
404 
405 
406         //file to write data into
407         using (StreamWriter writetext = new
             ↪  StreamWriter("Positions"+BiasFactor+".csv"))
```

```
408         {
409
410             //Write Initial headings
411             writetext.WriteLine("X Grid Position, Y  Grid Position, VecX, VecY, VecZ");
412
413             for (int x = 0; x < xSize; x++)
414             {
415                 for (int y = 0; y < ySize; y++)
416                 {
417                     writetext.WriteLine(x + "," + y + "," + elementPosition[x, y].x +
                        ↪  "," + elementPosition[x, y].y + "," + elementPosition[x,
                        ↪  y].z);
418                 }
419             }
420         }
421
422
423     }
424
425     //Function to write a csv file containing the times taken
426     void saveTimeCSV()
427     {
428
429         using (StreamWriter writetext = new StreamWriter("TimesTaken.csv"))
430         {
431
432             //Write Initial headings
433             writetext.WriteLine("Grid Size Y"+","+"Computational Time");
434
435             //cycle through the array and write the data value
436             for (int val = 1; val < timesTaken.Length; val++)
437             {
438                     writetext.WriteLine(val+","+timesTaken[val]);
439             }
440         }
441
442     }
443
444     //Function to write a csv file containing the times taken
445     void saveBiasCSV()
446     {
447
448         using (StreamWriter writetext = new StreamWriter("BiasTimesTaken.csv"))
449         {
450
451             //Write Initial headings
452             writetext.WriteLine("Bias Factor" + "," + "Computational Time");
453
454             //cycle through the array and write the data value
455             for (int val = 0; val < biasTimes.Length; val++)
456             {
457                 writetext.WriteLine(biasParameters[val] + "," + biasTimes[val]);
458             }
459         }
460
461     }
462
463     //Draw the elements so they're visible
464     private void OnDrawGizmos() // this shows control points as red spheres in the edit
        ↪  window
465     {
466
467         //cycle through all elements
```

```
468             for (int x = 0; x < xSize; x++)
469             {
470                 for (int y = 0; y < ySize; y++)
471                 {
472 
473                     //draw the gizmo
474                     Gizmos.DrawSphere(elementPosition[x, y], 0.1f);
475                 }
476             }
477 
478         }
479 
480     }
```

15 Appendix: Fixed Cluster Convection Script

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class convectionHandler : MonoBehaviour {

    //Variables for simulation purpose
    private int xSize;
    private int ySize;
    private float gridSpacing;
    private Vector3[,] elementPosition;
    private Vector3[,] elementVelocity;
    private Vector2[,] elementVorticity;
    private int clusterSize;
    private Vector3[,] vorT;

    //cluster related
    public Vector3[,] clusterPosition;
    private Vector3[,] clusterVorticity;

    //for recording/analysis purposes
    private float[] timesTaken;
    private float[] gridSizeUsed;

    // Use this for initialization
    void Start () {

        //Initialize Arrays
        elementPosition = new Vector3[255, 255];
        elementVelocity = new Vector3[255, 255];
        elementVorticity = new Vector2[255, 255];
        vorT = new Vector3[255, 255];
        clusterSize = 3;

        //for clusters
        clusterPosition = new Vector3[255, 255];
        clusterVorticity = new Vector3[255, 255];

        //specify testing variables
        xSize = 15;
        ySize = 21;
        gridSpacing = 0.5f;
        setInitialConditions();
        timesTaken = new float[255];
        gridSizeUsed = new float[255];
        setInitialConditions();


    }

        // Update is called once per frame
        void Update () {

        }

    //Physics related math
    void FixedUpdate()
    {

```

```
61          //Uncomment these to just run the simulation in real time
62          //findVorticities();
63          //calculateCoefficients();
64          //ClusteredConvection();
65          //updatePositions();
66
67          //Just a button press to activate testing
68          if (Input.GetKeyDown("space"))
69          {
70              testComputationalTime(2);
71
72          }
73
74      }
75
76      void testComputationalTime(int mode)
77      {
78
79          if (mode == 1)
80          {
81
82
83          //Index counter is
84          int indexCounter = 0;
85
86          //first for loop sets the cluster size
87          for (int gridSize = 5; gridSize < 26; gridSize += 5, indexCounter++)
88          {
89
90              //set initial conditions
91              xSize = 15;
92              ySize = gridSize;
93              setInitialConditions();
94
95              //take initial time
96              float initialTime = Time.realtimeSinceStartup;
97
98              //now different clustersizes are cycled through, the iterations need to be
                    ↪  done
99              for (int iterations = 0; iterations < 500; iterations++)
100             {
101                 findVorticities();
102                 calculateCoefficients();
103                 ClusteredConvection();
104                 updatePositions();
105             }
106
107             //take end time
108             float endTime = Time.realtimeSinceStartup;
109
110             //record values for writing later
111             timesTaken[indexCounter] = endTime - initialTime;
112             gridSizeUsed[indexCounter] = gridSize;
113         }
114
115         //Now we need to write the data to an excel files
116         saveTimeCSV();
117
118
119         }
120
121         //Second mode is to find errors
122         if (mode == 2)
```

```
123         {
124
125             //Run for the set ammount of iteration
126             for (int iterations = 0; iterations < 500; iterations++)
127             {
128                 findVorticities();
129                 calculateCoefficients();
130                 ClusteredConvection();
131                 updatePositions();
132             }
133
134             //Now write the results
135             savePositionCSV();
136         }
137
138     }
139
140     //Function to write positions to  csv file
141     void savePositionCSV()
142     {
143
144
145         //file to write data into
146         using (StreamWriter writetext = new StreamWriter("Output.csv"))
147         {
148
149             //Write Initial headings
150             writetext.WriteLine("X Grid Position, Y  Grid Position, VecX, VecY, VecZ");
151
152             for (int x = 0; x < xSize; x++)
153             {
154                 for (int y = 0; y < ySize; y++)
155                 {
156                     writetext.WriteLine(x + "," + y + "," + elementPosition[x, y].x +
                        ↪  "," + elementPosition[x, y].y + "," + elementPosition[x,
                        ↪  y].z);
157                 }
158             }
159         }
160
161
162     }
163
164     //Function to write a csv file containing the times taken
165     void saveTimeCSV()
166     {
167
168         using (StreamWriter writetext = new StreamWriter("TimesTaken.csv"))
169         {
170
171             //Write Initial headings
172             writetext.WriteLine("Grid Size Used" + "," + "Computational Time");
173
174             //cycle through the array and write the data value
175             for (int val = 0; val < timesTaken.Length; val++)
176             {
177                 writetext.WriteLine(gridSizeUsed[val] + "," + timesTaken[val]);
178             }
179         }
180
181     }
182
183     //This function calculates the coefficients for the clusters
```

```
184     void calculateCoefficients()
185     {
186
187         //Reinitialize arrays
188         clusterVorticity = new Vector3[255, 255];
189         clusterPosition = new Vector3[255, 255];
190
191         //For loop to cycle through cluster positions
192         for (int xClust = 0; xClust < (xSize/clusterSize); xClust++)
193         {
194             for (int yClust = 0; yClust < (ySize/clusterSize); yClust++)
195             {
196
197                 //Now each cluster is cycled through, every element in the cluster
                    ↪  needs to be cycled through
198                 for (int xIn = 0; xIn < clusterSize; xIn++)
199                 {
200                     for (int yIn = 0; yIn < clusterSize; yIn++)
201                     {
202
203                         // Now we can add up their vorticity
204                         clusterVorticity[xClust, yClust] += vorT[(xClust*clusterSize) +
                            ↪  xIn, (yClust*clusterSize) + yIn];
205                         clusterPosition[xClust, yClust] +=
                            ↪  elementPosition[(xClust*clusterSize) + xIn, (yClust *
                            ↪  clusterSize) + yIn] * (1.0f/Mathf.Pow(clusterSize,2));
206
207                     }
208                 }
209
210             }
211         }
212     }
213
214     void calculateClusterCoefficients()
215     {
216
217         //First set the values back to zero
218         for (int x = 0; x < (xSize / clusterSize); x++)
219         {
220             for (int y = 0; y < (ySize/clusterSize); y++)
221             {
222                 clusterVorticity[x, y] = new Vector3(0.0f, 0.0f, 0.0f);
223                 clusterPosition[x, y] = new Vector3(0.0f, 0.0f, 0.0f);
224             }
225         }
226
227         //Now we cycle through the clusters
228         for (int xClusterPos = 0; xClusterPos < xSize; xClusterPos += clusterSize)
229         {
230             for (int yClusterPos = 0; yClusterPos < xSize; yClusterPos += clusterSize)
231             {
232
233                 //Now we need to cycle through all the elements
234                 for (int xInCluster = 0; xInCluster < clusterSize; xInCluster++)
235                 {
236                     for (int yInCluster = 0; yInCluster < clusterSize; yInCluster++)
237                     {
238
239                         //Now all elements are cycled through, we need to add their
                            ↪  values to the cluster
```

```
240                         clusterVorticity[(xClusterPos / clusterSize), (yClusterPos /
                             ↪  clusterSize)] += vorT[xClusterPos + xInCluster,
                             ↪  yClusterPos + yInCluster];
241                         clusterPosition[(xClusterPos / clusterSize), (yClusterPos /
                             ↪  clusterSize)] += elementPosition[xClusterPos + xInCluster,
                             ↪  yClusterPos + yInCluster];
242                     }
243                 }
244             }
245         }
246
247         //Now take the average position
248         for (int xCluster = 0; xCluster < xSize/clusterSize; xCluster++)
249         {
250             for (int yCluster = 0; yCluster < ySize/clusterSize; yCluster++)
251             {
252                 clusterPosition[xCluster, yCluster] = clusterPosition[xCluster,
                     ↪  yCluster] * (1 / (clusterSize * clusterSize));
253             }
254         }
255
256
257     }
258
259     //update positions
260     void updatePositions()
261     {
262
263         //for loops to cycle through grid
264         for (int x = 0; x < xSize; x++)
265         {
266             for (int y = 0; y < ySize; y++)
267             {
268                 elementPosition[x, y] = elementPosition[x, y] + 0.02f *
                     ↪  elementVelocity[x, y];
269             }
270         }
271
272     }
273
274     //function to find vorticities
275     void findVorticities()
276     {
277
278         //find initial vorticity vectors
279         for (int x = 1; x < xSize - 1; x++)
280         {
281             for (int y = 1; y < ySize - 1; y++)
282             {
283
284                 //find vorticity in x and y
285                 Vector3 vorX = elementVorticity[x, y].x * (elementPosition[x + 1, y] -
                     ↪  elementPosition[x - 1, y]).normalized;
286                 Vector3 vorY = elementVorticity[x, y].y * (elementPosition[x, y + 1] -
                     ↪  elementPosition[x, y - 1]).normalized;
287                 vorT[x, y] = vorX + vorY;
288
289             }
290         }
291
292     }
293
294     //Function to convect elements
```

```
295     private void convectElements()
296     {
297
298         //First set of for loops cycle through the clusters
299         for (int xClust = 0; xClust < (xSize/clusterSize); xClust++)
300         {
301             for (int yClust = 0; yClust < (ySize/clusterSize); yClust++)
302             {
303
304                 //Now every blob in the clsuter needs to be cycled through
305                 for (int xInC = 0; xInC < clusterSize; xInC++)
306                 {
307                     for (int yInC = 0; yInC < clusterSize; yInC++)
308                     {
309
310
311                         //this for loop series convects the given element with the
                             ↪  elements in its cluster
312                         for (int xConv = 0; xConv < clusterSize; xConv++)
313                         {
314                             for (int yConv = 0; yConv < clusterSize; yConv++)
315                             {
316
317                                 //Check that the convection element isnt the element to
                                     ↪  be convected
318                                 if ( xInC != xConv && yInC != yConv)
319                                 {
320                                     //Debug.Log("x: "+(xClust * clusterSize + xInC)+"
                                         ↪  Y: "+(yClust * clusterSize + yInC));
321                                     //Convect the element
322                                     Vector3 r = elementPosition[xClust * clusterSize +
                                         ↪  xInC, yClust * clusterSize + yInC] -
                                         ↪  elementPosition[xClust * clusterSize + xConv,
                                         ↪  yClust * clusterSize + yConv];
323
324
                                         ↪  elementVelocity[xClust*clusterSize+xInC,yClust*clusterSize+yInC
                                         ↪  += (1f / (4f * Mathf.PI)) *
                                         ↪  (Vector3.Cross(vorT[xClust * clusterSize +
                                         ↪  xConv, yClust * clusterSize + yConv], r) /
                                         ↪  (Mathf.Pow(r.magnitude, 2)));
325
326                                     //elementVelocity[xN, yN] = elementVelocity[xN, yN]
                                         ↪  + (1f / (4f * Mathf.PI)) *
                                         ↪  (Vector3.Cross(vorT[xC, yC], r) /
                                         ↪  (Mathf.Pow(r.magnitude, 2)));
327
328                                 }
329
330                             }
331                         }
332
333
334                         //This next for loop convects the element int he cluster with
                             ↪  other clsuters
335
336
337                     }
338                 }
339
340             }
341         }
342
```

```
343     }
344
345     void clusterConvectElements()
346     {
347
348     //First cycle through all cluster
349     for (int xClusterPos = 0; xClusterPos < xSize; xClusterPos += clusterSize)
350         {
351             for (int yClusterPos = 0; yClusterPos < ySize; yClusterPos += clusterSize)
352             {
353                 //Debug.Log("X :" + xClusterPos + "   Y: " + yClusterPos);
354
355                 //Now that all clusters are cycled through, the elements in them need
                     ↪  to be cycled
356                 for (int xInCluster = 0; xInCluster < clusterSize; xInCluster++)
357                 {
358                     for (int yInCluster = 0; yInCluster < clusterSize; yInCluster++)
359                     {
360                         //Debug.Log("X :" + (xClusterPos+xInCluster) + "   Y: " +
                             ↪  (yClusterPos+yInCluster));
361
362                         //The velocity of the element to be convected needs to be
                             ↪  zeroed
363                         elementVelocity[xClusterPos + xInCluster, yClusterPos +
                             ↪  yInCluster] = Vector3.zero;
364
365                         //Now that all elements are cycled through, the elements in
                             ↪  their respective cluster need to be cycled through
366                         for (int xConvecting = 0; xConvecting < clusterSize;
                             ↪  xConvecting++)
367                         {
368                             for (int yConvecting = 0; yConvecting < clusterSize;
                                 ↪  yConvecting++)
369                             {
370                                 //Debug.Log("X :" + (xClusterPos + xInCluster) + "   Y:
                                     ↪  " + (yClusterPos + yInCluster));
371
372                                 //Now we need to check that we're not convecting the
                                     ↪  element with itself
373                                 if (xInCluster != xConvecting && yInCluster !=
                                     ↪  yConvecting)
374                                 {
375
376                                     //Now we can implement the biot savart law
377                                     //First the radius need to be calculated
378                                     Vector3 radius = elementPosition[xClusterPos +
                                         ↪  xInCluster, yClusterPos + yInCluster] -
                                         ↪  elementPosition[xClusterPos + xConvecting,
                                         ↪  yClusterPos + yConvecting];
379                                     //Now we can implement the biot-savart law itself
380                                     elementVelocity[xClusterPos + xInCluster,
                                         ↪  yClusterPos + yInCluster] += (1.0f / (4.0f *
                                         ↪  Mathf.PI)) * (Vector3.Cross(vorT[xClusterPos +
                                         ↪  xConvecting, yClusterPos + yConvecting],
                                         ↪  radius) / (Mathf.Pow(radius.magnitude,2)));
381
382                                 }
383                             }
384                         }
385
386                     //Now we need to cycle through clusters
387                     for (int xClusterID = 0; xClusterID < (xSize/clusterSize);
                         ↪  xClusterID++)
```

```
388                         {
389                             for (int yClusterID = 0; yClusterID < (ySize/clusterSize);
                                ↪  yClusterID++)
390                             {
391 
392                                 //Check we're not going to consider our own cluster
393                                 if (xClusterID != (xClusterPos*clusterSize) &&
                                    ↪  yClusterID != (yClusterPos * clusterSize))
394                                 {
395 
396                                     elementVelocity[xClusterPos + xInCluster,
                                        ↪  yClusterPos + yInCluster] += new Vector3(0.0f,
                                        ↪  0.5f, 0.5f);
397                                 }
398                             }
399                         }
400 
401                     }
402                 }
403 
404             }
405         }
406 
407     }
408 
409     //Third attempt at this function!
410     void ClusteredConvection()
411     {
412 
413         //First Cycle through all clusters
414         for (int xC = 0; xC < (xSize/clusterSize); xC++)
415         {
416             for (int yC = 0; yC < (ySize/clusterSize); yC++)
417             {
418 
419                 //Now we need to consider every element in the clsuter
420                 for (int xI = 0; xI < clusterSize; xI++)
421                 {
422                     for (int yI = 0; yI < clusterSize; yI++)
423                     {
424 
425                         //We need to reset their velocities so they dont add up
426                         elementVelocity[(xC * clusterSize) + xI, (yC * clusterSize) +
                                ↪  yI] = Vector3.zero;
427 
428                         //Now we need to cycle through the individual elements in that
                                ↪  cluster
429                         for (int xIC = 0; xIC < clusterSize; xIC++)
430                         {
431                             for (int yIC = 0; yIC < clusterSize; yIC++)
432                             {
433 
434                                 //Now we need to check that we're not convecting the
                                    ↪  element with itself
435                                 if (xI != xIC && yI != yIC)
436                                 {
437 
438                                     //now we need to find the radius
439                                     Vector3 Radius = elementPosition[(xC * clusterSize)
                                        ↪  + xI, (yC * clusterSize) + yI] -
                                        ↪  elementPosition[(xC * clusterSize) + xIC, (yC
                                        ↪  * clusterSize) + yIC];
440                                     //Now we can implement the biot-savart law
```

```
441                                     elementVelocity[(xC * clusterSize) + xI, (yC *
                                         ↪  clusterSize) + yI] += (1.0f / (4.0f *
                                         ↪  Mathf.PI)) * (Vector3.Cross(vorT[(xC *
                                         ↪  clusterSize) + xIC, (yC * clusterSize) + yIC],
                                         ↪  Radius) / Mathf.Pow(Radius.magnitude, 2));
442
443                                 }
444                             }
445                         }
446
447                         //Now we need to convect the given element with the clusters
448                         //First we cycle through all the clusters
449                         for (int xCC = 0; xCC < (xSize/clusterSize); xCC++)
450                         {
451                             for (int yCC = 0; yCC < (ySize/gridSpacing); yCC++)
452                             {
453
454                                 //Now we need to check that we're not going to convect
                                     ↪  with the clsuter the element is inside of
455                                 if (xC != xCC && yC != yCC)
456                                 {
457
458                                     //Now we need to calculate the radius
459                                     Vector3 Radius = elementPosition[(xC * clusterSize)
                                         ↪  + xI, (yC * clusterSize) + yI] -
                                         ↪  clusterPosition[xCC, yCC];
460                                     //Now we can implement the biot-savart law
461                                     if (Radius.magnitude != 0)
462                                     {
463                                         elementVelocity[(xC * clusterSize) + xI, (yC *
                                             ↪  clusterSize) + yI] += (1.0f / (4.0f *
                                             ↪  Mathf.PI)) *
                                             ↪  (Vector3.Cross(clusterVorticity[xCC, yCC],
                                             ↪  Radius) / Mathf.Pow(Radius.magnitude, 2));
464                                     }
465
466
467
468                                 }
469                             }
470                         }
471                     }
472                 }
473
474             }
475         }
476
477     }
478
479     //function to set initial conditions
480     void setInitialConditions()
481     {
482         //set initial positiions and velocities
483         for (int x = 0; x < xSize; x++)
484         {
485             for (int y = 0; y < ySize; y++)
486             {
487                 elementPosition[x, y] = new Vector3(x * gridSpacing, 0, y *
                     ↪  gridSpacing);
488                 elementVelocity[x, y] = new Vector3(0.0f, 0.0f, 0.0f);
489                 elementVorticity[x, y] = new Vector2(0.0f, 0.0f);
490
491                 if (x == 1)
```

```
492                 {
493                     elementVorticity[x, y] = new Vector2(0.0f, -0.5f);
494 
495                 }
496 
497                 if (x == xSize - 2)
498                 {
499                     elementVorticity[x, y] = new Vector2(0.0f, 0.5f);
500                 }
501             }
502         }
503     }
504 
505     //Draw the elements so they're visible
506     private void OnDrawGizmos() // this shows control points as red spheres in the edit
        ↪  window
507     {
508 
509         //cycle through all elements
510         for (int x = 0; x < xSize; x++)
511         {
512             for (int y = 0; y < ySize; y++)
513             {
514 
515                 //draw the gizmo
516                 Gizmos.DrawSphere(elementPosition[x, y], 0.1f);
517             }
518         }
519 
520     }
521 }
```

16 Appendix: Dynamic Cluster Convection Script

```
using System.Collections;
using System.Linq;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class TestingScripts : MonoBehaviour {

    //Variables related to elements
    private Vector3[,,] elementPosition = new Vector3[255,255,9];
    private Vector3[,,] elementVorticity = new Vector3[255, 255, 9];
    private Vector3[,] elementVelocity = new Vector3[255, 255];
    private Vector3[,,] vorT = new Vector3[255, 255,9];
    private int xSize = 32;
    private int ySize = 32;
    private float gridSpacing = 0.5f;
    private Vector3 freeStreamVelocity = new Vector3(0.0f, 0.0f, 0.5f);

    //clustering scheme variables
    private int clusterSize = 2;
    private int maxDegree = 4;

    //data gathering variables
    private float[] timesTaken = new float[255];
    private float[] gridSizeUsed = new float[255];

    //Variables for visualization
    private Mesh gridMesh;
    public Vector3[] meshVertices;
    public int removeLines = 2;




    //variables for wake mechanics
    private float spawnInterval = 3.0f;
    private float lastTime = Time.time;

    // Use this for initialization
    void Start () {

        //Set initial conditions
        setInitialConditions();
        createGrid();
        findVorticities();
        determineCoefficients();
        determineMaximumDegree();


        //Convect(5, 5, new Vector3(0.0f, 0.0f, 0.0f), maxDegree, 0, xSize-1, 0,
        ↪  xSize-1);

    }

        // Update is called once per frame
        void Update () {


        //updatePositions();

```

```
60          if (Input.GetKeyDown("space") == true)
61          {
62              //Convect(7, 0, 2, 0, 7, 0, 7, 0, 0);
63              //convectionDispatcher();
64              //gatherData(1,0,0);
65              //gatherData(2, 4, 4);
66              //gatherData(2, 8, 4);
67              //gatherData(2, 12, 4);
68              //gatherData(2, 16, 4);
69              //gatherData(2, 16, 8);
70              //gatherData(2, 16, 12);
71              //gatherData(2, 16, 16);
72              //gatherData(2, 32, 16);
73              //gatherData(2, 32, 32);
74              gatherData(2, 16, 20);
75
76
77          }
78
79          }
80
81      private void FixedUpdate()
82      {
83          convectionDispatcher();
84          updateGrid();
85      }
86
87      //this function is just a nice container to store the conditions
88      void gatherData(int mode, int xVal, int yVal)
89      {
90
91          //first mode, simple times taken
92          if (mode == 1)
93          {
94
95              //lets get some for loops to cycle through some values, actually, Only y is
                    ↪  needed really...
96              //for (int x = clusterSize)
97              xSize = 32;
98              int testCount = 0;
99              for (int y = 1; y < 7; y++, testCount++)
100             {
101
102                 //preamble necessary for the iterations to work
103                 ySize = (int) Mathf.Pow(clusterSize,y);
104                 xSize = (int)Mathf.Pow(clusterSize, y);
105                 setInitialConditions();
106                 findVorticities();
107                 determineMaximumDegree();
108
109                 float initialTime = Time.realtimeSinceStartup;
110
111                 //now the for loop to run through 20 seconds of time
112                 for (int iterations = 0; iterations < 500; iterations++)
113                 {
114                     convectionDispatcher();
115                 }
116
117                 float endTime = Time.realtimeSinceStartup;
118
119                 timesTaken[testCount] = endTime - initialTime;
120                 gridSizeUsed[testCount] = xSize * ySize;
121             }
```

```
122
123             saveTimeCSV();
124         }
125
126         //second mode
127         if (mode == 2)
128         {
129             //preamble necessary for the iterations to work
130             xSize = xVal;
131             ySize = yVal;
132             setInitialConditions();
133             findVorticities();
134             determineMaximumDegree();
135
136             float initialTime = Time.realtimeSinceStartup;
137
138             //now the for loop to run through 20 seconds of time
139             for (int iterations = 0; iterations < 500; iterations++)
140             {
141                 convectionDispatcher();
142             }
143
144             float endTime = Time.realtimeSinceStartup;
145             savePositionCSV();
146             Debug.Log("For x: " + xVal + " y: " + yVal + " Time was: " + (endTime -
                 ↪  initialTime) );
147
148         }
149
150
151         Debug.Log("Done");
152
153     }
154
155     void determineMaximumDegree()
156     {
157
158         //to determine maximum degreee, find the last degree where
159         bool shouldContinue = true;
160         int cureDegree = 0;
161         while (shouldContinue == true)
162         {
163             if (xSize % Mathf.Pow(clusterSize,cureDegree) == 0 && ySize %
                 ↪  Mathf.Pow(clusterSize,cureDegree) == 0)
164             {
165                 cureDegree++;
166             }
167             else
168             {
169                 shouldContinue = false;
170             }
171
172         }
173
174         maxDegree = cureDegree - 1;
175     }
176
177     //function to set initial conditions (STRAIGHT FROM SIMPLE CONVECTION)
178     void setInitialConditions()
179     {
180         //set initial positiions and velocities
181         for (int x = 0; x < xSize; x++)
182         {
```

```
183             for (int y = 0; y < ySize; y++)
184             {
185                 elementPosition[x, y, 0] = new Vector3(x * gridSpacing, 0, y *
                    ↪  gridSpacing);
186                 elementVelocity[x, y] = new Vector3(0.0f, 0.0f, 0.0f);
187                 elementVorticity[x, y, 0] = new Vector2(0.0f, 0.0f);
188 
189                 //complete horseshoe
190                 if (y == 1)
191                 {
192                     elementVorticity[x, y, 0] = new Vector2(0.6f, -0.0f);
193                 }
194 
195                 //complete horseshoe
196                 if (y == 31)
197                 {
198                     elementVorticity[x, y, 0] = new Vector2(-0.6f, -0.0f);
199                 }
200 
201                 if (x == 1)
202                 {
203                     elementVorticity[x, y, 0] = new Vector2(0.0f, -0.4f);
204 
205                 }
206 
207                 if (x == xSize - 2)
208                 {
209                     elementVorticity[x, y, 0] = new Vector2(0.0f, 0.4f);
210                 }
211 
212 
213 
214             }
215         }
216     }
217 
218     //function to find vorticities (STRAIGHT FROM SIMPLE CONVECTION)
219     void findVorticities()
220     {
221 
222         //find initial vorticity vectors
223         for (int x = 1; x < xSize - 1; x++)
224         {
225             for (int y = 1; y < ySize - 1; y++)
226             {
227 
228                 //find vorticity in x and y
229                 Vector3 vorX = elementVorticity[x, y, 0].x * (elementPosition[x + 1, y,
                    ↪  0] - elementPosition[x - 1, y, 0]).normalized;
230                 Vector3 vorY = elementVorticity[x, y, 0].y * (elementPosition[x, y + 1,
                    ↪  0] - elementPosition[x, y - 1, 0]).normalized;
231                 vorT[x, y, 0] = vorX + vorY;
232 
233             }
234         }
235 
236     }
237 
238     //This function deterines cluster coefficients (THIS FUNCTION WORKS - NO MESSING)
239     void determineCoefficients()
240     {
241 
242         //cycle through levels of "clustering"
```

```
243         for ( int degree = 1; degree < maxDegree + 1; degree++)
244         {
245 
246             //Determine step size and
247             int stepSize = (int) Mathf.Pow(clusterSize, degree);
248             int xSteps = xSize / stepSize;
249             int ySteps = ySize / stepSize;
250 
251             //cycle through the points
252             for ( int x = 0; x < xSteps; x++)
253             {
254                 for ( int y = 0; y < ySteps; y++)
255                 {
256 
257                     //First reset the values we're interested in
258                     elementPosition[x, y, degree] = Vector3.zero;
259                     vorT[x, y, degree] = Vector3.zero;
260 
261                     //now that every index is cycled through for the given degree, we
                         ↪  need to determine their values
262                     for (int xIn = 0; xIn < clusterSize; xIn++)
263                     {
264                         for (int yIn = 0; yIn < clusterSize; yIn++)
265                         {
266 
267                             //perform sumations
268                             vorT[x, y, degree] += vorT[x * clusterSize + xIn, y *
                                 ↪  clusterSize + yIn, degree - 1];
269                             elementPosition[x, y, degree] += elementPosition[x *
                                 ↪  clusterSize + xIn, y * clusterSize + yIn, degree - 1];
270 
271                         }
272                     }
273 
274                     //Now the average needs to be taken for the position
275                     elementPosition[x, y, degree] = elementPosition[x, y, degree] /
                         ↪  Mathf.Pow(clusterSize, 2);
276 
277 
278                 }
279             }
280 
281 
282         }
283 
284     }
285 
286     //Function that ties all the parts together
287     void convectionDispatcher()
288     {
289 
290         //Perform Subroutines to find necessary values
291         findVorticities();
292         determineCoefficients();
293         resetAllVelocities();
294 
295 
296         //Now we cycle through all elements and run the convection routine
297         for (int x = 0; x < xSize; x++)
298         {
299             for (int y = 0; y < ySize; y++)
300             {
301                 Convect(x, y, maxDegree, 0, xSize, 0, ySize, 0, 0);
```

```
302             }
303         }
304 
305         //Now implement the discretizaton scheme
306         updatePositions();
307         updateWake();
308 
309     }
310 
311     //Function to reset velocities
312     void resetAllVelocities()
313     {
314 
315         //Reset all velocities to zero
316         for (int x = 0; x < xSize; x++)
317         {
318             for (int y = 0; y < ySize; y++)
319             {
320                 //this cycles through all element, now reset their velocities
321                 elementVelocity[x, y] = Vector3.zero;
322             }
323         }
324 
325     }
326 
327     //This function finds the appropriate elements and abstracted clusters for the
        ↪  given element
328     void Convect(int xg, int yg, int degree, int xs, int xe, int ys, int ye, int xBias,
        ↪  int yBias)
329     {
330 
331         //determine grid spacing
332         int spacing = (int) Mathf.Pow(clusterSize, degree);
333         //Debug.Log(spacing);
334         //Debug.Log("Current Degree: " + degree);
335 
336         //first step is to determine the current cluster size
337         int xGridSize = 0;
338         int yGridSize = 0;
339         if (degree != 0)
340         {
341             xGridSize = (xe - xs + 1) / spacing;
342             yGridSize = (ye - ys + 1) / spacing;
343         }
344         else
345         {
346             xGridSize = 2;
347             yGridSize = 2;
348         }
349         //Debug.Log(xGridSize);
350         //Debug.Log(yGridSize);
351 
352 
353         //determine current position on the grid
354         int xgp = 0;    //start by declaring variables for our position on the "local"
            ↪  grid
355         int ygp = 0;    //And againt for the y position
356         //for for loops to determine their values
357         for (int x = 0; x < xGridSize; x++)
358         {
359             if ( xg >= (xs + x*spacing) && xg < (xs + (x + 1) * spacing))
360             {
361                 xgp = x;
```

```
362             }
363         }
364         //now determine the y coordinate
365         for (int y = 0; y < yGridSize; y++)
366         {
367             if ( yg >= (ys + y * spacing) && yg < (ys + (y + 1) * spacing))
368             {
369                 ygp = y;
370             }
371         }
372 
373         //Now cycle through clsuters
374         for ( int x = 0; x <  xGridSize; x++)
375         {
376             for ( int y = 0; y < yGridSize; y++)
377             {
378                 if ( x == xgp && y == ygp)
379                 {
380                     //Check if we're at the base level abstraction
381                     if (degree > 0)
382                     {
383                         Convect(xg, yg, degree - 1, xs + (xgp * spacing), xs +
                             ↪  (xgp+1)*spacing, ys + (ygp * spacing), ys + (ygp + 1) *
                             ↪  spacing, (xBias + x) * clusterSize, (yBias + y) *
                             ↪  clusterSize);
384                     }
385                 }
386                 else
387                 {
388                     biotSavart(xg, yg, x + xBias, y + yBias, degree);
389                 }
390             }
391         }
392 
393     }
394 
395     //This function implements the biot-savart law maybe
396     void biotSavart(int ex, int ey, int cx, int cy, int degree)
397     {
398 
399         //This calculate the influence
400         //First the radius,
401         Vector3 radius = elementPosition[ex, ey, 0] - elementPosition[cx, cy, degree];
402         //Now the Biot-Savart law itself
403         elementVelocity[ex, ey] += (1f / (4f * Mathf.PI)) * (Vector3.Cross(vorT[cx, cy,
             ↪  degree], radius) / (Mathf.Pow(radius.magnitude, 2)));
404 
405     }
406 
407     //update positions (FROM SIMPLE CONVECTION - IS EULERS METHOD)
408     void updatePositions()
409     {
410 
411         //for loops to cycle through grid
412         for (int x = 0; x < xSize; x++)
413         {
414             for (int y = 0; y < ySize; y++)
415             {
416                 if (y != 0)
417                 {
418                     elementPosition[x, y, 0] = elementPosition[x, y, 0] + 0.02f *
                         ↪  (elementVelocity[x, y] + freeStreamVelocity);
419                 }
```

```
420
421             }
422         }
423
424     }
425
426     //Function to write a csv file containing the times taken (STRAIGHT FROM SIMPLE
        ↪  CONVECTION)
427     void saveTimeCSV()
428     {
429
430         using (StreamWriter writetext = new StreamWriter("TimesTaken.csv"))
431         {
432
433             //Write Initial headings
434             writetext.WriteLine("Number of Elements" + "," + "Computational Time");
435
436             //cycle through the array and write the data value
437             for (int val = 0; val < timesTaken.Length; val++)
438             {
439                 writetext.WriteLine(gridSizeUsed[val] + "," + timesTaken[val]);
440             }
441         }
442
443     }
444
445     //Function to write positions to  csv file
446     void savePositionCSV()
447     {
448
449
450         //file to write data into
451         using (StreamWriter writetext = new StreamWriter("Output.csv"))
452         {
453
454             //Write Initial headings
455             writetext.WriteLine("X Grid Position, Y  Grid Position, VecX, VecY, VecZ");
456
457             for (int x = 0; x < xSize; x++)
458             {
459                 for (int y = 0; y < ySize; y++)
460                 {
461                     writetext.WriteLine(x + "," + y + "," + elementPosition[x, y, 0].x
                        ↪  + "," + elementPosition[x, y, 0].y + "," + elementPosition[x,
                        ↪  y, 0].z);
462                 }
463             }
464         }
465
466
467     }
468
469     //This function creates the inital grid
470     void createGrid()
471     {
472
473         //Get required references via searches
474         GetComponent<MeshFilter>().mesh = gridMesh = new Mesh(); gridMesh.name = "Grid
            ↪  Mesh";
475
476         //start by clearing the grid (so this function can be called numerous times in
            ↪  the progra)
477         gridMesh.Clear();
```

```
478
479         //now buffer the grid to the correct size
480         meshVertices = new Vector3[xSize * ySize];
481
482         //set initial values of the vertices
483         for (int y = 0; y < ySize; y++)
484         {
485             for (int x = 0; x < xSize; x++)
486             {
487                 meshVertices[y * xSize + x] = elementPosition[x, y, 0];
488             }
489         }
490
491         //set the newly found verices to the mesh vertices
492         gridMesh.vertices = meshVertices;
493
494         //now create the triangles array
495         int[] triangles = new int[xSize * (ySize - 1) * 3 * 2];
496         int gridRowsInt = xSize;
497         int gridColumnsInt = ySize;
498
499         //now populate the array
500         for (int nc = 0, n = 3; nc < (gridColumnsInt - 1); nc++)
501         {
502             for (int nr = 0; nr < gridRowsInt - 1; nr++, n = n + 3)
503             {
504
505                 //These tri's are the lower quarter of a quad           (this could be
                    ↪  done far more compactly, combining upper and lower quarters, but
                    ↪  this is far easier to read)
506                 triangles[n] = gridRowsInt * nc + nr;
507                 triangles[n + 1] = gridRowsInt * (nc + 1) + nr + 1;
508                 triangles[n + 2] = gridRowsInt * nc + nr + 1;
509
510                 //These tri's are the upper quarter of a quad
511                 triangles[n + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt *
                    ↪  nc + nr;
512                 triangles[n + 1 + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt
                    ↪  * (nc + 1) + nr;
513                 triangles[n + 2 + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt
                    ↪  * (nc + 1) + nr + 1;
514
515             }
516         }
517
518         //Create the grid described by triangles and vertices arrays
519         gridMesh.triangles = triangles.Concat(triangles.Reverse().ToArray()).ToArray();
             ↪  ;
520         //gridMesh.uv = uv;
521         gridMesh.RecalculateNormals();
522
523     }
524
525     //function to update grid
526     void updateGrid()
527     {
528
529         //This functions just manages the the vertex-vortonpositon relationship
530         //Seed the vorton position array with coordinates for the vortons
531         for (int nc = 0, n = 0; nc < ySize; nc++)
532         {
533             for (int nr = 0; nr < xSize; nr++, n++)
534             {
```

```
                meshVertices[n] = elementPosition[nc, nr, 0];
            }
        }

        //create the mesh for the grids
        gridMesh.vertices = meshVertices;
        gridMesh.RecalculateNormals();

    }

    //function to handle wake mechanics
    void updateWake()
    {

        //this checks if its time to shift the cells
        if (Time.time - lastTime > spawnInterval)
        {
            lastTime = Time.time;
            //Debug.Log("doin a work");

            for (int y = ySize - 2; y > 0; y--)
            {
                //Debug.Log(y);

                for (int x = 0; x < xSize; x++)
                {
                    elementPosition[x, y + 1, 0] = elementPosition[x, y, 0];
                    //elementVorticity[x, y + 1, 0] = elementPosition[x, y, 0];
                    //elementVelocity[x, y + 1] = elementVelocity[x, y];
                }
            }

            //create new elements
            for (int x = 0; x < xSize; x++)
            {
                elementPosition[x, 1, 0] = new Vector3(x * gridSpacing, 0,
                 ↪  gridSpacing);
            }

        }


    }

    //Draw the elements so they're visible
    private void OnDrawGizmos() // this shows control points as red spheres in the edit
     ↪  window (MODIFIED FROM SIMPLE CONVECTION)
    {

        Gizmos.color = Color.red;
        //cycle through all elements
        for (int x = 0; x < xSize; x++)
        {
            for (int y = 0; y < ySize; y++)
            {

                //draw the gizmo
                Gizmos.DrawSphere(elementPosition[x, y,0], 0.1f);
            }
        }

        Gizmos.color = Color.blue;
        //cycle through all elements
```

```
596         for (int x = 0; x < xSize / 2; x++)
597         {
598             for (int y = 0; y < ySize / 2; y++)
599             {
600 
601                 //draw the gizmo
602                 //Gizmos.DrawSphere(elementPosition[x, y, 1] + new Vector3(0.0f, 0.5f,
                        ↪  0.0f), 0.1f);
603             }
604         }
605 
606         Gizmos.color = Color.green;
607         //cycle through all elements
608         for (int x = 0; x < xSize / 4; x++)
609         {
610             for (int y = 0; y < ySize / 4; y++)
611             {
612 
613                 //draw the gizmo
614                 //Gizmos.DrawSphere(elementPosition[x, y, 2] + new Vector3(0.0f, 1.0f,
                        ↪  0.0f), 0.1f);
615             }
616         }
617 
618     }
619 
620 }
```

17 Appendix: Discretization Experiments Script

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Discretization : MonoBehaviour {

    //Test Variables
    public Vector3 testPosition = new Vector3(5.0f, 5.0f, 5.0f);
    public Vector3 testVelocity = new Vector3(2.0f, 2.0f, 2.0f);
    public float testTimeStep = 2.0f;

    //Shared Variables for all Schemes
    private Vector3[] elementPositions = new Vector3[255];
    private Vector3[] elementVelocities = new Vector3[255];
    public float timeIncrement = 2.0f;

    //Variables Unique to the Quadratic Position Scheme
    private Vector3[] elementPastPositions = new Vector3[255];
    public bool[] eulerMode;

    //Variables Unique to the Quadratic Velocity Scheme
    private Vector3[,] elementPastVelocities = new Vector3[255, 2];
    public int[] eulerCount;


    //variables for testing purpose
    public int iterationCounter;
    public int currentIterations;
    public int iterationIncrement;

    // Use this for initialization
    void Start () {

        //set initial conditions to allow for testings of schemes
        //None specific conditions
        elementPositions[0] = new Vector3(5.0f, 5.0f, 5.0f);
        elementVelocities[0] = new Vector3(2.0f, 2.0f, 2.0f);
        //QP conditions
        elementPastPositions[0] = new Vector3(3.0f, 3.0f, 3.0f);
        eulerMode = new bool[255];
        eulerMode[0] = false;
        //QV conditions
        eulerCount = new int[255];
        eulerCount[0] = 0;
        elementPastVelocities[0, 0] = new Vector3(2.0f, 2.0f, 2.0f);
        elementPastVelocities[0, 1] = new Vector3(2.0f, 2.0f, 2.0f);
        //other
        iterationCounter = 50;
        currentIterations = 0;
        iterationIncrement = 500000;

        Debug.Log(Time.realtimeSinceStartup);

    }

    // Update is called once per frame
    void Update () {


        if (iterationCounter > 0)
```

```
61          {
62              // Debug.Log("For "+currentIterations+": "+testIterationTime(2,
                ↪   currentIterations, 0));
63              currentIterations = currentIterations + iterationIncrement;
64              iterationCounter = iterationCounter - 1;
65          }
66
67
68      }
69
70      //Function to Iterate Eulers Method
71      public Vector3 EulerIterate(int elementID)
72      {
73
74          //Get Required Information from the Element ID
75          Vector3 currentPosition = elementPositions[elementID];
76          Vector3 currentVelocity = elementVelocities[elementID];
77
78          //Apply Eulers Methods
79          Vector3 newPosition = currentPosition + (timeIncrement * currentVelocity);
80
81          //Return calcualted new position
82          return newPosition;
83
84      }
85
86      //Function to Iterate using the Quadratic Position Scheme
87      public Vector3 QPIterate(int eID)
88      {
89
90          Vector3 newPos = new Vector3(0.0f, 0.0f, 0.0f);
91
92          //Determine whether Euler's method should be used
93          if (eulerMode[eID] == true)
94          {
95              newPos = EulerIterate(eID);
96              eulerMode[eID] = false;
97          }
98          else
99          {
100             newPos = elementPastPositions[eID] + (2f * timeIncrement *
                    ↪   elementVelocities[eID]);
101         }
102
103         elementPastPositions[eID] = elementPositions[eID];         //This stores the
                ↪   value for the next iteration
104
105        return newPos;
106     }
107
108     //Function to Iterate using the Quadratic Velocity Scheme
109     public Vector3 QVIterate(int eID)
110     {
111
112         //Declare variable with null value to define local scope in function
113         Vector3 newPos = new Vector3(0.0f, 0.0f, 0.0f);
114
115         if (eulerCount[eID] != 0)
116         {
117
118             //Call Function to Iterate via Eulers Method
119             newPos = EulerIterate(eID);
120
```

```
121             //reduce the ammont eulers method needs to be used
122             eulerCount[eID] -= 1;
123 
124         }
125         else
126         {
127             //Debug.Log("Doin a ting");
128             //Use the QV Scheme to find new position
129             newPos = elementPositions[eID] + timeIncrement * ((5f / 12f) *
                ↪   elementPastVelocities[eID, 0]-(4f/3f)* elementPastVelocities[eID,
                ↪   1]+(23f/12f)*elementVelocities[eID]);
130         }
131 
132         //shift past velocities array
133         elementPastVelocities[eID, 0] = elementPastVelocities[eID, 1];
134         elementPastVelocities[eID, 1] = elementVelocities[eID];
135 
136         return newPos;
137 
138     }
139 
140     //Function to perform
141     public float testIterationTime(int scheme, int iterationCount, int eID)
142     {
143 
144 
145         //create variable to set iterations to
146         Vector3 testPosition;
147         float iterationTime = 0;
148 
149         //series of if loops to determine which scheme to use
150         if (scheme == 1)
151         {
152             //Take initial time reading
153             var initialTime = Time.realtimeSinceStartup;
154 
155             for (int itNo = 0; itNo < iterationCount; itNo++)
156             {
157 
158                 //Use Eulers Method
159                 testPosition = EulerIterate(eID);
160 
161             }
162 
163             //Take end time
164             var endTime = Time.realtimeSinceStartup;
165 
166             //return time take
167             iterationTime = (endTime - initialTime);
168         }
169         if (scheme == 2)
170         {
171             //Take initial time reading
172             var initialTime = Time.realtimeSinceStartup;
173 
174             for (int itNo = 0; itNo < iterationCount; itNo++)
175             {
176 
177                 //Use QP Scheme
178                 testPosition = QPIterate(eID);
179 
180             }
181 
```

```
182             //Take end time
183             var endTime = Time.realtimeSinceStartup;
184 
185             //return time take
186             iterationTime = (endTime - initialTime);
187         }
188 
189         if (scheme == 3)
190         {
191             //Take initial time reading
192             var initialTime = Time.realtimeSinceStartup;
193 
194             for (int itNo = 0; itNo < iterationCount; itNo++)
195             {
196 
197                 //Use QV scheme
198                 QVIterate(eID);
199 
200             }
201 
202             //Take end time
203             var endTime = Time.realtimeSinceStartup;
204 
205             //return time take
206             iterationTime = (endTime - initialTime);
207         }
208 
209         //return the tie taken
210         return iterationTime;
211 
212     }
213 
214 }
```

18 Appendix: Algorithm Options Script (Interim)

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;
using System;

public class VortonOptionsScript : MonoBehaviour {

    //Declaring Input Variables
    public InputField rows;
    public InputField columns;
    public Button spawnButton;
    public Button clearButton;
    public GameObject vortexProbeParticle;
    public Transform probePointCollection;

    //Declaring spawn related variables
    public float vortonSpacing;
    public float vortonStrength;
    private Vector3 vortonPosition;
    public Vector3 vortonVelocity;

    //Conversion related vaiables
    public int rowsInt;
    public int columnsInt;

    //Array to hold blobs
    private GameObject[] blobArray;

    // Use this for initialization
    void Start () {

        //Add click events
        spawnButton.onClick.AddListener(SpawnButton);
        clearButton.onClick.AddListener(ClearButton);

    }

        // Update is called once per frame
        void Update () {

    }

    void SpawnButton() {

        //Button even indicator


        //Convert InputField strings to Integers
        rowsInt = Convert.ToInt32(rows.text);
        columnsInt = Convert.ToInt32(columns.text);

        //Two for loops to produce vortons in a grid shape
        for (int x = 0; x < rowsInt; x++)
        {
            for (int y = 0; y < columnsInt; y++)
            {

                //Produce necessary variables for the instantiated vortons
                vortonPosition = new Vector3(x * vortonSpacing, 1, y * vortonSpacing);

```

```
61                  //Produce actual Vortons
62                  GameObject clone = Instantiate(vortexProbeParticle,vortonPosition,
                     ↪  Quaternion.identity) as GameObject;
63
64                  //Puts the new blobs in a eay to handle place
65                  clone.transform.parent = probePointCollection;
66
67                  //Sets some random values for vortex parameters so the result math isnt
                     ↪  zero
68                  VortexBlobBehaviour scr = clone.GetComponent<VortexBlobBehaviour>();
                     ↪  //Get the blob behaivour script
69                  scr.VortexBlobStrength = vortonStrength;
70                  scr.initialisationVelocity = vortonVelocity;
71                  scr.initialVelocity = Vector3.zero;
72
73              }
74
75          }
76
77      }
78
79      void ClearButton()
80      {
81
82          //Populate array with current blobs
83          blobArray = GameObject.FindGameObjectsWithTag("VortexBlob");
84
85          //Delete all current blobs
86          for (int id = 0; id <blobArray.Length; id++)
87          {
88              Destroy(blobArray[id]);
89          }
90
91      }
92
93  }
```

19 Appendix: Discretization script (Interim)

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class positionInterpolations : MonoBehaviour {

    //This script is for different interpolating scheme for the velocity time integration

    //Variable for simple linear interpolation
    public Vector3[,] newPositions;

    //Variables for quadratic interpolation
    public int quadraticIterations = 3;
    public Vector3[,,] oldVelocities;

    //test variables
    public Vector3[,] testPositions = new Vector3[5,5];
    public Vector3[,] testVelocities = new Vector3[5, 5];

    // Use this for initialization
    void Start () {

        //Set test variables up
        for (int x = 0; x < 5; x++)
        {
            for (int y = 0; y < 5; y++)
            {
                testPositions[x, y] = new Vector3(x, y, 0.0f);
            }
        }
        for (int x = 0; x < 5; x++)
        {
            for (int y = 0; y < 5; y++)
            {
                testVelocities[x, y] = new Vector3(x, y, 1.0f);
            }
        }

        //Create buffers for quadratic interpolation
        oldVelocities = new Vector3[50, 50, 3];
    }

        // Update is called once per frame
        void Update () {

        }

    //Fixed update, most interpolating scheme should go in here to be time controlled with the blob method
    void FixedUpdate()
    {
        testPositions = quadraticInterpolation(testPositions, testVelocities);
    }

    //Simple Linear Interpolation
    public Vector3[,] linearInterpolation(Vector3[,] position, Vector3[,] velocity)
    {

        //Define initial variables
```

```
59          int xLength = position.GetLength(0);
60          int yLength = position.GetLength(1);
61          newPositions = new Vector3[xLength,yLength];
62
63          //Simple linear interpolation for position
64          for (int x = 0; x < xLength; x++)
65          {
66              for (int y = 0; y < yLength; y++)
67              {
68                  newPositions[x, y] = position[x, y] + velocity[x,y]*Time.deltaTime;
69              }
70          }
71
72
73          //return new position
74          return newPositions;
75
76      }
77
78      //Quadratic Interpolation
79      public Vector3[,] quadraticInterpolation(Vector3[,] position, Vector3[,] velocity)
80      {
81
82          //Define initial variables
83          int xLength = position.GetLength(0);
84          int yLength = position.GetLength(1);
85          newPositions = new Vector3[xLength, yLength];
86
87          //Check to see if a first order scheme is required to start the interpolation
             ↪  (requried to get 3 data points for the polynomial
88          if (quadraticIterations != 0)
89          {
90              //reduce iteration counter for linear interpolation and active the linear
                 ↪  interpolation function
91              //quadraticIterations = quadraticIterations - 1;
92              newPositions = linearInterpolation(position, velocity);
93
94              //shift the old values matrix (programmed in a modular way so higher order
                 ↪  schemes can be implemented
95              for (int t = 0; t < oldVelocities.GetLength(2)-2; t++)
96              {
97                  for (int x = 0; x < xLength; x++)
98                  {
99                      for (int y = 0; y < yLength; y++)
100                     {
101                         oldVelocities[x, y, t + 1] = oldVelocities[x, y, t];
102                     }
103                 }
104             }
105             //Creat new values for the current time
106             for (int x = 0; x < xLength; x++)
107             {
108                 for (int y = 0; y < yLength; y++)
109                 {
110                     oldVelocities[x, y, 0] = velocity[x, y];
111                 }
112             }
113
114         }
115         else     //This is where the actual interpolation goes
116         {
117
118         }
```

```
119
120         //return the new positions
121         return newPositions;
122
123     }
124
125     //This function draws gizmos so that each blob can be visualised
126     private void OnDrawGizmos()
127     {
128
129         Gizmos.color = Color.red;
130         for (int x = 0; x < 5; x++)
131         {
132             for (int y = 0; y < 5; y++)
133             {
134                 Gizmos.DrawSphere(testPositions[x, y], 0.1f);     // this shows control
                    ↪  points as red spheres in the edit window
135             }
136         }
137
138     }
139 }
```

20 Appendix: Iteration Handler (Interim)

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

//This script manages whether or not the physics engine should run

public class PhysicsHandler : MonoBehaviour {

    //References to necessary UI components
    public Toggle contToggle;                   //This determines whether the physics engine should run continously or for a set number of iteration
    public InputField iterationCountInput;          //If the physics engine should run a certain ammount of iteration, this is the number of iterations
    public Button pauseButton;                  //Button to stop iterations
    public Button startButton;                  //Button to start iterations

    //Varaibes for iteration counter
    public int iterationCount;
    private bool toggleBool = true;


        // Use this for initialization
        void Start () {

        //Add listener function so button presses active required methods
        pauseButton.onClick.AddListener(pausePhysics);
        startButton.onClick.AddListener(startPhysics);

        //Add toggle listener to flip flop the bool
        contToggle.onValueChanged.AddListener(flipFlopToggle);

        //Set the initial time step to zero so the physics engine doesnt run at startup
        Time.timeScale = 0;

        }

        // Update is called once per frame
        void Update () {

        //Determine whether of not physics engine should be active
        if (iterationCount == 0)
        {
            Time.timeScale = 0;             //These are doen in Update as FixedUpdate() may not exectute under some conditions
        }
        else
        {
            Time.timeScale = 1;
        }
        }

    //Update called once per update of physics engine
    void FixedUpdate()
    {
        if (iterationCount > 0)
        {

            if (toggleBool == false)
```

```
58              {
59                  iterationCount = iterationCount - 1;
60              }
61
62          }
63
64
65
66      }
67
68      //Function to flip flop the toggle boolean
69      void flipFlopToggle(bool flipVar)
70      {
71
72          //In unity you cant access the togges value directly, only when its changed,
             ↪  this keeps track of changes (toggleBool must be private or editor
             ↪  interations can override its initial value)
73          if (toggleBool == true)
74          {
75              toggleBool = false;
76          }
77          else
78          {
79              toggleBool = true;
80          }
81
82
83      }
84
85      //Function for when the start button is pressed
86      void startPhysics()
87      {
88          if (toggleBool == true)
89          {
90              iterationCount = 1;                                                    //Simple
                 ↪  way to use the iteration counter for inifinite iterations
91          }
92          else
93          {
94              iterationCount = Convert.ToInt32(iterationCountInput.text);     //Sets
                 ↪  iteration count to required number
95          }
96      }
97
98      //Function for if the pause button is pressed
99      void pausePhysics()
100     {
101         iterationCount = 0;         //Ensure physics engine is paused
102     }
103 }
```

21 Appendix: Convection Script (Interim)

```
using System;
using System.Linq;
//using System.Collections;
//using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class GridHandler : MonoBehaviour {

    //Section for all necessary interface references
    public Button createGrid;               //This is the button that create the grid, this will probable be changed at a later date to real time production
    public InputField  gridRows;                    //variable to store the inpit field for number of rows
    public InputField gridColumns;              //variable to store input field for number of columns
    public InputField gridSpacing;              //variables to store input field for the the grid spacing (this could be changed later for non square spacing)
    private int gridRowsInt;
    private int gridColumnsInt;
    private int gridSpacingInt;

    //Section for arrays that govern the vortex blob position and strength
    private Vector3[,] vortonPositions;
    private Vector3[] meshVertices;

    //Section for the grid mesh arrays+variables+objects
    private Mesh gridMesh;
    private bool gridSpawned = false;
    private bool meshSpawned = false;
    private float spawnTimer = 5f;
    private float lastTime;
    private int initialRows = 0;

    //Convection related variables
    private float blobDistance;
    private Vector3 blobVelocityDirection;
    private float blobVelocityMagnitude;
    private float[,] blobVorticity;
    private Vector3 blobFilamentDifferential;
    private Vector3[,] blobInitialVelocity;
    private Vector3[,] blobVelocity;

    //Clustered convection variables and arrays (the above are shared betwean the simple and clustered algorithms)
    private Vector3[,] clusterPositionCoef;
    public float[,] clusterVorticityCoef;
    private Vector3[,] clusterInitialVelocityCoef;
    private float[,] clusterVelocityCoef;
    private int clusterSize = 3;
    private float vorticityCount;
    private int clusterXIndex;
    private int clusterYIndex;


        // Use this for initialization
        void Start () {

        //Assign all UI buttons their respective listeners (functions/methods called per click
```

```
55          createGrid.onClick.AddListener(CreateGridCall);
56
57          //Get required references via searches
58          GetComponent<MeshFilter>().mesh = gridMesh = new Mesh();    gridMesh.name =
             ↪  "Grid Mesh";
59
60
61      }
62
63          // Update is called once per frame
64          void Update () {
65
66
67      }
68
69      //Update is called once per physics engine update (use for convection math if the
         ↪  rigid body technique is used, however may be replaced with a second order
         ↪  accurate scheme for the integration)
70      void FixedUpdate()
71      {
72
73          //call functions required
74          RenderGridMesh();               //This function is responsible for creating
             ↪  the mesh object on the vortex plane
75                                          //sampleAnimation();            //Use this to
                                             ↪  test the vertex/position arrayy relation
                                             ↪  (or rendering stuff if convection cant be
                                             ↪  used)
76          handleGrid();
77          //simpleConvection();
78          clusteredConvection();
79
80
81      }
82
83      //This is the function to create the grid, it is attahed as a listener to a button
         ↪  component above
84      void CreateGridCall()
85      {
86
87          //Necessary variable conversions
88          gridSpacingInt = Convert.ToInt32(gridSpacing.text);
             ↪  //This is converted here as its required for every grid spacing (as
             ↪  opposed to the other conversions that are required only once)
89          gridRowsInt = Convert.ToInt32(gridRows.text);
             ↪  //Same as above
90          gridColumnsInt = Convert.ToInt32(gridColumns.text);
             ↪  //Same as above
91          vortonPositions = new Vector3[Convert.ToInt32(gridRows.text),
             ↪  Convert.ToInt32(gridColumns.text)];   //2D Array to hole vorton position
             ↪  (public and declared earlier, however size specified here)
92          blobInitialVelocity = new Vector3[Convert.ToInt32(gridRows.text),
             ↪  Convert.ToInt32(gridColumns.text)];
93          blobVelocity = new Vector3[Convert.ToInt32(gridRows.text),
             ↪  Convert.ToInt32(gridColumns.text)];
94          blobVorticity = new float[Convert.ToInt32(gridRows.text),
             ↪  Convert.ToInt32(gridColumns.text)];
95          meshVertices = new Vector3[(gridRowsInt * gridColumnsInt)];
             ↪  //1D Array of vorton points, used for vertices on the
96          gridMesh.Clear();
             ↪  //Ensures the mesh is blank, helps with retroactively creating new grids
             ↪  in enviroment but considerations to all buffers is not yet given
97
```

```
98          //The following viarbes/arrays are for the clustered convection algorithm
99          clusterVorticityCoef = new float[gridRowsInt, gridColumnsInt];
100         clusterVelocityCoef = new float[gridRowsInt, gridColumnsInt];
101         clusterPositionCoef = new Vector3[gridRowsInt, gridColumnsInt];
102         clusterInitialVelocityCoef = new Vector3[gridRowsInt, gridColumnsInt];
103
104
105         //Seed the vorton position array with coordinates for the vortons
106         for (int nc = 0, n = 0; nc <gridColumnsInt; nc++)
107         {
108             for (int nr = 0; nr < gridRowsInt; nr++, n++)
109             {
110                 //vortonPositions[nr, nc] = new Vector3(nr * gridSpacingInt, 0.1f, nc *
                        gridSpacingInt);
111                 vortonPositions[nr, nc] = new Vector3(0, 0.1f, nc * gridSpacingInt);
112                 blobVorticity[nr, nc] = 0;
                        //-0.1f*Mathf.Pow((nc-(gridRowsInt/2)),3);
113                 blobInitialVelocity[nr,nc] = new Vector3(2.5f, 0, 0.0f);
114                 blobVelocity[nr,nc] = new Vector3(0.0f, 0, 0);
115                 meshVertices[n] = new Vector3(nr * gridSpacingInt, 0.1f, nc *
                        gridSpacingInt);
116             }
117         }
118
119         //create the mesh for the grids
120         gridMesh.vertices = meshVertices;
121
122         //Create triangle array of appropriate size
123         int[] triangles = new int[gridRowsInt*(gridColumnsInt-1)*3*2];
124
125         //Populate the triangle array with vertices
126         for (int nc = 0, n = 3; nc < (gridColumnsInt-1); nc++)
127         {
128             for (int nr = 0; nr < gridRowsInt-1; nr++, n = n + 3)
129             {
130
131                 //These tri's are the lower quarter of a quad           (this could be
                        done far more compactly, combining upper and lower quarters, but
                        this is far easier to read)
132                 triangles[n] = gridRowsInt * nc + nr;
133                 triangles[n + 1] = gridRowsInt * (nc + 1) + nr + 1;
134                 triangles[n + 2] = gridRowsInt * nc + nr + 1;
135
136                 //These tri's are the upper quarter of a quad
137                 triangles[n + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt *
                        nc + nr;
138                 triangles[n + 1 + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt
                        * (nc + 1) + nr;
139                 triangles[n + 2 + gridRowsInt * (gridColumnsInt - 1) * 3] = gridRowsInt
                        * (nc + 1) + nr + 1;
140
141             }
142         }
143
144         //Create the grid described by triangles and vertices arrays
145         gridMesh.triangles = triangles.Concat(triangles.Reverse().ToArray()).ToArray();
            ;
146         //gridMesh.uv = uv;
147         gridMesh.RecalculateNormals();
148         //gridMesh.Optimize();
149
150         //Set values for all variables other functions rely on
```

```
151         gridSpawned = true;                                         //This is used when
             ↪  new grids are made after an itial grid and the old grid needs to be
             ↪  removed
152         lastTime = Time.realtimeSinceStartup;                       //this is used by
             ↪  the grid handler to determine when to spaum new rows
153         initialRows = gridRowsInt;
154
155
156     }
157
158     void RenderGridMesh()
159     {
160
161         //This functions just manages the the vertex-vortonpositon relationship
162         //Seed the vorton position array with coordinates for the vortons
163         for (int nc = 0, n = 0; nc < gridColumnsInt; nc++)
164         {
165             for (int nr = 0; nr < gridRowsInt; nr++, n++)
166             {
167                 meshVertices[n] = vortonPositions[nc,nr];
168             }
169         }
170
171         //create the mesh for the grids
172         gridMesh.vertices = meshVertices;
173         gridMesh.RecalculateNormals();
174
175     }
176
177     void sampleAnimation()
178     {
179
180         //simple sine wave animation to check the grids moves properly
181         if (gridSpawned == true)
182         {
183             for (int x = 0; x < gridRowsInt; x++)
184             {
185                 for (int y = 0; y < gridRowsInt; y++)
186                 {
187                     vortonPositions[x, y] = new Vector3(gridSpacingInt *
                         ↪  x,Mathf.Sin(Time.time+x*y),gridSpacingInt*y);
188                 }
189             }
190         }
191
192     }
193
194     //Function to handle simple un optimized convection
195     void simpleConvection()
196     {
197
198         //Clear Required Variables
199
200         //Loop to cycle through all vortex blobs
201         for (int x = 1; x < gridRowsInt; x++)
202         {
203             for (int y =0; y < gridColumnsInt; y++)
204             {
205                 //Reset filament differential
206                 blobFilamentDifferential = Vector3.zero;
207
208                 //Now this will cycle through all blobs, but then all other blobs must
                     ↪  be cycled through
```

```
209                 for (int nr = 0; nr < gridRowsInt; nr++)
210                 {
211                     for(int nc = 0; nc < gridColumnsInt; nc++)
212                     {
213                         if ( x != nr && y != nc)
214                         {
215 
216                             //Convect the blobs effect
217                             blobDistance =
                                ↪  Vector3.Distance(vortonPositions[x,y],vortonPositions[nr,nc]);
218                             blobVelocityDirection = Vector3.Cross((vortonPositions[nr,
                                ↪  nc] - vortonPositions[x, y]).normalized,
                                ↪  blobInitialVelocity[nr, nc]);
219                             blobVelocityMagnitude = blobVorticity[nr,nc] / (2.0f *
                                ↪  Mathf.PI * blobDistance);
220                             blobFilamentDifferential += blobVelocityMagnitude *
                                ↪  blobVelocityDirection;
221 
222                         }
223                     }
224                 }
225 
226                 //set new velocity value
227                 blobVelocity[x, y] = blobInitialVelocity[x,y] +
                    ↪  blobFilamentDifferential;
228                 //Debug.Log(blobVelocity[x, y]);
229                 blobFilamentDifferential = Vector3.zero;
230                 vortonPositions[x,y] += blobVelocity[x,y]*Time.deltaTime;
231             }
232         }
233     }
234 
235     //This is the clustered convection agorithm, this should run much faster
236     void clusteredConvection()
237     {
238 
239         //First step in the method is to create a matrix of cluster vorticity and
            ↪  location coficients
240         for (int x = 0, xindex = 0; x < (gridRowsInt); x = x + clusterSize, xindex++)
241         {
242             for (int y = 0, yindex = 0; y < (gridColumnsInt); y = y + clusterSize,
                ↪  yindex++)
243             {
244 
245                 //Reset the vorticity count
246                 vorticityCount = 0.0f;
247 
248                 //inside this loop will give cycle through the corner points of eah
                    ↪  custer
249                 //This next loop cycles through the vortons that should be in the
                    ↪  current cluster at corner point x,y
250                 for (int nx = 0; nx < clusterSize; nx++)
251                 {
252                     for (int ny = 0; ny < clusterSize; ny++)
253                     {
254 
255                         //summate all the vorticities
256                         vorticityCount = vorticityCount + blobVorticity[x+nx,y+ny];
257 
258                     }
259                 }
260 
261                 //set the vorticity coefficient
```

```
262                 clusterVorticityCoef[xindex, yindex] = vorticityCount;
263
264                 //Set the clusters equivalent position
265                 clusterPositionCoef[xindex, yindex] = vortonPositions[x+1,y+1];
                     ↪   //This is a very crude midpoint aproximation, however as a first
                     ↪   order scheme it will suffice
266                 clusterInitialVelocityCoef[xindex, yindex] = blobInitialVelocity[x + 1,
                     ↪   y + 1];
267
268             }
269         }
270
271         //The blobs are now clustered into clusters of size clusterSize, now the
             ↪   convection maths needs to be performed
272         //This loop cycles through the clusters and convects each blob in a given
             ↪   cluster
273         for (int cx = 0; cx < (gridRowsInt/clusterSize); cx++)
274         {
275             for (int cy = 0; cy < (gridColumnsInt/clusterSize); cy++)
276             {
277
278                 // This loop cycles through the individual blobs in a cluster
279                 for (int bx = 0; bx < clusterSize; bx++)
280                 {
281                     for (int by = 0; by < clusterSize; by++)
282                     {
283
284                         //variables to get indices of blobs
285                         clusterXIndex = cx * clusterSize;
286                         clusterYIndex = cy * clusterSize;
287
288                         //this loop cycles through the clusters individual blobs
289                         blobFilamentDifferential = Vector3.zero;
290
291                         //First the blobs should be convected with the blobs inside
                             ↪   their own cluster
292                         for (int ix = 0; ix < clusterSize; ix++)
293                         {
294                             for (int iy = 0; iy < clusterSize; iy++)
295                             {
296
297                                 //This ensures the convected blob doesnt convect itself
298                                 if ( ix != bx && iy != by)
299                                 {
300
301                                     //Convect the blob!
302                                     blobDistance =
                                         ↪   Vector3.Distance(vortonPositions[clusterXIndex
                                         ↪   + bx, clusterYIndex + by],
                                         ↪   vortonPositions[clusterXIndex + ix,
                                         ↪   clusterYIndex + iy]);
303                                     blobVelocityDirection =
                                         ↪   Vector3.Cross((vortonPositions[clusterXIndex +
                                         ↪   ix, clusterYIndex + iy] -
                                         ↪   vortonPositions[clusterXIndex + bx,
                                         ↪   clusterYIndex + by]).normalized,
                                         ↪   blobInitialVelocity[clusterXIndex + ix,
                                         ↪   clusterYIndex + iy]);
304                                     blobVelocityMagnitude = blobVorticity[clusterXIndex
                                         ↪   + ix, clusterYIndex + iy] / (2.0f * Mathf.PI *
                                         ↪   blobDistance);
305                                     blobFilamentDifferential += blobVelocityMagnitude *
                                         ↪   blobVelocityDirection;
```

```
306
307                                 }
308
309                             }
310                         }
311
312                         //Now its been convected with the blobs in its cluster, it
                             ↪  needs to be convected by the clusters them selves
313                         for ( int ccx = 0; ccx <(gridRowsInt/clusterSize); ccx++)
314                         {
315                             for ( int ccy = 0; ccy <(gridColumnsInt/clusterSize);
                                 ↪  ccy++)
316                             {
317                                 if ( cx != ccx && cy != ccy)
318                                 {
319
320                                     //this loop cycles through the other clusters
321                                     blobDistance =
                                         ↪  Vector3.Distance(vortonPositions[clusterXIndex
                                         ↪  + bx, clusterYIndex +
                                         ↪  by],clusterPositionCoef[ccx,ccy]);
322                                     blobVelocityDirection =
                                         ↪  Vector3.Cross((clusterPositionCoef[ccx, ccy] -
                                         ↪  vortonPositions[clusterXIndex + bx,
                                         ↪  clusterYIndex + by]).normalized,
                                         ↪  clusterInitialVelocityCoef[ccx,ccy]);
323                                     blobVelocityMagnitude =
                                         ↪  clusterVorticityCoef[ccx,ccy]/ (2.0f *
                                         ↪  Mathf.PI * blobDistance);
324                                     blobFilamentDifferential += blobVelocityMagnitude *
                                         ↪  blobVelocityDirection;
325
326                                 }
327                             }
328                         }
329
330                         //This math solves discretized and solves the velocity time
                             ↪  relationship, but checks for the first row to keep it
                             ↪  bound
331                         if ((clusterXIndex + bx) > 0)
332                         {
333
334                             blobVelocity[clusterXIndex + bx, clusterYIndex + by] =
                                 ↪  blobInitialVelocity[clusterXIndex + bx, clusterYIndex
                                 ↪  + by] + blobFilamentDifferential;
335                             vortonPositions[clusterXIndex + bx, clusterYIndex + by] +=
                                 ↪  blobVelocity[clusterXIndex + bx, clusterYIndex + by] *
                                 ↪  Time.deltaTime;
336
337                         }
338                     }
339                 }
340
341             }
342         }
343
344
345     }
346
347     //This function handles the removal and creation of new blobs
348     void handleGrid()
349     {
350
```

```
351         //check whether grid needs to be updates
352         if (Time.realtimeSinceStartup > (lastTime + spawnTimer))
353         {
354 
355             //Reset te timer
356             lastTime = Time.realtimeSinceStartup;
357             //Debug.Log("I should spawn now");
358             //Debug.Log(clusterPositionCoef[2, 2]);
359             //Debug.Log(clusterVorticityCoef[2, 2]);
360 
361             //Remove blobs that shouldnt be alive yet
362             //for ( int i = 1; i <)
363 
364             //Sub routine to shift rows, this automatically deletes the last rows
365             for (int x = gridRowsInt-2; x > 0; x--)
366             {
367                 for (int z = 0; z < gridColumnsInt; z++)
368                 {
369 
370                     vortonPositions[x + 1, z] = vortonPositions[x, z];
371                     blobVorticity[x + 1, z] = blobVorticity[x, z];
372                     blobVelocity[x + 1, z] = blobVelocity[x, z];
373                     blobInitialVelocity[x + 1, z] = blobInitialVelocity[x, z];
374                     //vortonPositions[x + 1, z] = vortonPositions[x, z];
375                     //vortonPositions[x + 1, z] = vortonPositions[x, z];
376 
377                 }
378             }
379 
380             //creat new blobs
381             for (int y =  0; y < gridRowsInt; y++)
382             {
383                 vortonPositions[1, y] = new Vector3(0.0f , 0.1f, y * gridSpacingInt);
384                 blobVorticity[1, y] = -0.04f * Mathf.Pow((y - (gridRowsInt / 2)), 3);
385 
386                 //biasedly creat vorticity
387 
388 
389                     blobInitialVelocity[1, y] = new Vector3(18.5f, 0, 0.0f);
390                 blobVelocity[1, y] = new Vector3(18.5f, 0.0f, 0.0f);
391             }
392 
393         }
394 
395     }
396 
397     //This function draws gizmos so that each blob can be visualised
398     private void OnDrawGizmos()
399     {
400 
401         Gizmos.color = Color.red;
402         for (int x = 0; x < gridRowsInt; x++)
403         {
404             for (int y = 0; y < gridColumnsInt; y++)
405             {
406                 Gizmos.DrawSphere(vortonPositions[x, y], 0.1f);     // this shows
                    ↪  control points as red spheres in the edit window
407             }
408         }
409 
410     }
411 
412 }
```