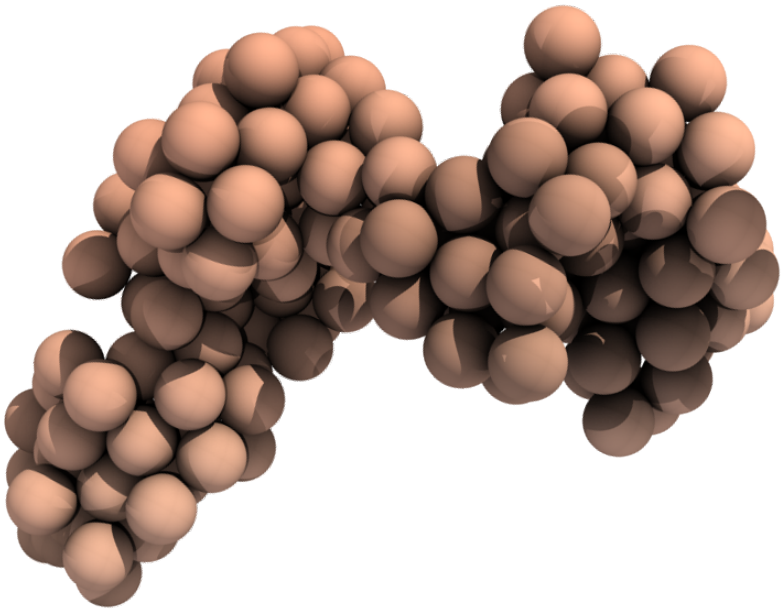
GPU Enabled Analysis of Agglomeration in Large Particle Populations

Elijah Andrews

Supervised by Professor John Shrimpton

April 24, 2018

Word count: 9981



This report is submitted in partial fulfillment of the requirements for the MEng Aeronautics and Astronautics, Faculty of Engineering and the Environment, University of Southampton

Declaration

I, Elijah Andrews declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

I confirm that:
1. This work was done wholly or mainly while in candidature for a degree at this University;
2. Where any part of this thesis has previously been submitted for any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. None of this work has been published before submission.

Acknowledgements

Many thanks to my supervisor, Professor John Shrimpton, for his support and guidance during this project.

Thanks also to my father, Mark Andrews, for assistance in testing and benchmarking the project on different computer systems.

Contents

Abstract

Simulation is an important tool in modern day engineering. It allows engineers to test designs and predict behaviours without requiring expensive experimental testing. Simulations can often be time consuming and require hours or days to complete in order to achieve acceptable accuracy. One way to improve performance without simply adding more computational power is to use Graphics Processing Units (GPUs). Unlike CPUs, GPUs run computations in parallel which can make simulations a lot faster if the same calculations are run many times. This project implements the Discrete Element Method using GPUs in order to analyse particle agglomeration.

A simple particle simulation is developed with Python in order to understand the methodologies required for the project. A GPU based particle simulation is developed using OpenCL, capable of running simulations with large numbers of particles (tested up to 10^7 particles). This simulation is then used to observe how agglomerates form with varying simulation properties. The analysis provides a strong basis for future analysis of agglomerate formation in Taylor-Green Vortex flow, showing general system behaviour and key areas for future analysis.

Nomenclature

δ	Particle Body Surface Overlap Distance
δ_e	Particle Effect Surface Overlap Distance
η	Damping Coefficient
$\hat{\mathbf{n}}$	Collision Normal Unit Vector
$\hat{\mathbf{t}}$	Collision Tangent Unit Vector
$\mathbf{F}$	Force
$\mathbf{F}_c$	Cohesion Force
$\mathbf{F}_d$	Drag Force
$\mathbf{F}_g$	Gravitational Force
$\mathbf{F}_n$	Normal Contact Force
$\mathbf{F}_t$	Tangential Contact Force
$\mathbf{g}$	Gravitational Acceleration Vector
$\mathbf{u}$	Particle Velocity Vector
$\mathbf{u}_f$	Fluid Velocity Vector
$\mathbf{u}_n$	Relative Velocity Normal to Collision
$\mathbf{u}_t$	Relative Velocity Tangential to Collision
$\mathbf{x}$	Particle Position Vector
μ	Coefficient of Friction
μ	Dynamic Viscosity
μ	Friction Coefficient
ρ_p	Particle Density
τ_p	Particle Relaxation Time
ζ	Tangential Displacement During Interaction

a	Acceleration
d_b	Particle Body Surface Diameter
d_e	Particle Effect Surface Diameter
k_c	Cohesion Stiffness
k_e	Normal Collision Stiffness
k_f	Friction Stiffness
m	Mass
N	Number of Particles
O_p	Particle Origin
s_b	Particle Body Surface
s_e	Particle Effect Surface
t	Time
u	Particle Speed
u_0	Initial Particle Speed

List of Abbreviations

GPU	Graphics Processing Unit
CPU	Central Processing Unit
DEM	Discrete Element Method
OpenCL	Open Computing Language

Chapter 1

Introduction

1.1 Project Overview

Simulation is an important tool in modern day engineering. It allows engineers to test designs and predict behaviours without requiring expensive experimental testing. Simulation is in widespread use in many key areas of engineering such as fluid dynamics and structural design. Simulations can often be time consuming and require hours or days to complete in order to achieve acceptable accuracy. It is common to construct super-computers in order to achieve better performance and faster simulations. However, this is often not economical or widely available. One way to improve performance without simply adding more computational power is to use Graphics Processing Units (GPUs). Unlike CPUs, GPUs run computations in parallel which can make simulations a lot faster if the same calculations are run many times.

Methods used to simulate systems vary and there are many programs that offer different tools for different jobs. The method used in this project is the Discrete Element Method, a way of simulating the motion and interactions of particles. This project implements the Discrete Element Method using GPUs in order to analyse particle agglomeration.

1.2 Aims and Objectives

The aim of this project is to observe how agglomerates form with varying simulation properties using a particle simulation developed with OpenCL. This can be separated into three main objectives:

1. Develop a simple particle simulation in order to understand the methodologies required for the project.

2. Develop a particle simulation that can simulate large numbers of particles (10^7) using OpenCL for GPUs.

3. Simulate particles in a fluid with collisions and observe how agglomerates form, including statistical analysis of agglomerate properties, as different simulation properties are varied.

1.3 Report Structure

This report has three main parts. Firstly, the model being used and its numerical implementation are discussed. Chapter 2 provides detail on the mathematical model and the analytic equations of motion for the model. Chapter 3 shows how the mathematical model is implemented numerically and has some comparisons of method accuracy.

The second part explains how the model is implemented programmatically. Chapter 4 discusses the initial implementation of the algorithm in Python. Chapter 5 contains a background in Graphics Processing Units and how OpenCL functions. It also discusses the details of the final OpenCL implementation. Both of these chapters contain relevant simulation verification test cases.

The final part is the application of the simulation to the study of how agglomerates form with varying particle properties and initial conditions. This demonstrates the capabilities of the simulation and provides some useful insights.

1.4 Previous Work

The project directly preceding this one was 'Programming GPU Cards with OpenCL to Predict the Motion of Billions of Particles'[1] by Andrew Chow. His project developed a parallelised particle-fluid simulator using OpenCL. Particle-particle interactions were not considered in his project. This project will expand upon his by implementing particle-particle interactions using the Discrete Element Method (explained in Chapter 2).

The DEM has been implemented many times since it was originally devised by Cundall in 1971[2]. Many implementations have been CPU based with no parallelisation, but in recent years implementations have tended to be parallelised. The vast majority of parallel DEM implementations[3][4][5][6][7] have used NVIDIA's CUDA platform which can only run on NVIDIA GPUs. These implementations are not usable on other hardware. One solution to this problem is to use the Open Computing Language (OpenCL). OpenCL code can be executed across heterogeneous platforms. This means that an implementation programmed in OpenCL can be accessible to most users. One paper described an implementation that did use OpenCL but the application was for a real-time interactive simulation and so relatively low numbers of particles were used (16,000 was the maximum benchmarked)[8]. Another paper briefly describes an adaptation of a simple existing implementation and its performance, however the details are not extensive and testing was only done with 2^{17} (131,072) particles[9].

This project draws from the work of Rob Tuley whose PhD thesis[10] outlined some key aspects of the DEM as well as its application in powder simulations.

Chapter 2

The Discrete Element Method

The Discrete Element Method (DEM) is a numerical method for simulating how particles move and interact. Individual particles of a medium are treated as separate rather than making continuum assumptions. This makes it a good method for modelling behaviours in granular materials such as sand, grain, or powder.

There are two main categories of particle simulation: soft and hard models. Soft models allow overlap and treat collisions as sustained events whereas hard models treat collisions as an instantaneous event with no overlap and model forces as an impulse. Soft collision models have broader applicability as they can model sustained contact and multiple simultaneous collisions[11].

2.1 Particle Definition

The DEM can be used with arbitrary polyhedra, however for simplicity this project will only consider spherical particles as defined in figure 2.1 where O_p is the particle origin, s_b is the body surface, and s_e is the effect surface. The diameters of the particle body surface and particle effect surface are denoted by d_b and d_e, respectively.

The body surface is the surface of the particle considered to be the physical boundary, if two particles' body surfaces are touching or overlapping they are considered to be in contact. The effect surface is the surface of a particle within which cohesion effects are considered. If two particles' effect surfaces are touching or overlapping they are considered to be interacting.

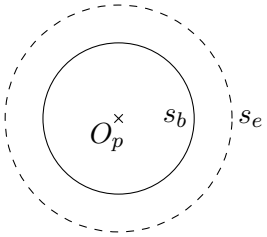


Figure 2.1: Definition of a particle.

2.2 DEM Forces

The Discrete Element Method can simulate a number of different forces using a variety of models. The merits of some of the most common models are discussed in Tuley[10]. For this project the simplest force models have been chosen to reduce the overall complexity of the simulation.

2.2.1 Normal Contact Force

In a real elastic collision there will be some deformation of the particles. Calculating the deformation itself would be computationally expensive and would not be of interest in the study of particle population behaviours. The interaction can be modelled as a linear spring-dashpot arrangement where the overlap between the two particles is the compression of the spring. The damping is based on the relative velocity in the normal direction. The force is thus described by Equation 2.1 where k_e is the normal contact stiffness, δ is the particle overlap, $\hat{\mathbf{n}}$ is the unit vector normal to the collision, η is the damping coefficient, and $\mathbf{u}_n$ is the normal velocity.

$$\mathbf{F}_n = k_e \delta \hat{\mathbf{n}} - \eta \mathbf{u}_n \tag{2.1}$$

2.2.2 Tangential Contact Force

The tangential contact force is the friction between two particle surfaces. There are two regimes of friction force, static and dynamic. In the static regime there is no tangential motion and the friction acts to stop motion. In the dynamic regime two surfaces are sliding across one another and the friction acts to arrest this motion. The static regime usually has a higher friction coefficient than the dynamic regime. The simplest and least computationally expensive model for friction is a 'complex friction model'. This calculates values for both of the regimes and applies the minimum of the two calculations.[10]

Tuley[10] mentions two common static friction models, one without damping and one with damping. These are shown in Equation 2.2 and Equation 2.3 where k_f is the friction stiffness, ζ is the tangential displacement during the interaction, $\hat{\mathbf{t}}$ is the unit vector tangential to the collision, η is the damping coefficient, and $\mathbf{u}_t$ is the velocity tangential to the collision. The merit of these models is discussed in section 3.2.

$$\mathbf{F}_t^{static} = -k_f \zeta \hat{\mathbf{t}} \tag{2.2}$$

$$\mathbf{F}_t^{static} = -k_f \zeta \hat{\mathbf{t}} - \eta \mathbf{u}_t \tag{2.3}$$

The dynamic regime friction force is calculated with Equation 2.4 where μ is the coefficient of friction.

$$\mathbf{F}_t^{dynamic} = -\mu |\mathbf{F}_n| \hat{\mathbf{t}} \tag{2.4}$$

The final tangential friction force is defined in Equation 2.5.

$$\mathbf{F}_t = -\hat{\mathbf{t}} min(|\mathbf{F}_t^{static}|, |\mathbf{F}_t^{dynamic}|) \tag{2.5}$$

2.2.3 Cohesion Force

Cohesion is the attractive force between two bodies of the same material, adhesion is the attractive force between two bodies of different materials. For this project all of the particles and walls are assumed to be of the same material and so only cohesion is considered, however adhesion could be modelled using varying cohesion stiffnesses. Although there are many complex effects that could be considered[10], a basic linear approximation can be used to model a cohesion force. This is defined in Equation 2.6 where k_c is the cohesion stiffness and δ_e is the particle effect surface overlap.

$$\mathbf{F}_c = k_c \delta_e \hat{\mathbf{n}} \tag{2.6}$$

2.2.4 Drag Force

Assuming low Reynolds numbers, Stokes's flow relationships can be used. This includes Stokes's drag, defined in Equation 2.7, where m is particle mass, τ_p is particle relaxation time, $\mathbf{u}_f$ is fluid velocity, and $\mathbf{u}$ is particle velocity. Stokes's drag applies to spheres moving through a very low Reynolds number fluid. In this situation the drag force is approximately proportional to velocity.

$$\mathbf{F}_d = \frac{m}{\tau_p}(\mathbf{u}_f - \mathbf{u}) \tag{2.7}$$

The particle relaxation time, τ_p is an approximate timescale describing how the particle's velocity changes in a fluid due to drag. Assuming Stokes's drag on a spherical particle, τ_p can be defined by Equation 2.8 where ρ_p is the particle density, d_b is the particle body diameter, and μ is the dynamic viscosity of the fluid.

$$\tau_p = \frac{\rho_p d_b^2}{18\mu} \tag{2.8}$$

2.2.5 Gravitational Force

The gravity force is simple and defined in Equation 2.9.

$$\mathbf{F}_g = m\mathbf{g} \tag{2.9}$$

2.3 Equations of Motion

The motion of particles in the simulation is governed by Equation 2.11. This is derived from Newton's Second Law of motion. The total force, $\mathbf{F}$ is a combination of forces as shown in Equation 2.12. $\mathbf{F}_n$, $\mathbf{F}_t$, $\mathbf{F}_c$, $\mathbf{F}_g$, and $\mathbf{F}_d$ are the forces defined in section 2.2.

$$\frac{d\mathbf{x}}{dt} = \mathbf{u} \tag{2.10}$$

$$\frac{d\mathbf{u}}{dt} = \frac{\mathbf{F}}{m} \tag{2.11}$$

$$\mathbf{F} = \mathbf{F}_n + \mathbf{F}_t + \mathbf{F}_c + \mathbf{F}_g + \mathbf{F}_d \tag{2.12}$$

2.4 Useful Results

2.4.1 Reduced Mass

When considering the motion of two particles relative to each other a useful property is the reduced mass. The reduced mass is an equivalent combined mass of two particles when calculating relative motion between them. Equation 2.13 shows how the reduced mass is used instead of a single particle mass when relating the force in a collision to the relative acceleration between the particles.

The reduced mass is derived from the relative acceleration of two particles in a collision. To determine the relative acceleration between the two particles, the two accelerations must be combined. It is known that F_1 and F_2 are equal and opposite. Finding the relative acceleration results in Equation 2.13. It is of a similar form as Newton's Second Law but with the usual mass replaced with the reduced mass of the two particles (Equation 2.14). This reduced mass can be used instead of particle mass in Equation 2.11 when considering relative motion of particles.

$$
\begin{aligned}
F_1 &= m_1 a_1,\ F_2 = m_2 a_2 \\
F_1 &= -F_2 \\
a_{rel} &= a_2 - a_1 \\
&= \frac{F_2}{m_2} - \frac{F_1}{m_1} \\
&= \frac{F_2}{m_2} + \frac{F_2}{m_1} \\
&= F_2\Big(\frac{m_1 + m_2}{m_1 m_2}\Big) \\
F_2 &= \Big(\frac{m_1 m_2}{m_1 + m_2}\Big) a_{rel} \qquad (2.13) \\
m_{reduced} &= \frac{m_1 m_2}{m_1 + m_2} \qquad (2.14)
\end{aligned}
$$

It should be noted that Equation 2.14 reduces to m_1 if m_2 tends to infinity. This is useful when considering static particles with infinite density and is used extensively for verifying simulation results (see section 3.3).

2.4.2 Collision Duration

The collision duration is the time it takes for a collision to occur. This is important because if the simulation timestep is not low enough there will not be enough steps within a collision to produce accurate results. The collision duration is fully derived in Section A.1.1 and is calculated using Equation 2.15. The collision duration is often simplified to Equation 2.16.

$$t_{col} = \sqrt{\frac{m}{k_e}}\sqrt{\pi^2 + ln(\epsilon)^2} \qquad (2.15)$$

$$t_{col} = \pi\sqrt{\frac{m}{k_e}} \qquad (2.16)$$

2.4.3 Coefficient of Restitution

The coefficient of restitution is the ratio of speed before and after a collision. For a damped collision the coefficient of restitution is between 0 and 1. The coefficient of restitution for a normal collision is fully derived in Section A.1.2 and is related to the damping coefficient, η, by Equation 2.17.

$$\eta = -2ln(\epsilon)\sqrt{\frac{mk_e}{\pi^2 + ln(\epsilon)^2}} \tag{2.17}$$

2.5 Particle Rotation

For arbitrary polyhedral particles rotation can be important since the particles will collide at different angles and it can make a significant difference to the result. With spherical particles this is less important since the collision geometry is always the same. Spherical particles can rearrange more easily if rotation is considered but reasonably accurate results can still be obtained without implementing rotation. The aim of this project is to analyse how agglomerates form with a change in simulation properties so having fully accurate results is not necessary as long as the changes in behaviour can be captured.

2.6 Collision Detection

A key part of the DEM is effective collision detection. If approached naively collision detection is simply calculating the overlap for every particle with every other particle. This is extremely inefficient and causes the simulation to run in $O(N^2)$ time, where N is the number of particles. To improve efficiency this process can be split into two phases. Firstly, the broad phase collision detection determines which particles *could* collide with which other particles. This reduces the number of particle collisions that have to be resolved. The second phase, collision resolution, resolves the collisions by measuring overlap and then calculating forces if necessary. The goal of the broad phase is to allow the simulation to run in $O(N)$ time. Figure 2.2 shows the basic simulation loop and the different simulation phases.

There are many different algorithms that can be used for broad phase collision detection, varying in complexity and efficiency. Although some of the more complex algorithms were considered, the gain in efficiency was not significant enough to outweigh the significant increase in implementation complexity. For this reason the simplest algorithm, spatial zoning, has been chosen for this project.

Spatial zoning separates the simulation domain into control volumes. Particles are then sorted into the control volumes. Particles in neighbouring, or the same, control volumes are then resolved fully. It is most efficient to have the control volumes as small as possible because this reduces the number of particles in neighbouring control volumes. Control volumes must be at least as large as the largest particle in the simulation to ensure that neighbouring control volumes contain all possible collisions. For monodisperse particle populations this means that the control volumes should be the same size as the particles. For polydisperse

particle populations this means that the control volumes should be the size of the largest particle in the population. As mentioned in Tuley[10] this can decrease efficiency for particle populations with a large range of sizes. Sections 4.2.2 and 5.3.2 provide further detail as to how this is implemented computationally in Python and with OpenCL, respectively.

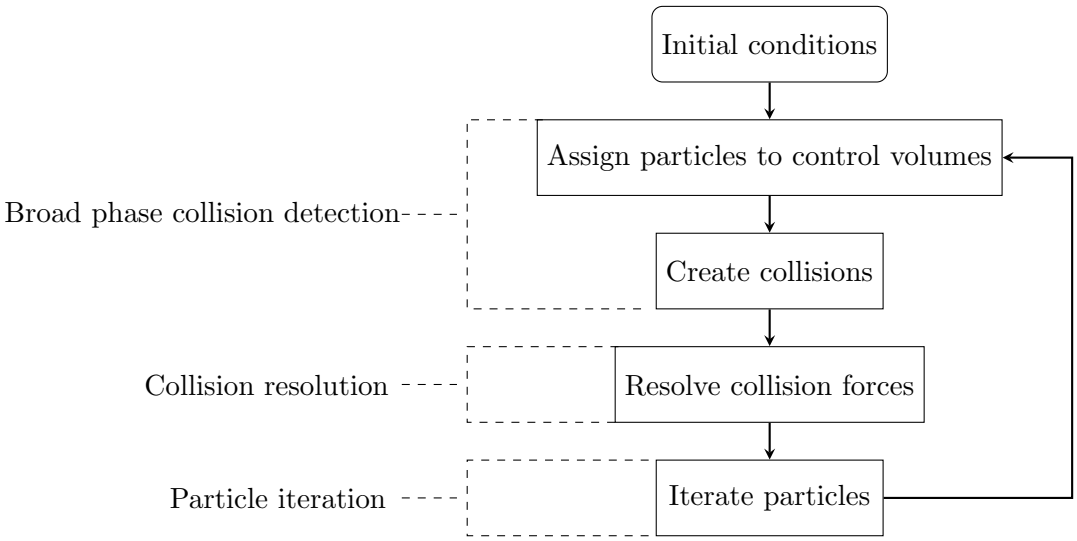


Figure 2.2: The basic simulation loop and simulation phases.

Chapter 3

Numerical Methods

3.1 Numerical Integration Schemes

The DEM model used in this project has stiff governing equations. This means that the simulation may become unstable if the timestep is not sufficiently small. Thus, the choice of integration scheme is important. For simplicity, there are three main schemes that will be considered for this project: the Euler method (or forward Euler method), the backward Euler method (or implicit Euler method), and the trapezoidal rule.

The forward Euler method is an explicit integration method, using the current value to estimate the next value. It has first order accuracy and requires only the data from the current iteration.

The backward Euler method is an implicit integration method, assuming the next value and solving the equation for it. It also has first order accuracy and requires only the data from the current iteration.

The trapezoidal rule is an implicit integration method that uses the average of the current and next values. It has second order accuracy.

For the implicit methods either an analytic solution to the implicit equations or a numerical solution can be used. For simplicity, only the functions that can form analytic solutions from the implicit equations will use the implicit methods.

3.1.1 Velocity

For Equation 2.11 the forward Euler method for integrating acceleration to get velocity at iteration $n+1$ is shown in Equation 3.1. In this case the force, F, is a function of u_n.

$$u_{n+1} = u_n + \frac{F(u_n)}{m}\Delta t \tag{3.1}$$

For the same equation, the backward Euler method is shown in Equation 3.2. In this case

the force, F, is a function of u_{n+1}.

$$u_{n+1} = u_n + \frac{F(u_{n+1})}{m}\Delta t \tag{3.2}$$

For the DEM the total force is a combination of forces (see Equation 2.12). Drag and gravity forces are only calculated once per iteration and so can be calculated at the same time as the velocity. However, collision forces are calculated multiple times per iteration to get contributions from all collisions and so cannot be easily calculated at the same time as the velocity. For this reason, the normal force, tangential force, and cohesion force are treated as fixed at the time of velocity calculation. Gravity is also fixed so Equation 2.12 can be represented as in Equations 3.3 and 3.5. $F(u)$ can then be used in Equation 3.1 or 3.2.

$$F(u) = F_n + F_t + F_c + F_g + F_d(u) \tag{3.3}$$

$$F_{fixed} = F_n + F_t + F_c + F_g \tag{3.4}$$

$$F(u) = F_{fixed} + F_d(u) \tag{3.5}$$

For Equation 3.1 it is trivial to calculate the next velocity as the RHS is only dependant on u_n which is known. This results in Equation 3.6.

$$u_{n+1} = u_n + \frac{F_{fixed} + F_d(u_n)}{m}\Delta t \tag{3.6}$$

Equation 3.2 must be rearranged to get u_{n+1} as a function of u_n only. This results in Equation 3.7.

$$\begin{aligned}
F(u_{n+1}) &= F_{fixed} + \frac{m}{\tau_p}(u_f - u_{n+1}) \\
u_{n+1} &= u_n + \frac{F_{fixed} + \frac{m}{\tau_p}(u_f - u_{n+1})}{m}\Delta t \\
&= u_n + \frac{F_{fixed}}{m}\Delta t + \frac{\Delta t}{\tau}(u_f - u_{n+1}) \\
u_{n+1}(1 + \frac{\Delta t}{\tau}) &= u_n + \frac{F_{fixed}}{m}\Delta t + \frac{\Delta t}{\tau}(u_f) \\
u_{n+1} &= \frac{\tau u_n + F_{fixed}\tau\Delta t/m + u_f\Delta t}{\tau + \Delta t}
\end{aligned} \tag{3.7}$$

Equation 3.7 can be rearranged to be in the same form as Equation 3.6 as shown in Equation 3.8.

$$u_{n+1} = u_n + \Big(\frac{u_f - u_n + \tau F_{fixed}/m}{\tau + \Delta t}\Big)\Delta t \tag{3.8}$$

A comparison of the accuracy of these integration schemes is in section 3.1.3.

3.1.2 Position

Equation 2.10 is a very simple differential equation that can be solved by integrating. Since the velocity does not directly depend on the position, and the velocity for the next iteration is calculated before the position, it is trivial to use the trapezoidal rule to calculate position. It has all of the advantages of accuracy without the disadvantage of increased complexity. The position can thus be calculated with Equation 3.9.

$$x_{n+1} = x_n + \frac{u_n + u_{n+1}}{2}\Delta t \tag{3.9}$$

3.1.3 Comparison of Integration Schemes

This analysis was performed using the Python implementation discussed in Chapter 4.

Figure 3.1 shows the speed of a particle in a fluid starting from rest and accelerating to the fluid velocity. It shows that the explicit velocity equation (Equation 3.6) over-estimates the speed and the implicit velocity equation (Equation 3.7) under-estimates the speed. The error in the implicit result is less than the error in the explicit result.

Figure 3.2 shows the average percentage difference between each numerical method and the analytic solution for varying timestep. This shows that the explicit method is exactly first order accurate whereas the implicit result is slightly higher than first order accurate. This means that it is better to use the implicit method than the explicit method.

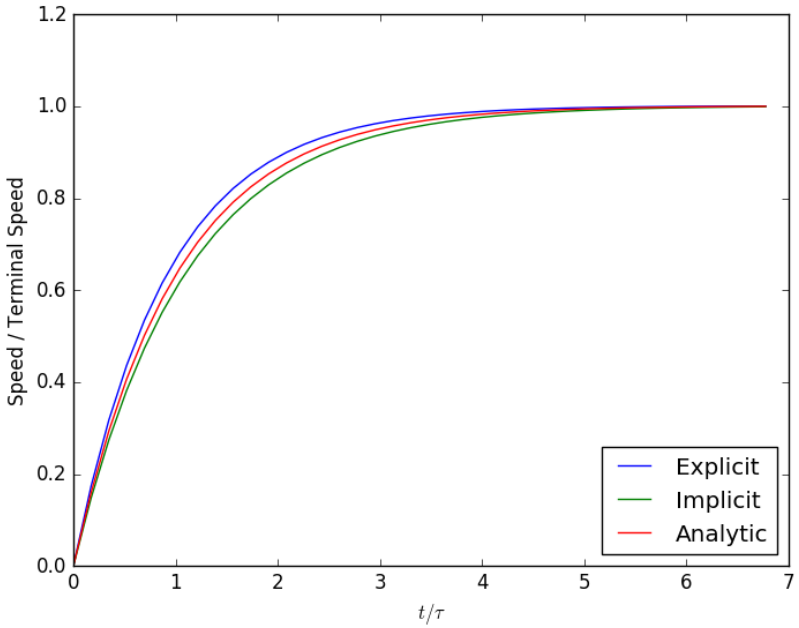


Figure 3.1: Particle speed against time with different integration schemes.

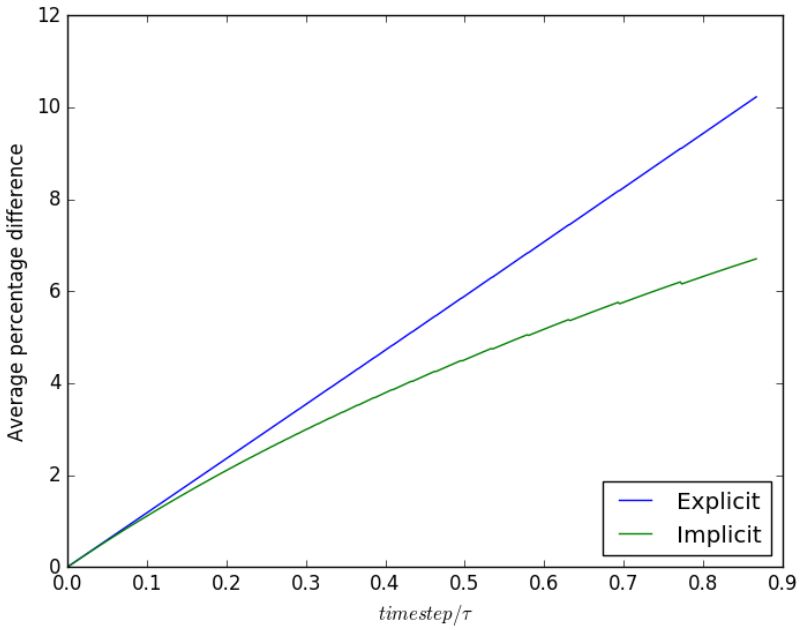


Figure 3.2: Average percentage difference between the numerical method and analytical solution against varying timestep. The jaggedness of the lines is due to the discrete changes in timestep causing small rounding errors.

3.2 Friction Model

The static friction models outlined in Section 2.2.2 both rely on the tangential displacement during an interaction, ζ. This is a difficult property to determine as it requires a collision history to be maintained for it to be correctly measured. Maintaining a collision history adds significant complication to a simulation as it requires data to be stored between timesteps. As this implementation of the DEM ignores rotation, the accuracy of the simulation when considering sliding is limited. Increasing the simulation complexity in order to slightly improve the friction model is not necessary so various alternatives have been considered. The graphs presented below use the friction verification case from Section 3.3.3.

3.2.1 Dynamic Friction Only

Removing the static friction part of the model entirely is tempting, it would simplify the calculation and make the code slightly faster. However, the dynamic friction model is prone to instability, especially at high timesteps, the particle ends up oscillating around a point because the model does not handle overshoot well. This behaviour is shown in Figure 3.3.

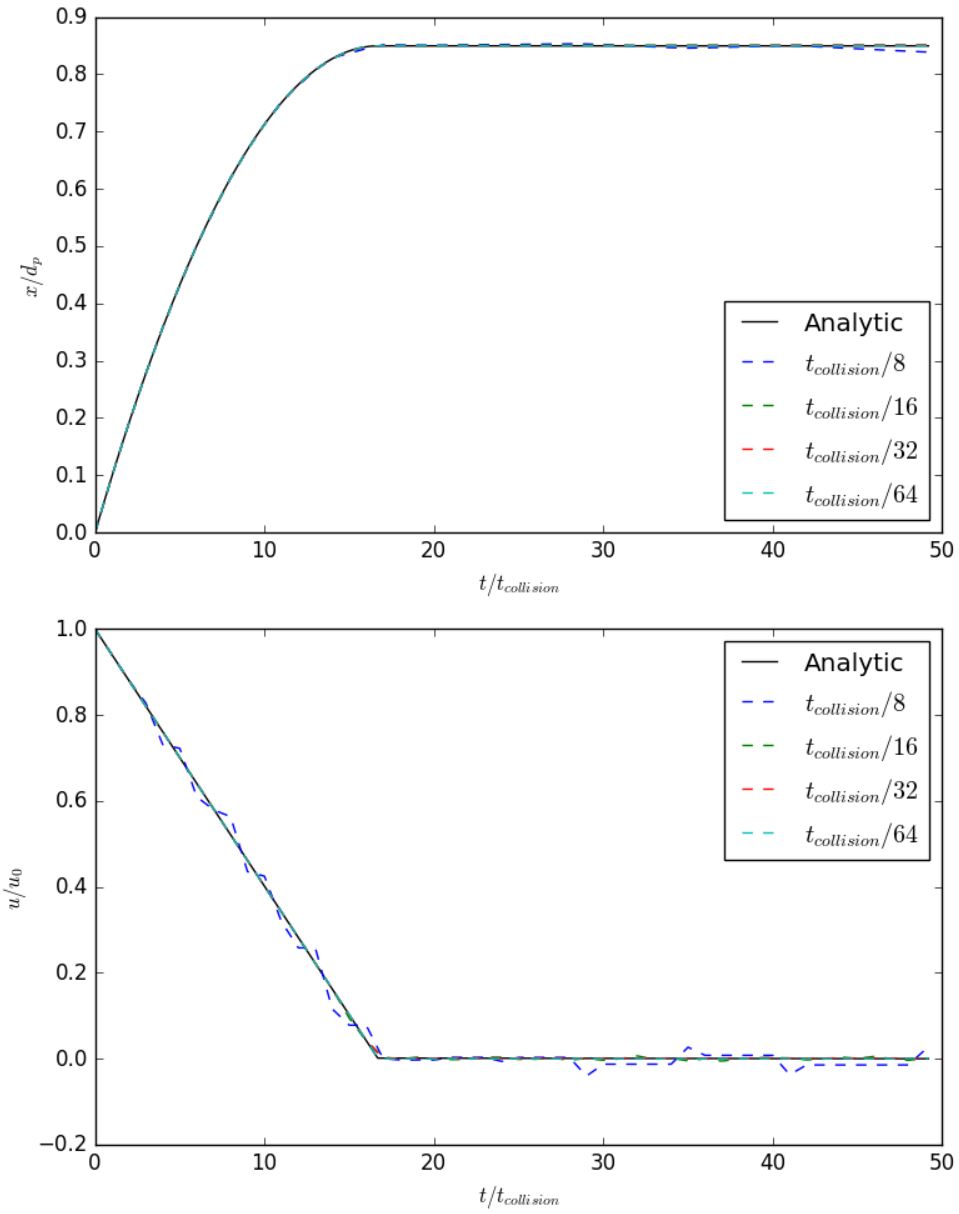


Figure 3.3: Position and velocity against time for a dynamic only friction model. Note high levels of instability, especially in the velocity.

3.2.2 Static Friction Without Damping

Two methods were considered for calculating the tangential displacement, ζ. The first was to simply estimate how far the particle moved in a timestep by multiplying the tangential velocity by the timestep. This method does not work because as the timestep decreases the static friction decreases and the error increases. This effect is demonstrated in Figure 3.4.

The alternative method was to estimate ζ by multiplying the tangential velocity by the collision duration (see Section 2.4.2). The results for this model are shown in Figure 3.5. This provides far more accurate results and solves both the instability problem from the dynamic model and the underestimation problem from the timestep based static friction model.

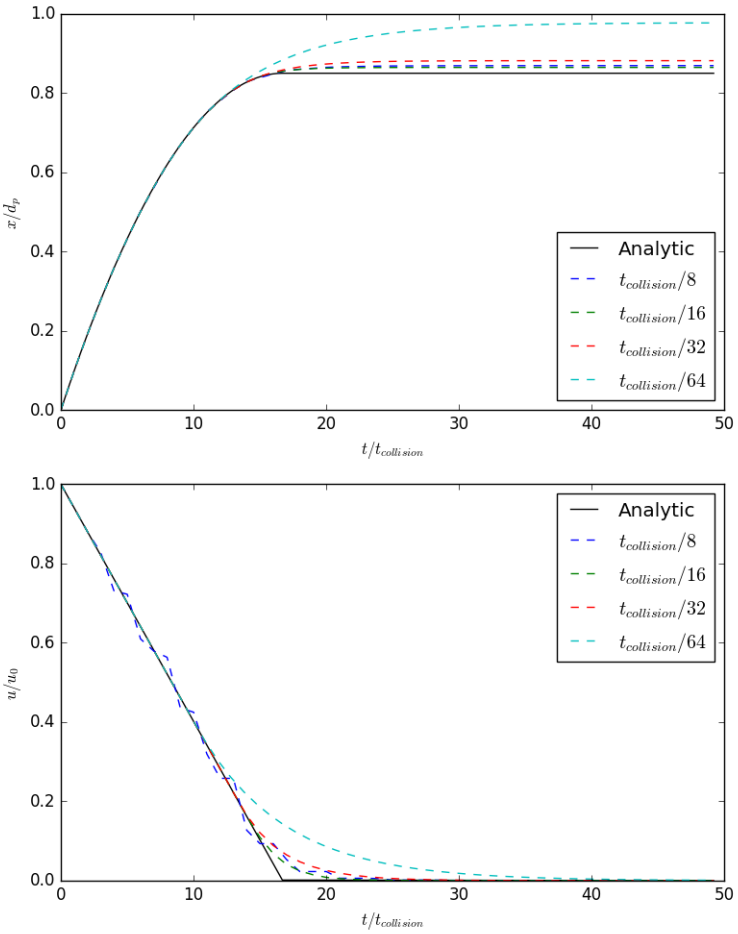


Figure 3.4: Position and velocity against time for a timestep based static model. Note the increasing error with decreasing timestep.

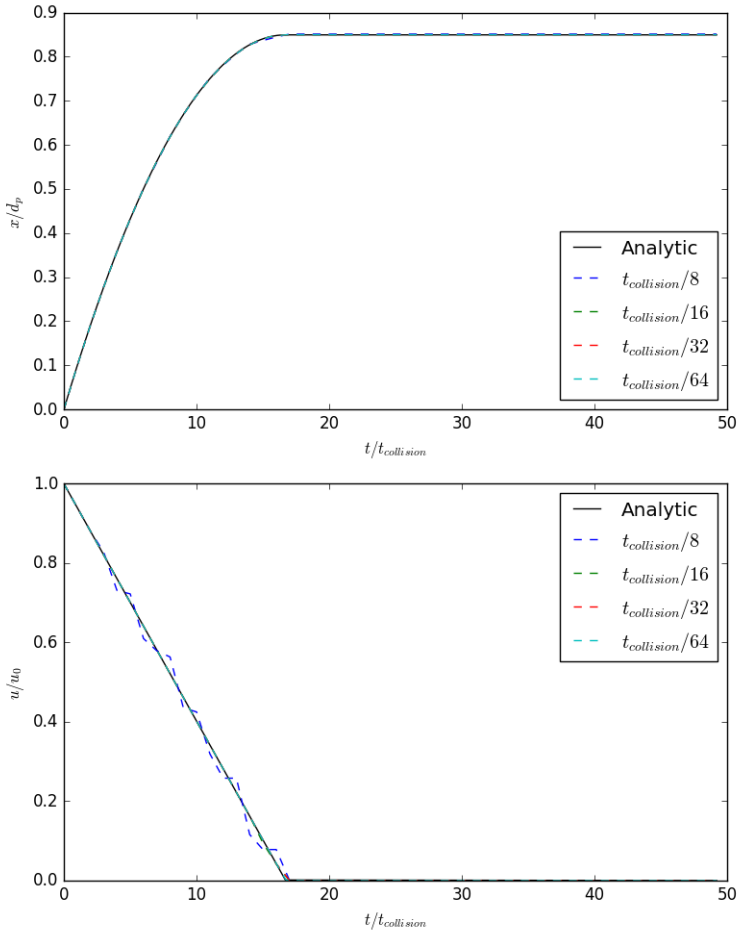


Figure 3.5: Position and velocity against time for a collision duration based static model.

3.2.3 Static Friction With Damping

The static friction model with damping from Equation 2.3 was investigated but did not provide any significant improvement in accuracy or stability over the static model without damping.

3.3 Verification

A series of verification test cases have been developed in order to assess the accuracy of the numerical model in the program implementations. The definitions and equations for these models are in this section and the implementation comparisons are in Sections 4.3 and 5.4.

3.3.1 Drag

The drag verification case is a particle in a fluid. The fluid has a constant velocity in the x direction denoted by u_f. There is no gravity and no other particles are present. The particle accelerates to the speed of the fluid.
The position and velocity of the particle are determined by Equations 3.10 and 3.11, respectively. These equations are derived in Section A.5.

$$x = u_f\tau(e^{-t/\tau} - 1) + u_f t \tag{3.10}$$

$$\dot{x} = u_f(1 - e^{-t/\tau}) \tag{3.11}$$

$$\tag{3.12}$$

3.3.2 Normal Collision

The normal collision verification case is the most simple normal collision. The first particle (p1) has infinite density and so is fixed in space. The second particle (p2) has an initial velocity, u_0, towards the first particle and starts at $x = d_b$ so that the body surfaces are just touching. There is no drag, no gravity, and no cohesion. This arrangement is depicted in Figure 3.6.
The position and velocity of the second particle are determined by Equations 3.13 and 3.14, respectively. These equations are derived in section A.1.

$$x = e^{at}\frac{u_0}{b}sin(bt) + d_b \tag{3.13}$$

$$\dot{x} = u_0 e^{at}(\frac{a}{b}sin(bt) + cos(bt)) \tag{3.14}$$

$$\text{Where: } a = \frac{-\eta}{2m},\ b = \frac{\sqrt{4mk_e - \eta^2}}{2m}$$

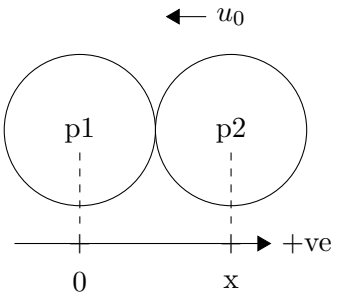


Figure 3.6: The initial setup of the normal collision verification case.

3.3.3 Friction Sliding

The friction sliding test case is a particle sliding along a wall. The particle is started at the theoretical height at which the normal force balances the gravitational force. This ensures that the friction force is as stable as possible and not affected by changing normal force. The particle also has an initial velocity. This arrangement is depicted in Figure 3.7. The analytic solution to this case uses the dynamic friction only but is sufficient to assess the accuracy of the model. The position and velocity of the particle are determined by Equations 3.16 and 3.15, respectively.

$$\dot{x} = -\mu g t + u_0 \tag{3.15}$$

$$x = \frac{-\mu g}{2} t^2 + u_0 t + x_0 \tag{3.16}$$

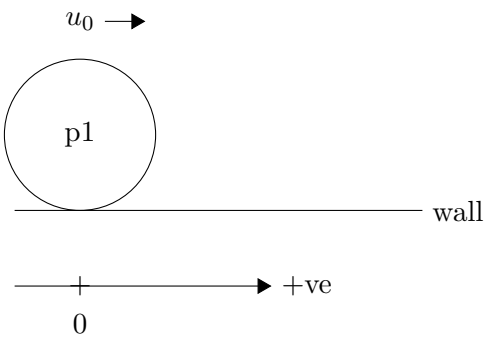


Figure 3.7: The initial setup of the friction sliding verification case.

3.3.4 Normal Collision with Cohesion

The cohesion collision verification case is very similar to the normal collision verification case (section 3.3.2). The first particle (p1) has infinite density and so is fixed in space. The second particle (p2) has an initial velocity, u_0, towards the first particle and starts at $x = d_e$ so that the effect surfaces are just touching. There is no drag and no gravity. This arrangement is depicted in Figure 3.8.

The position and velocity of the second particle are determined by the set of equations

below. These equations are derived in section A.2.

When $d_e > x > d_b, u < 0$:

$$x = u_0\sqrt{\frac{m}{k_c}}sinh\Big(t\sqrt{\frac{k_c}{m}}\Big) + d_e \tag{3.17}$$

$$\dot{x} = u_0 cosh\Big(t\sqrt{\frac{k_c}{m}}\Big) \tag{3.18}$$

When $x < d_b$:

$$x = e^{at}(\frac{u_i - ac}{b}sin(bt) + ccos(bt)) + \frac{k_e d_b - k_c d_e}{k_e - k_c} \tag{3.19}$$

$$\dot{x} = e^{at}\Big(\Big(\frac{u_i - ac}{b}a - cb\Big)sin(bt) + u_i cos(bt)\Big) \tag{3.20}$$

When $d_e > x > d_b, u > 0$:

$$x = (d_b - d_e)cosh\Big(\sqrt{\frac{k_c}{m}}t\Big) + u_r\sqrt{\frac{m}{k_c}}sinh\Big(\sqrt{\frac{k_c}{m}}t\Big) + d_e \tag{3.21}$$

$$\dot{x} = \sqrt{\frac{k_c}{m}}(d_b - d_e)sinh\Big(\sqrt{\frac{k_c}{m}}t\Big) + u_r cosh\Big(\sqrt{\frac{k_c}{m}}t\Big) + d_e \tag{3.22}$$

Where: $a = \dfrac{-\eta}{2m}$, $b = \dfrac{\sqrt{4mk_e - \eta^2}}{2m}$, $c = \dfrac{k_c(d_e - d_b)}{k_e - k_c}$

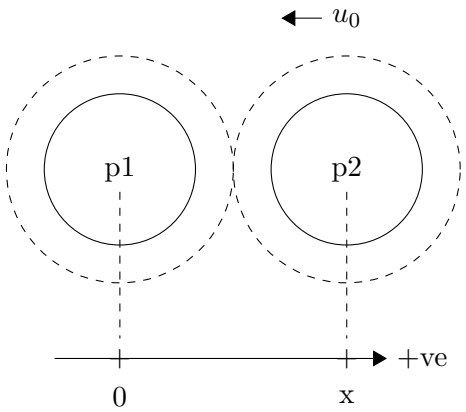


Figure 3.8: The initial setup of the cohesion collision verification case.

Chapter 4

Python Implementation

4.1 Overview

An initial implementation of the DEM has been developed in Python. The objective of this implementation is to gain an understanding of the DEM and any inherent computational difficulties. Python has been chosen as a testing environment for its simplicity and ease of development. The code for the Python implementation can be found in the DEMApples GitHub repository[12].

4.2 Program Structure

4.2.1 Element Types

Different element types are required for different types of geometry and particle. For the Python implementation the two simplest have been chosen, a spherical particle and an axis-aligned wall. Python allows for object oriented programming which makes element tracking easier.

Particle

The basic particle element is a spherical particle with pre-determined properties. All of these properties can be set upon instantiation of each particle object and so can be easily modified for different simulations.
There are two objects for particles, the main object, 'Particle', tracks a full particle state history which is very memory intensive and unnecessary for most applications. The second object, 'LowMemParticle', inherits from 'Particle' and only keeps track of the current state and, during iteration, one future state.

Axis-Aligned Simple Wall

The basic wall element is an axis-aligned simple wall. This object is defined by two points, minimum and maximum, that must lie in the same plane. From them a rectangle is formed. A normal is calculated for the wall and stored in the object to save time in collision calculations. The wall is treated as fixed, eliminating the need for complex material properties

or calculation of motion. An axis-aligned wall has been chosen because it eliminates a lot of the complex calculations required when calculating arbitrary planar geometry. Non-axis-aligned geometry, such as a slope, can still be used in simulations by having gravity act along different vectors. This has been used in 'gravity_shift_closed_box.py' simulation[12].

4.2.2 Collisions

Collision Detection

Broad phase collision detection uses the simple spatial zoning technique from Section 2.6. This approach has been chosen because it is quick and simple to implement. Other options (such as triangulation[13]) were considered for this implementation but the benefits of using them were far outweighed by their complexity. Since the initial Python implementation will not be fast anyway it was not deemed necessary to implement optimised algorithms at this stage.

The domain is represented by a three-dimensional array where each entry is a control volume. The control volume has a list of particles in its bounds. The global list of particles is iterated over and each particle assigns itself to the correct control volume. This results in a three-dimensional array where each control volume has all of the particles within its bounds as an array. Collision objects are then created for each pair of particles in the same, or neighbouring, control volumes. This approach reduces the problem from $O(N^2)$ to almost $O(N)$ as shown in figure 4.1.

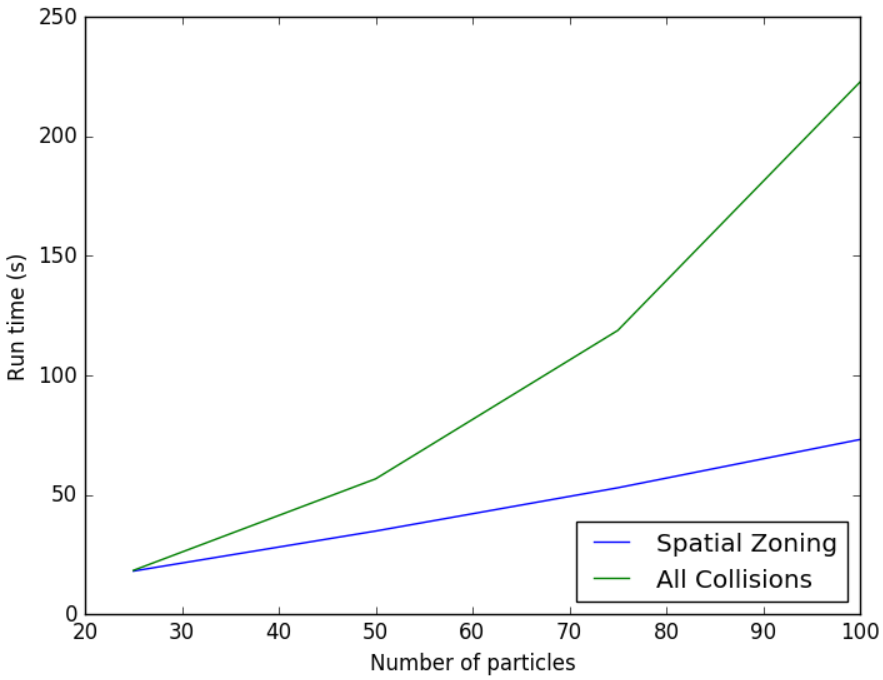


Figure 4.1: This graph shows that the simple spatial zoning technique reduces the problem from $O(N^2)$ down to almost $O(N)$.

Collision Resolution

After an array of collisions has been generated they are iterated over and each collision is resolved. First, the distance between particles is calculated to determine if they are in contact. Often this reveals that they are not in contact and the calculation ends there. If particles are in contact then collision forces are determined.
In the Python implementation only the simple normal and tangential contact forces are calculated. These are enough to run sufficient initial test cases.

4.2.3 Calculating Forces

The mathematics for how the forces are calculated can be found in Section 3. This section discusses some of the implementation details.

Drag

To determine drag a flow velocity must be calculated. In this implementation it is calculated using a function that is passed into the Particle object upon instantiation. This allows a variety of flow field functions to be used without modifying the Particle object code. The default for this function is a perfectly stationary flow.

Gravity

A gravity function can also be passed into the Particle object upon instantiation. Although this defaults to a simple -9.81ms^{-2} in the z direction, it can be chosen to simulate a rotating frame of reference or other complex configurations. A rotating frame of reference has been implemented in the 'gravity_shift_closed_box' example simulation[12].

DEM Forces

The DEM forces that are calculated in collisions are stored in an array within the Particle object. When the particle is iterated the array is summed and used in the iteration calculation. After this calculation the array is cleared so that forces do not get incorrectly added multiple times. This configuration makes it simple to add and remove forces to the simulation whenever necessary and could also be used in general to add any force to the particle.

4.3 Verification

This section contains the results of the verification cases from Section 3.3 for the Python implementation. The graphs in this section compare the results of simulations against the analytic solutions. Overall, the results show that the simulation is suitably accurate and as timestep decreases the simulation accuracy increases, as expected.

4.3.1 Drag

This is the drag verification case from Section 3.3.1. The properties used are in Table 4.1 and the results are shown in Figure 4.2. The results show that the simulation results closely match the analytic solution and that the error decreases as the timestep decreases.

Property	Value
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m
Fluid Viscosity μ	0.00193 Ns/m^2
x Fluid Velocity u_f	1 m/s

Table 4.1: Properties used in the Python drag verification case.

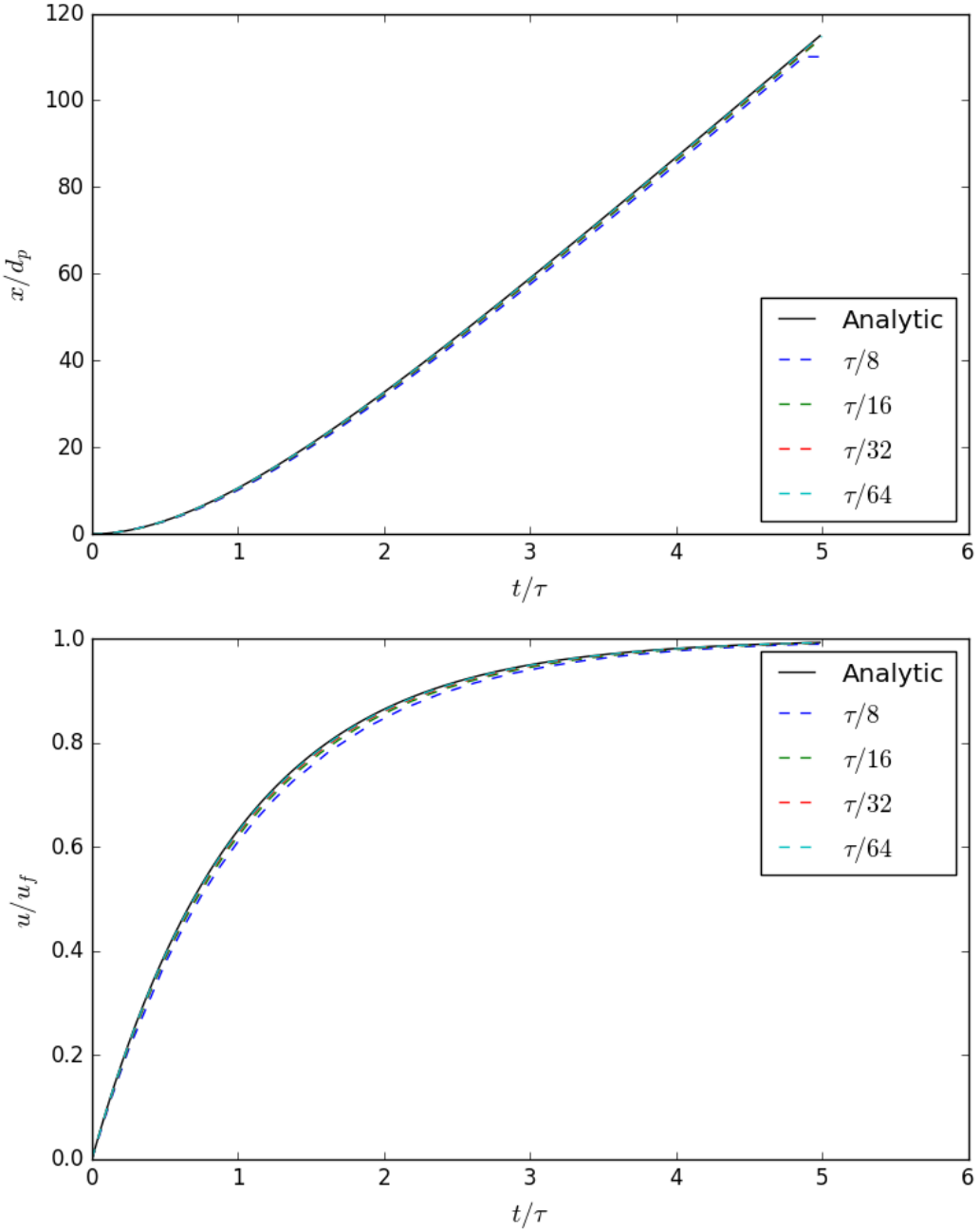


Figure 4.2: Normalized position and speed against time of a particle in a moving fluid. In this graph time is normalized with the particle relaxation time τ, position is normalized with particle diameter d_p, and velocity is normalized with fluid velocity u_f.

4.3.2 Normal Collision

This is the normal collision verification case from Section 3.3.2. The properties used are in Table 4.2 and the results are shown in Figure 4.3. The results show that the simulation results closely match the analytic solution and that the error decreases as the timestep decreases.

Property	Value
Initial x Velocity u_0	-2 m/s
Stiffness k_e	1e5 N/m
Coefficient of Restitution ϵ	0.8
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m

Table 4.2: Properties used in the Python normal collision verification case.

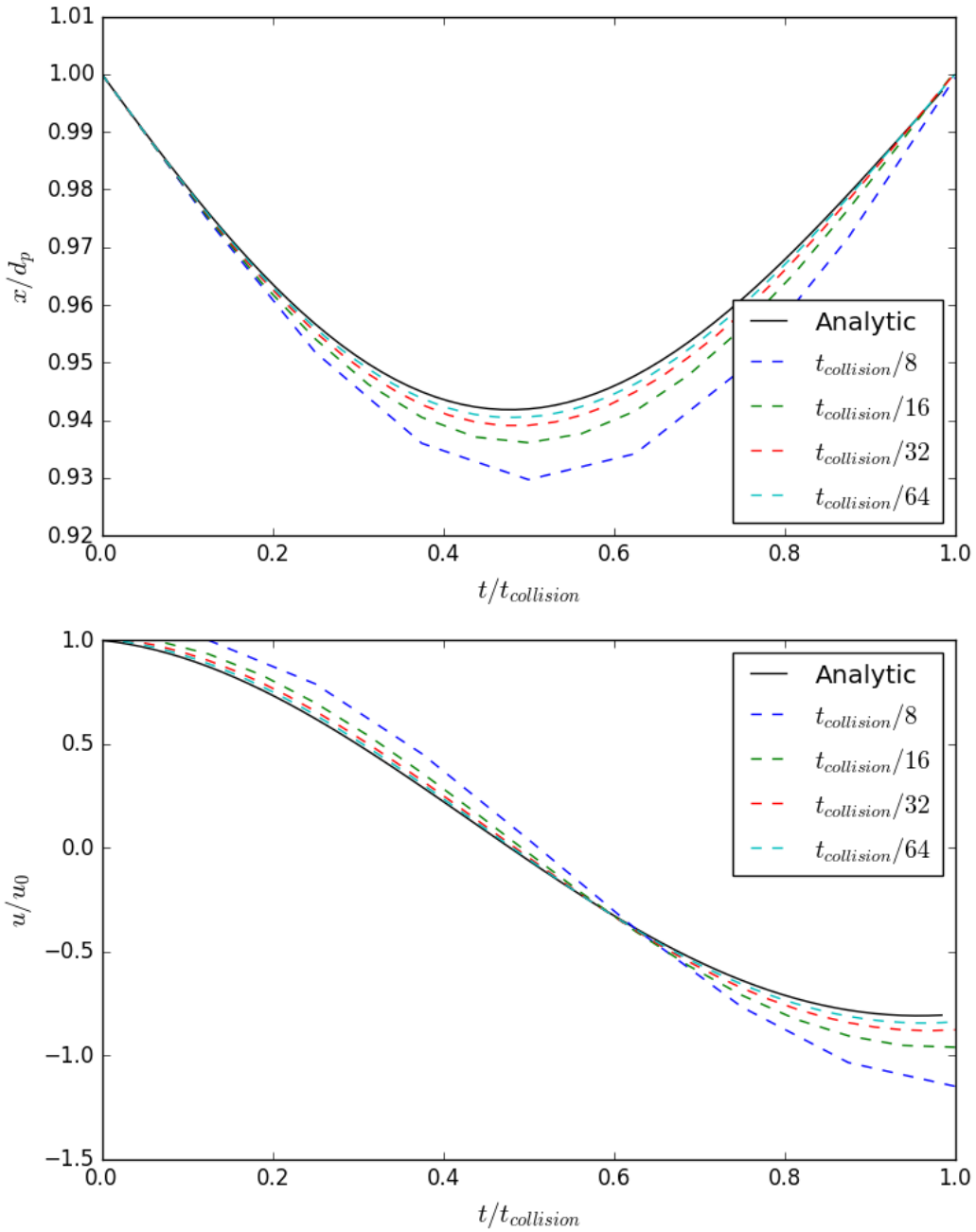


Figure 4.3: Normalized position and speed against time during a normal collision. In this graph time is normalized with the collision duration $t_{collision}$, position is normalized with particle diameter d_p, and velocity is normalized with initial velocity u_0.

4.3.3 Friction Sliding

This is the normal collision verification case from Section 3.3.3. The properties used are in Table 4.3 and the results are shown in Figure 4.4. The results show that the simulation results closely match the analytic solution. The error does decrease as the timestep decreases but there is no clear evidence that it does so in a predictable way.

Property	Value
Initial x Velocity u_0	1 m/s
Initial x Position x_0	0 m
Gravitational Acceleration g	9.81 m/s^2
Stiffness k_e	1e5 N/m
Coefficient of Restitution ϵ	0.8
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m
Coefficient of Friction μ	0.6
Friction Stiffness k_f	1e8 N/m

Table 4.3: Properties used in the Python friction sliding verification case.

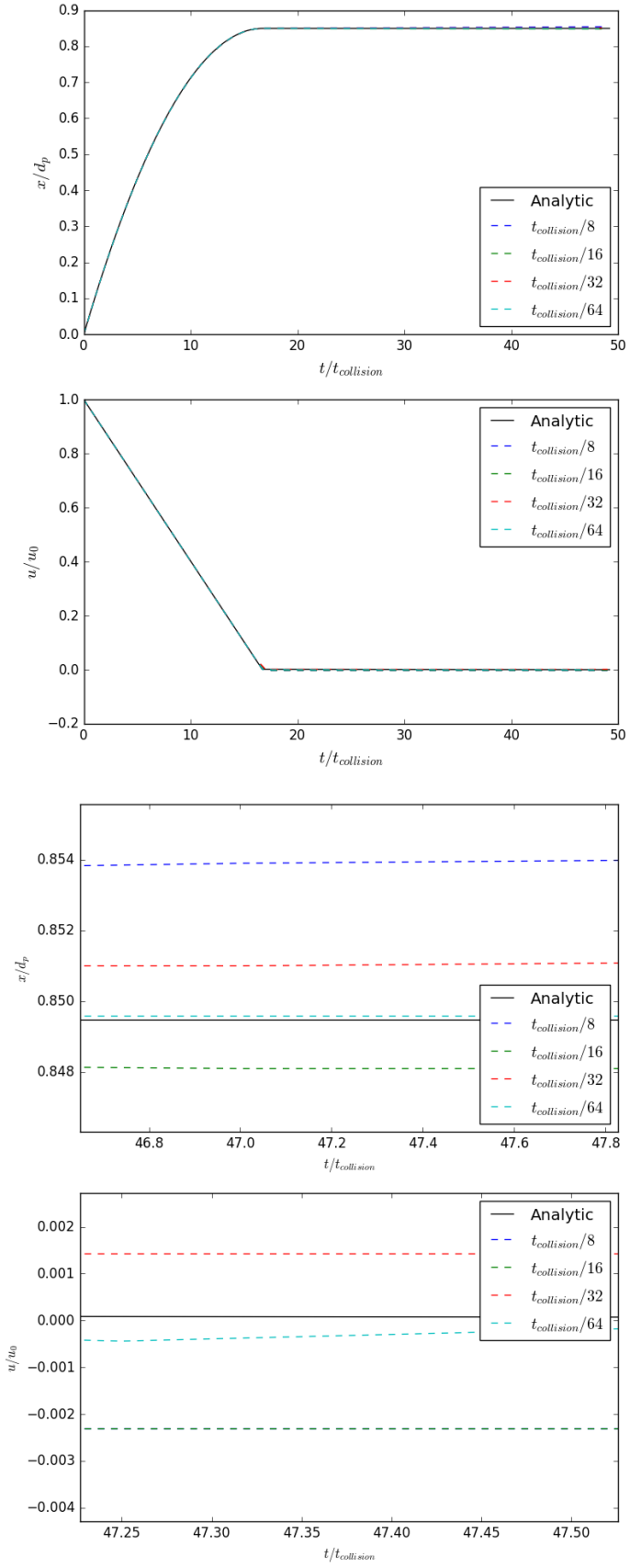


Figure 4.4: Normalized position and speed against time during a normal collision. In this graph time is normalized with the collision duration $t_{collision}$, position is normalized with particle diameter d_p, and velocity is normalized with initial velocity u_0.

Chapter 5

OpenCL Implementation

5.1 OpenCL and Graphics Processing Units

The main computational device in a computer is the Central Processing Unit (CPU). The CPU takes a series of instructions and calculates pretty much anything that a computer requires. For most applications this is ideal as they require a sequence of different instructions to be executed to achieve their goal. However, for some applications this is inefficient. The most common occurrence of this is in graphics. Graphics require the same calculations to be performed a very large number of times to render images. To run this on a CPU would take a long time because it would all run in sequence. To speed up this process Graphics Processing Units (GPUs) were developed. GPUs run the same operation many times in parallel rather than different operations in series. Individual GPU computation cores are slower than CPU cores but for graphics this is fine as the gains from running everything in parallel are massive.

Originally this was applied to graphics but more recently has been used to do scientific calculations where the same operation is repeated across a lot of data. In order to facilitate this languages have been developed to allow a developer to easily run calculations on GPUs. The two most common such languages are CUDA and OpenCL. CUDA is NVIDIA's language that is specifically designed for NVIDIA GPUs. OpenCL is an open standard maintained by the Khronos group, a consortium of companies dedicated to open standard graphics and parallel computing interfaces. OpenCL allows a developer to write programs for a variety of devices (e.g. GPUs, CPUs, and FPGAs) without having to modify the code. This makes OpenCL code highly portable and reusable. For this reason OpenCL has been chosen for use in this project.

There are two main parts of an OpenCL program, the host code and the device code. The host is whatever computer system is being used, usually a CPU, some memory, a storage device, and other standard elements. The device is whatever processing unit is being used to run the main calculations. In this project the device is the GPU. The host runs setup, memory handling, and other miscellaneous processing tasks. The device is then passed kernels to run on a group of data.

5.2 Overview

The main DEM implementation used in this project uses OpenCL to run on a GPU. The host code is written in C to simplify writing code for both the host and the device. More features are available in C++, however only OpenCL 2.1 has C++ kernel support and NVIDIA only supports up to OpenCL 1.2. Since one goal of using OpenCL is to allow the code to be used on multiple platforms, it does not make sense to use OpenCL 2.1.
A lot of the OpenCL utility functions used in this implementation are based on code from the previous project[1]. These functions allow for easy implementation of code without having to repeat long OpenCL function calls.

The code for the OpenCL implementation can be found in the DEMOranges GitHub repository[14]. Technical documentation can be found in the DEMOranges GitHub wiki[15].

5.3 Program Structure

5.3.1 Data Structures

Unlike Python, C does not support objects. This means that more care must be taken in data storage. The most sensible way to store data is in structures. In this implementation structures are used as the primary storage method. There are three types of structures used: particle, wall, collision.

Particle

The particle structure contains particle properties (density, diameter, position, velocity etc.) as well as the fluid viscosity to make it easier to access in calculations.
The structure is aligned to the nearest 128 bytes of memory to make access to it faster. This does waste a little under half of this memory but for 10^7 particles the particle array requires a total of 1.2GB which is within workable limits. Benchmarks could be performed to determine whether this trade-off is necessary, but the downside is small and so has not been considered significant. Another benefit of this alignment is that additional particle data can be stored within the particle array without costing any more memory than is already used.

Wall

The 'aa_wall' structure simply contains the minimum, maximum, and normal vector for an axis aligned wall. This structure is not aligned to the nearest 128 bytes of memory.

Collision

There are two collision structures: 'pp_collision' for particle-particle collisions and 'pw_collision' for particle-wall collisions. The structures only contain the two relevant IDs. The collision structures are not aligned to the nearest 128 bytes of memory as this would waste almost all of the memory and there can be many collisions.

The only notable difference between the particle-particle collision structure and the particle-wall collision structure is that the particle-particle collision structure contains data that tracks whether a collision is across a periodic boundary.

Buffers

To access the data from the device it must be passed into a buffer. None of the structure data is passed into the buffer so the device must have a copy of the definition of the structure. This is problematic as the host and device have different compilers. To work around this problem the host and device structures are written with members in descending order of size. This encourages the compiler to order them correctly in memory.

In addition, when using the MSVC compiler, padding must be added to ensure correct alignment. This padding is not required when using gcc compilers.

The alignment attribute specifier is also not the same between different compilers so "if defined" statements are implemented for both MSVC and gcc compilers to allow the code to be compiled on either without modification.

5.3.2 Kernels

The main calculations for this implementation are performed on the device. This means that the program must be separated into kernels to be passed over sets of data. There are three main sets of kernels: collision detection, collision resolution, and particle iteration.

Collision Detection

To improve efficiency, in both speed and resource usage, performing naive collision detection is not viable for large numbers of particles. To improve on this the spatial zoning technique is used, similar to the initial Python implementation. However, C does not make arrays of varying sizes easy or efficient so the data structures used and the algorithm implementation must be significantly different.

The basic problem is how to store control volumes as lists of references to particles. In Python this was easy, a simple 3D array of control volumes with lists of particle objects inside was sufficient. Various approaches to solving this problem were considered. One approach was to encode particle IDs (equal to the index of a particle in the particles array) with a hashing function into a single number that could be turned back into particle IDs on the device. However, this approach was infeasible because the numbers would get so large that they could not be stored accurately or efficiently.

The approach used is to have multiple passes of assignment of particles to control volumes. The first pass simply counts how many particles are in each control volume so that appropriately sized arrays can be created before the data must be added. This is stored in a one dimensional array of control volumes represented by integers of how many particles

each contains. From this array another array is created. This array is of all the particle IDs but sorted into control volumes. The control volumes are of lengths defined by the count array and start at indexes stored in a third array. If a control volume has no particles, the start index of the array is set to -1. For the unsigned long data type this overflows to the maximum value (approximately 4.3 billion) which will never be used to index particles. This arrangement of arrays is shown in Figure 5.1.

This approach is somewhat similar to how memory is handled on a computer, but using indexes instead of pointers. For an entirely host-side method an array of pointers to arrays of particles could be used, but this would not be sensible when dealing with device memory as each array would need to be moved to the device before use. Having three arrays that hold all the necessary properties simplifies the memory buffer process significantly. The maths required for turning positions and control volume coordinates into indexes in these arrays are contained in cvUtils.c and kernelUtils.cl for host and device, respectively.

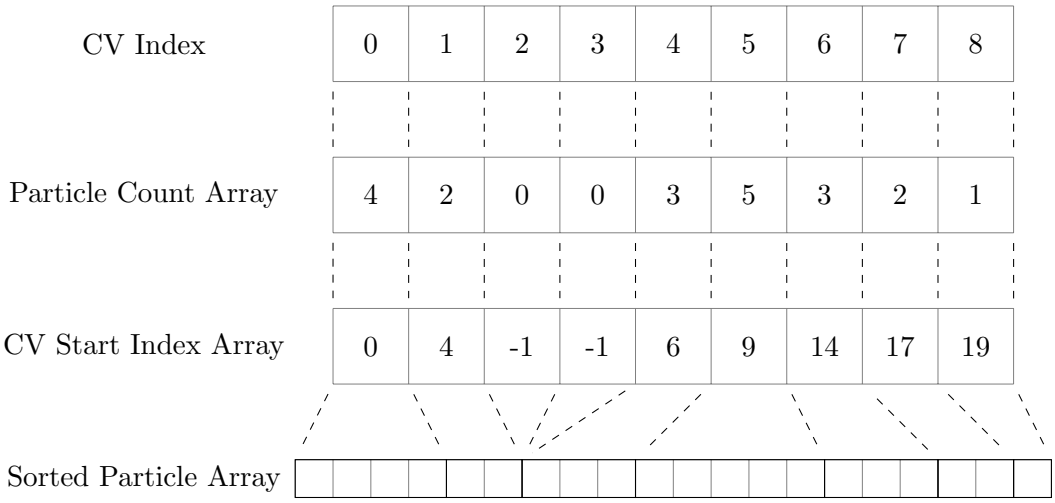


Figure 5.1: Diagram showing the structure and relationship between arrays representing Control Volumes.

One weakness of this approach is that the CV start index array is generated sequentially. This could be done in parallel but would be more complicated and would still need to count in sequence. Although interesting, this not a significant performance problem and so has been left as it is.

Collision Resolution

The collision resolution kernel is actually separated into multiple kernels for different types of collision (particle-particle and particle-wall) however the behaviour of these kernels is almost identical. For this discussion we will use the particle-particle kernel as an example. The kernel takes a pointer to an array of collision structures and a pointer to the array of particles. The DEM collision force calculations are run for each collision and the forces are added to the DEM forces vector in the relevant particle structures. This approach has been chosen because it is easier to sum the forces as they are calculated rather than attempt to predict the length of the necessary force array to store each force separately as in the

Python implementation.

This approach causes a serious problem as it is possible that multiple collision kernels will need to write data to the same particle at the same time. The solution to this is to use the atomic operations available in OpenCL. Unfortunately, OpenCL only natively supports full atomic operations for int and unsigned int data types whereas the forces are stored in a float vector. OpenCL does support an exchange atomic operator for single precision floats, but this is not the best approach for doing atomic arithmetic for floats.

An approach for doing atomic addition (as is necessary in this case) is recommended in an online article[16]. This approach uses the comparison exchange atomic operator by creating a union of the floats with unsigned ints. This works well because the bits being exchanged are the same for the float and unsigned int and actual atomic arithmetic is not necessary so the difference between the data types does not matter. Thus the new value is calculated and if the value used to calculate it has changed in that time the calculation is repeated with the updated value.

Particle Iteration

Particle iteration is performed almost identically to the Python implementation. The main difference, as discussed in section 5.3.2, is that the OpenCL implementation performs the summation of DEM forces as they are calculated in collisions whereas the Python implementation performs the summation when iterating the particle.

5.3.3 Unit Tests

Due to the large size of the project and algorithmic complexity, it is important to test each unit of code individually rather than trying to trace bugs through the whole code-base. In addition, this project is intended to be run on heterogeneous devices and so differences in runtime environment could cause problems. For these reasons a unit testing approach has been chosen so that tests can be quickly re-run to check code unit functionality without assuming identical behaviour between systems.
Some functions are not included in the unit testing system due to their simplicity and the relatively long time it would take to program unit tests for all of them. For example, checking that a function multiplies numbers correctly does not need to be tested every time whereas checking that structures are correctly aligned in memory is important to test every time.

Testing Framework

Often a framework is used for unit testing, however there is not a quick, easy framework available for C with OpenCL so a simple system has been set up to make running tests easy. Each tested feature has a directory within the 'tests' directory with its header and C code files. A feature may have multiple functions, each with its own testing function. Each testing function takes a boolean parameter, 'verbose', that determines whether it prints

intermediate results and debugging outputs. The functions return a boolean that indicates whether the test passed or not. In some cases, if a function fails, it may not be obvious why and so debugging outputs will be printed. For example, test_assign_particle_count could have the wrong number of control volumes or incorrectly assigned particles so both of these outcomes has its own printed debugging output.
To make it easy to run these tests repeatedly 'run_tests.c' has been created to run all of the tests and indicate which, if any, fails. A similar implementation is executed at simulation runtime to ensure that all tests are passed before starting a simulation run.

5.4 Verification

This section contains the results of the verification cases from Section 3.3 for the OpenCL implementation. The graphs in this section compare the results of simulations against the analytic solutions. Overall, the results show that the simulation is suitably accurate and as timestep decreases the simulation accuracy increases, as expected.

5.4.1 Drag

This is the drag verification case from Section 3.3.1. The properties used are in Table 5.1 and the results are shown in Figure 5.2. The results show that the simulation results closely match the analytic solution and that the error decreases as the timestep decreases.

Property	Value
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m
Fluid Viscosity μ	0.00193 Ns/m^2
x Fluid Velocity u_f	1 m/s

Table 5.1: Properties used in the OpenCL drag verification case.

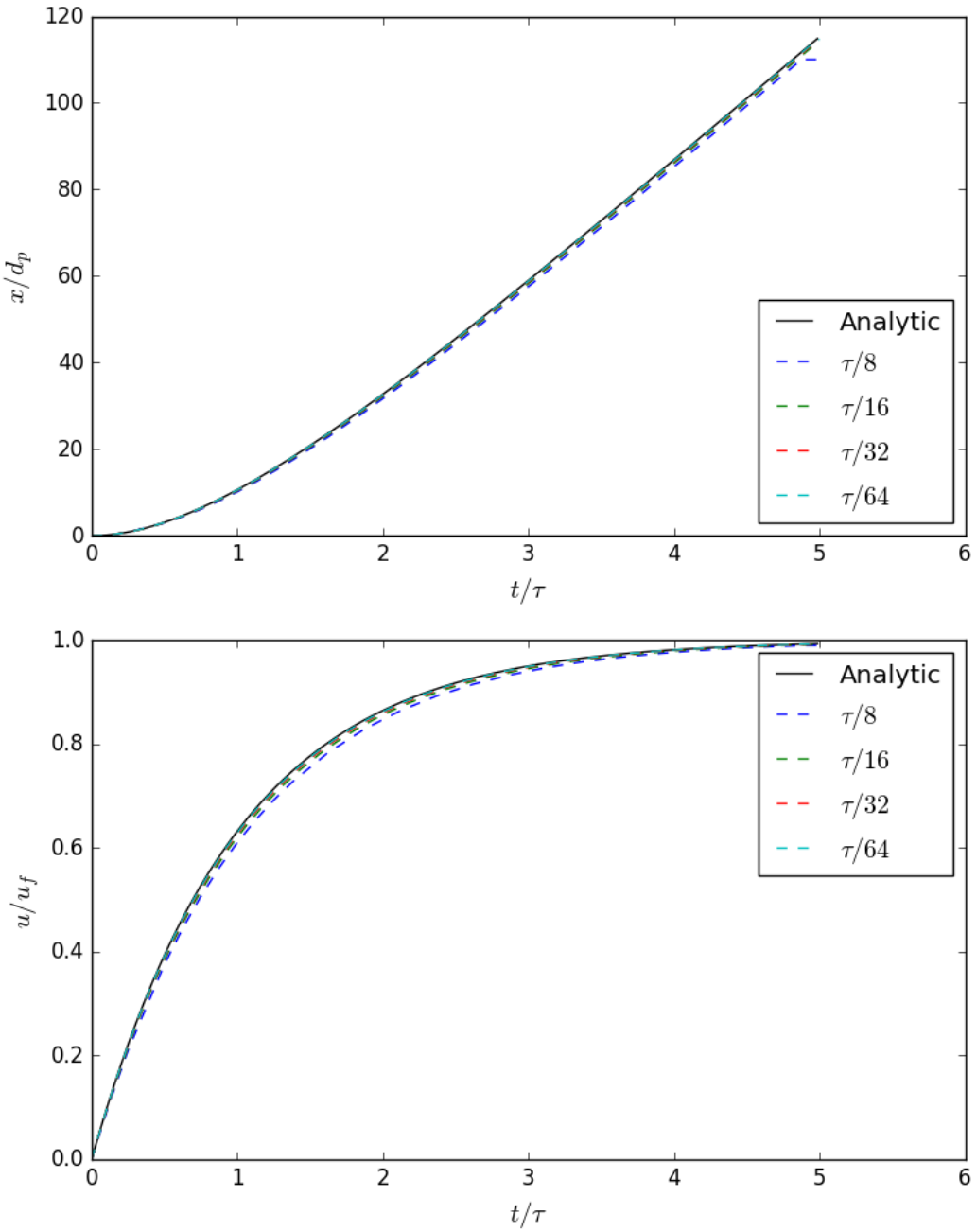


Figure 5.2: Normalized position and speed against time of a particle in a moving fluid. In this graph time is normalized with the particle relaxation time τ, position is normalized with particle diameter d_p, and velocity is normalized with fluid velocity u_f.

5.4.2 Normal Collision

This is the normal collision verification case from Section 3.3.2. The properties used are in Table 5.2 and the results are shown in 5.3. The results show that the simulation results closely match the analytic solution and that the error decreases as the timestep decreases.

Property	Value
Initial x Velocity u_0	-2 m/s
Stiffness k_e	1e5 N/m
Coefficient of Restitution ϵ	0.8
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m

Table 5.2: Properties used in the OpenCL normal collision verification case.

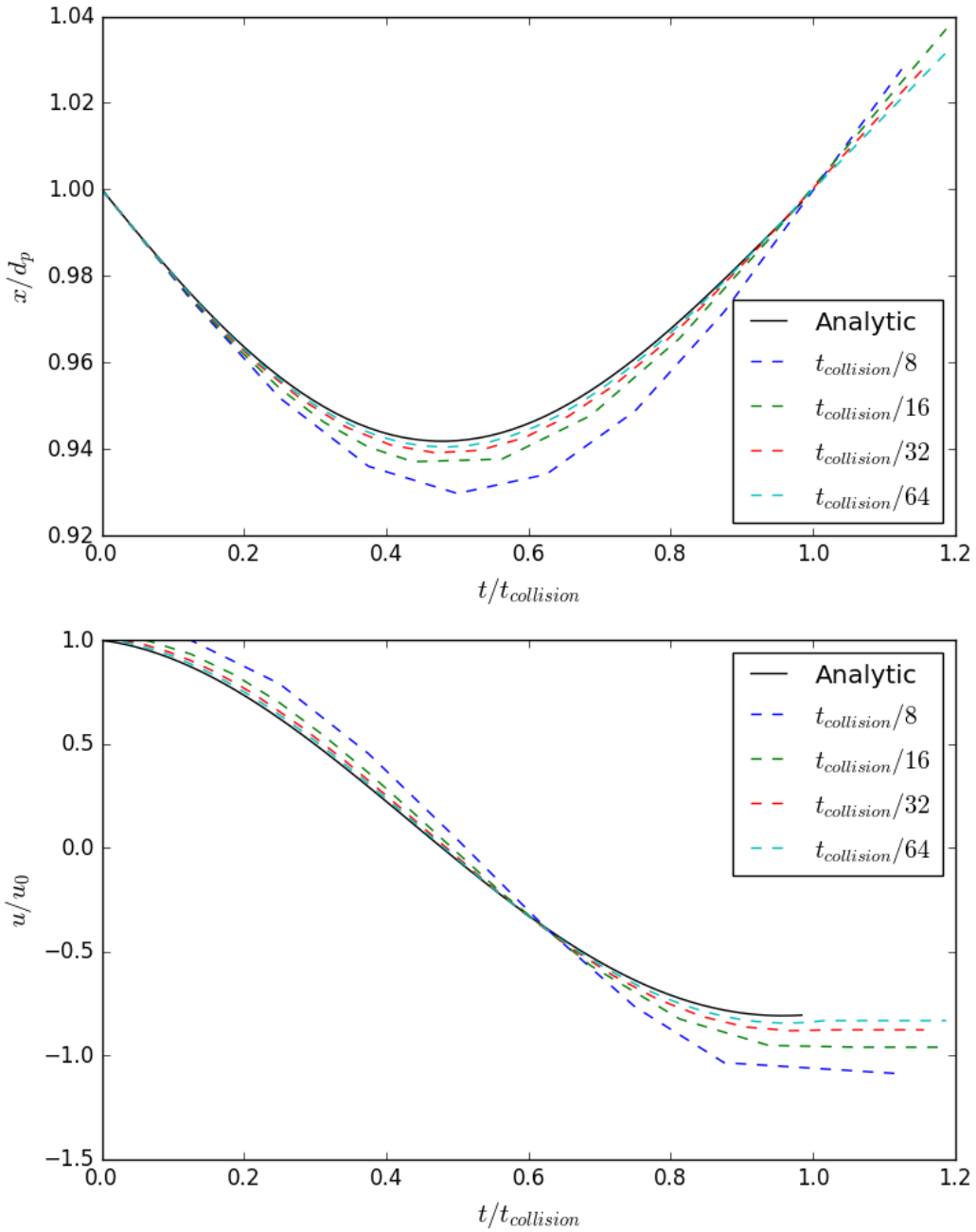


Figure 5.3: Normalized position and speed against time during a normal collision. In this graph time is normalized with the collision duration $t_{collision}$, position is normalized with particle diameter d_p, and velocity is normalized with initial velocity u_0.

5.4.3 Friction Sliding

This is the normal collision verification case from Section 3.3.3. The properties used are in Table 5.3 and the results are shown in Figure 5.4. The results show that the simulation results closely match the analytic solution. The error generally decreases as the timestep decreases but it is unpredictable. The results show that the position error is actually higher for a timestep of $t_{collision}/64$ than for a timestep of $t_{collision}/32$. However, the results are accurate enough for the purpose of this simulation.

Property	Value
Initial x Velocity u_0	1 m/s
Initial x Position x_0	0 m
Gravitational Acceleration g	9.81 m/s^2
Stiffness k_e	1e5 N/m
Coefficient of Restitution ϵ	0.8
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m
Coefficient of Friction μ	0.6
Friction Stiffness k_f	1e5 N/m

Table 5.3: Properties used in the OpenCL friction sliding verification case.

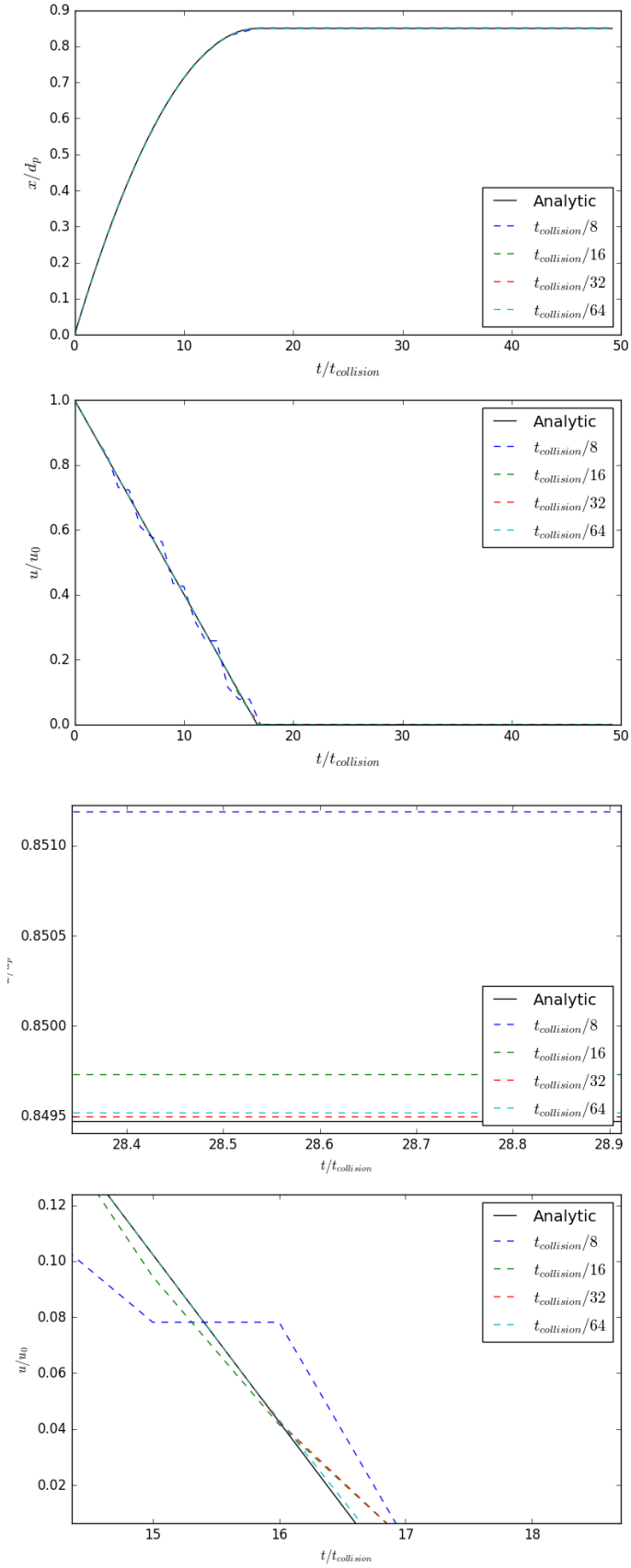


Figure 5.4: Normalized position and speed against time during a friction event. In this graph time is normalized with the collision duration $t_{collision}$, position is normalized with particle diameter d_p, and velocity is normalized with initial velocity u_0.

5.4.4 Normal Collision with Cohesion

This is the normal collision with cohesion verification case from Section 3.3.4. The properties used are in Table 5.4 and the results are shown in 5.3. The results show that the simulation results closely match the analytic solution and that the error decreases as the timestep decreases. Notably, the error causes overshoot on the position so particles are less likely to stick in a simulation than they would in the analytic solution.

Property	Value
Initial x Velocity u_0	-2 m/s
Stiffness k_e	1e5 N/m
Cohesion Stiffness k_c	100 N/m
Coefficient of Restitution ϵ	0.8
Particle Density ρ	10 kg/m^3
Particle Diameter d_b	0.1 m
Particle Effect Diameter d_e	0.15 m

Table 5.4: Properties used in the OpenCL normal collision with cohesion verification case.

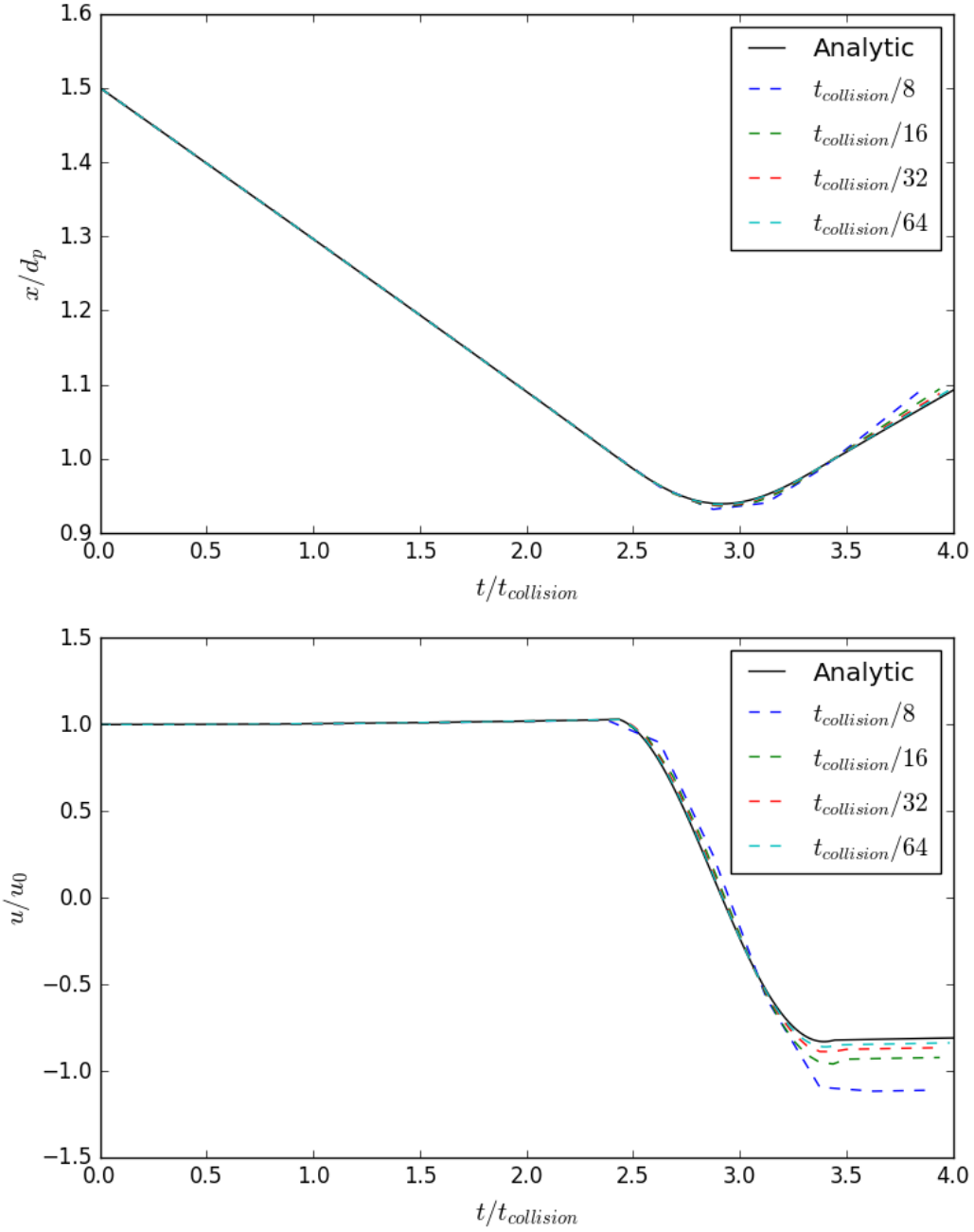


Figure 5.5: Normalized position and speed against time during a normal collision with cohesion. In this graph time is normalized with the collision duration $t_{collision}$, position is normalized with particle diameter d_p, and velocity is normalized with initial velocity u_0.

5.5 Optimization and Performance

5.5.1 Optimizations

Only preliminary optimization has been performed for this implementation as it is not a focus of the project. The optimization efforts have focussed on reducing unnecessary calculations, reducing the number of atomic operations, and avoiding memory transfers between host and device where possible.

5.5.2 Performance

Govender et al. have also developed GPU based DEM implementations and have provided performance measures in one of their papers[17]. The key performance measure they use is the Cundall Number. The Cundall Number is defined in Equation 5.1 where N is the number of particles and FPS is the number of simulation frames that can be calculated in a second. The simulations performed in Govender et al.[17] were run on an NVIDIA Quadro K6000 GPU with an Intel i7 3.5 GHZ Extreme Edition CPU. The benchmarks for this project have been run on an NVIDIA Quadro P5000 GPU with an Intel Xeon 3.5 GHz CPU.

To calculate the C Number of this project a 10 million particle simulation has been run with a timestep of 0.0005 seconds and a simulation length of 5 seconds. This simulation ran in 145.22 minutes yielding an FPS of 1.148. This gives an overall C Number of 11.5×10^6.

Table 5.5 compares this project with Govender et al.[17]. It shows that Govender et al. have achieved significantly better performance. In addition, Govender et al. have improved their performance in a later paper[3]. Although greater performance has been achieved in other codes, this project has achieved satisfactory performance. With further optimization, and implementation of more complex algorithms, it is conceivable that similar performance could be achieved.

$$C = N \times FPS \tag{5.1}$$

Author	Shape	N Particles	C Number
This project	Sphere	10×10^6	11.5×10^6
Govender et al.[17]	Sphere	50×10^6	55×10^6
BLAZE-DEM[3]	Sphere	32×10^6	100×10^6

Table 5.5: Performance comparison with another GPU DEM code.

5.5.3 Run time linearity

One key objective of particle simulations with collisions is to reduce the simulation run time from $O(N^2)$ to $O(N)$. To assess this performance, simulations with increasing numbers of particles have been run. Notably, these simulations have logging to file disabled in order to assess the algorithm performance only. Figure 5.6 shows the results of these runs. The results show that the simulation is very close to being linear. At high numbers of particles the non-linearity is more evident, but still remains close to the extrapolated line.

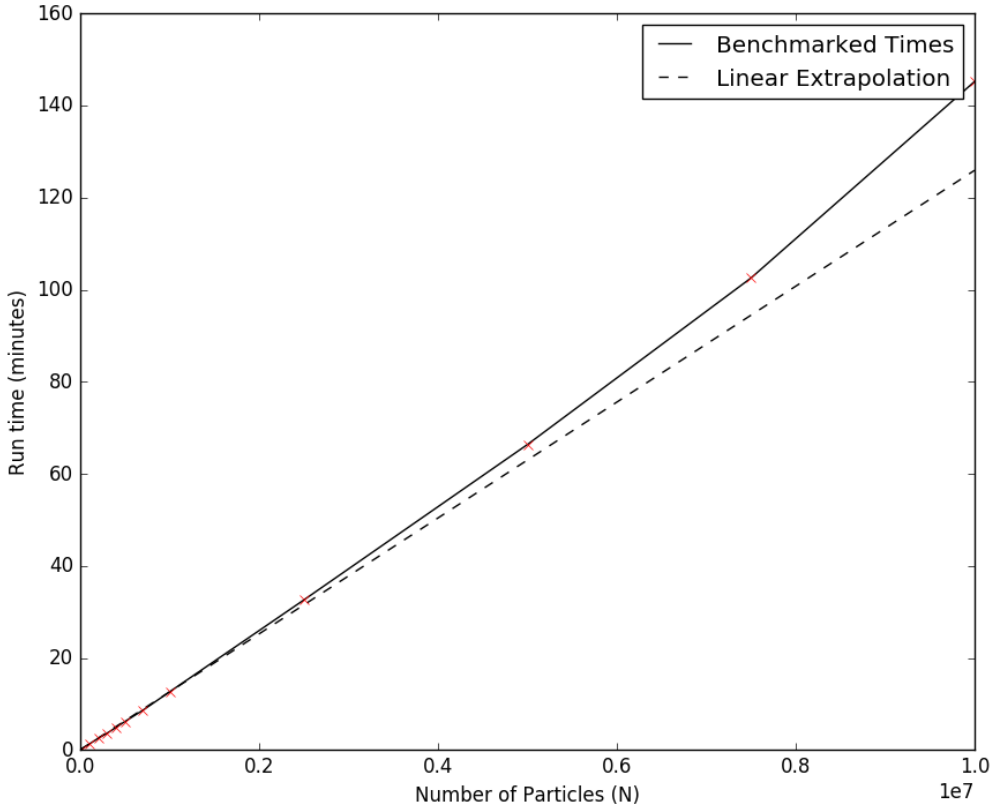


Figure 5.6: Run time against number of particles for the OpenCL simulation compared to a linear extrapolation through $N = 1e4$ and $N = 1e6$.

Chapter 6

Application

An agglomerate is a group of particles held together by cohesive forces. Agglomerates can be formed from electrostatically charged dust, magnetic particles, or a variety of other forces. When designing anything that handles cohesive particles it is important to understand agglomeration behaviour. The simulation tool developed in this project can be applied to studying this behaviour. The two properties that are examined in this chapter are the Stokes Number (see Section 6.1.2) and the Stickiness Number (see Section 6.1.3).

6.1 Simulation Setup

The simulation uses a Taylor-Green Vortex fluid flow field (see Section 6.1.1). It is a relatively simple flow field that can be easily set to fit a cubic domain boundary. It also works well for bringing particles together so that collisions are more frequent than a stationary fluid. It is a constant source of additional energy so that the particles do not simply lose energy and eventually stay at rest.

The simulation uses periodic boundary conditions such that if a particle were to leave the domain it re-enters the domain on the opposite side. This provides a decent approximation of a larger domain with more particles by assuming that the simulation domain is representative of the larger domain as a whole.

Section 6.1.1 describes in more detail the mathematical setup of a Taylor-Green Vortex flow. Section 6.1.2 defines the Stokes Number and how it is varied in the simulation. Section 6.1.3 defines the Stickiness Number and how it is varied in the simulation. Section 6.1.4 defines all of the constant properties used in the simulation. Section 6.1.5 defines the initial conditions of the simulation.

6.1.1 Taylor-Green Vortex Flow

Taylor-Green Vortex flow is a specific solution to the incompressible Navier-Stokes equations. Chow[1] simplifies these equations for cubic vortices with a control variable A and vortex frequency a as shown in Equations 6.1, 6.2, and 6.3. This formulation has vortex boundaries

at $-\pi$ and π. The flow is plotted in Figure 6.1.

$$u = Acos\Big(a\Big(x + \frac{\pi}{2a}\Big)\Big)sin\Big(a\Big(y + \frac{\pi}{2a}\Big)\Big)sin\Big(a\Big(z + \frac{\pi}{2a}\Big)\Big) \tag{6.1}$$

$$v = Asin\Big(a\Big(x + \frac{\pi}{2a}\Big)\Big)cos\Big(a\Big(y + \frac{\pi}{2a}\Big)\Big)sin\Big(a\Big(z + \frac{\pi}{2a}\Big)\Big) \tag{6.2}$$

$$w = -2Asin\Big(a\Big(x + \frac{\pi}{2a}\Big)\Big)sin\Big(a\Big(y + \frac{\pi}{2a}\Big)\Big)cos\Big(a\Big(z + \frac{\pi}{2a}\Big)\Big) \tag{6.3}$$

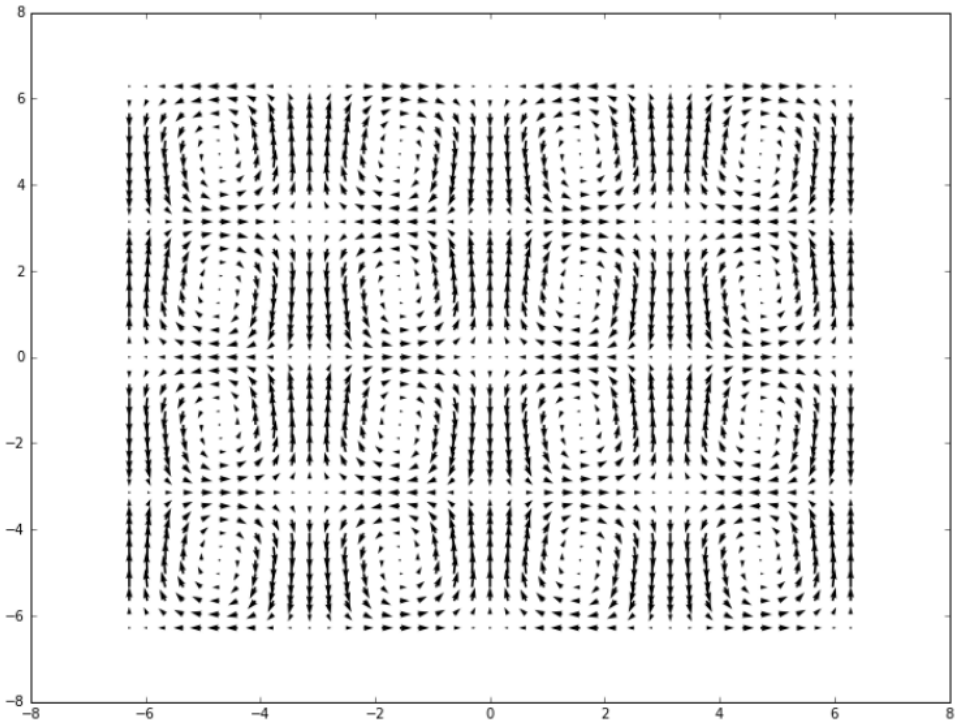


Figure 6.1: A z-y view of a Taylor-Green Vorted flow field. (Figure 5a from Chow[1]).

6.1.2 Stokes Number

The Stokes Number, Stk, describes how a particle moves through a fluid. A high Stokes Number indicates an inertial regime where a particle is relatively unaffected by the fluid. A low Stokes Number indicates a viscous flow regime where the particle closely follows the fluid. A Stokes Number around 1 is in a transitional regime between viscous and inertial. This is shown in Figure 6.2. The Stokes Number for a particle in Stokes flow is defined as in Equation 6.4[1].

For the Taylor-Green Vortex flow used in these simulations l_0 is the diameter of one of the vortices and u_0 is the average fluid flow speed $u_{f,avg}$. Chow[1] calculated that the average flow speed for Taylor-Green Vortex flow is $u_{f,avg} = 0.7839A$.

$$Stk = \frac{\rho_p d_p^2 u_0}{18\mu l_0} \tag{6.4}$$

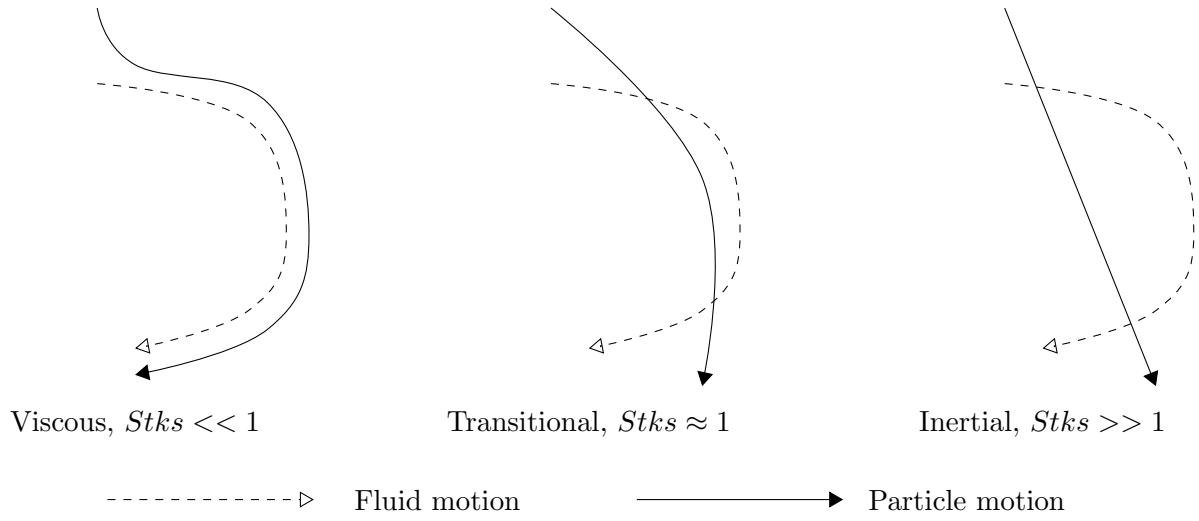


Figure 6.2: Stokes Number regimes.

6.1.3 Stickiness Number

The Stickiness Number, Sy, describes how likely particles are to stick to each other through collisions. L_e is the effect length and is usually $d_e - d_b$, where d_e is the effect diameter and d_b is the body diameter. k_c is the cohesion stiffness from Equation 2.6. ϵ is the coefficient of restitution, u is the average particle speed, and m is the average particle mass.

$$Sy = \frac{L_e\sqrt{k_c}}{\epsilon\sqrt{L_e^2 k_c + u^2 m}} \tag{6.5}$$

This number is derived from the analytic solution for a normal cohesive collision, the full derivation can be found in Section A.3. When the effect length is $d_e - d_b$ and u is the initial collision speed, the Stickiness Number determines whether the particles will stick after the collision.

$$Sy < 1 \ \textit{Does not stick}$$
$$Sy > 1 \ \textit{Sticks}$$

For the general case, lower Stickiness Numbers cause the particles to be less likely to stay together and higher Stickiness Numbers cause the particles to be more likely to stay together. Stickiness Numbers in the range $0 < Sy < 1.5$ are common, higher Stickiness Numbers are uncommon with normal parameters.

6.1.4 Simulation Properties

The constant properties used in the simulations are in Table 6.1. The properties used to achieve chosen Stickiness Numbers and Stokes Numbers are in Tables 6.2 and 6.3, respectively. Note that, due to the long simulation durations, the timestep is only $t_{collision}/8$. Although this is not of the highest accuracy, it does allow for longer simulations and for more data points to be collected. Absolute accuracy is not as important since the focus of these simulations is to observe how agglomerates vary with different properties. A smaller

timestep would be recommended if these simulations were required to predict specific outcomes of real scenarios.

Property	Value
Gravitational Acceleration g	0 m/s^2
Stiffness k_e	1e5 N/m
Cohesion Stiffness k_c	100 N/m
Particle Density ρ	2000 kg/m^3
Particle Diameter d_b	0.05 m
Effect Diameter d_e	0.075 m
Coefficient of Friction μ	0.1
Friction Stiffness k_f	1e5 N/m
Domain Length	2π m
Number of Particles N	10,000
Timestep Δt	$t_{collision}/8 = 0.000449$ s
Simulation Length	600 s

Table 6.1: Constant properties used in the simulations.

	Stickiness Number			
Property	0.1	0.6	1.0	1.5
Coefficient of Restitution ϵ	0.5	0.2	0.174	0.15
Cohesion Stiffness k_c	8.06	47.01	100.0	117.24

Table 6.2: Properties used to achieve chosen Stickiness Numbers.

	Stokes Number			
Property	0.1	1.0	5.0	10.0
Fluid Viscosity μ	10.40	1.040	0.2080	0.1040

Table 6.3: Properties used to achieve chosen Stokes Numbers.

6.1.5 Initial Conditions

The particles are initially arranged in a cubic formation in the centre of the domain as shown in Figure 6.3. The particles are at intervals of approximately $3d_b$. Each particle has a small random offset to avoid the results being affected by an unrealistically ordered system (this has been encountered in other simulations). The particles are given initial velocities with random directions and random speeds. The speeds are distributed with a mean of $1m/s$ and standard deviation of $0.1m/s$. The directions are evenly distributed.

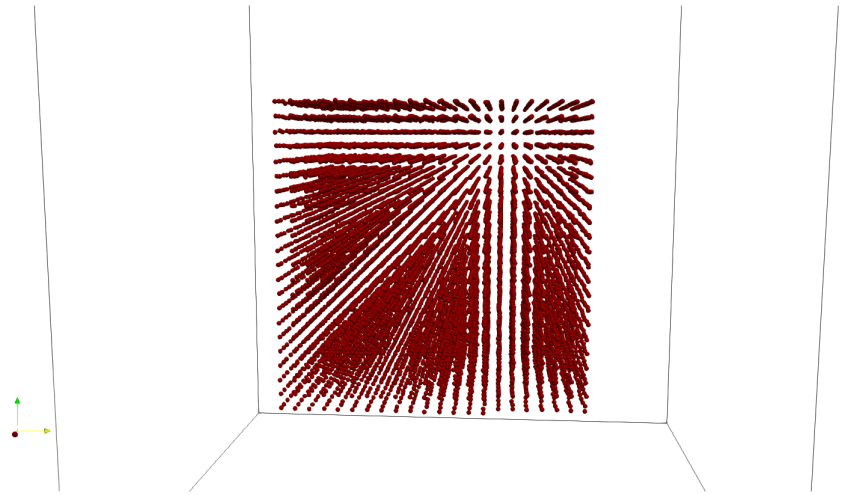


Figure 6.3: A rendered image of the initial positions of the particles. The red spheres are the particles and the black lines are the domain boundary.

6.2 Results and Analysis

6.2.1 Definitions

The two main statistics that are measured from the results of these simulations are the mean agglomerate size and the mean void fraction.

Agglomerate Size

The agglomerate size is the number of particles in an agglomerate. Particles are considered to be in an agglomerate when they are in contact with other particles in the agglomerate. Particles interacting with cohesive forces are not considered to be part of an agglomerate until they are in contact.

Void Fraction

The void fraction of an agglomerate is defined as the fraction of a bounding sphere's volume that does not contain particles. The bounding sphere is defined as the smallest sphere into which the agglomerate can fit, as shown in Figure 6.4. The void fraction can be calculated with Equation 6.6 where $\sum V_p$ is the sum of the particle volumes and V_{bs} is the volume of the bounding sphere. This is a useful statistic because it provides some insight into the structure of an agglomerate. A high void fraction means that the agglomerate has a lower density and so is likely longer and thinner. A low void fraction indicates a tightly packed spherical agglomerate.

$$\text{Void Fraction} = 1 - \frac{\sum V_p}{V_{bs}} \tag{6.6}$$

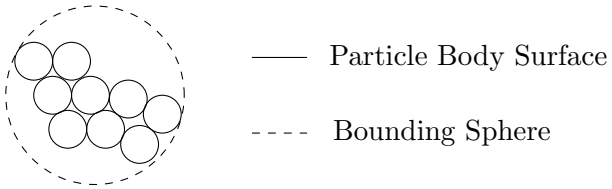


Figure 6.4: The bounding sphere of an agglomerate.

.

6.2.2 General Behaviour

In general the simulations show that the particles initially spread out but are drawn together at the 'corners' of the vortices. Particles begin to bunch up in these areas and form agglomerates. The sizes and shapes of the agglomerates that form depend on the Stokes and Stickiness numbers.

Figure 6.5 shows how the particles are arranged in the flow. The particle traces show where the vortices are located. By comparing this with Figure 6.1 it is evident that the agglomerates are formed where the flow comes together in the corners of the vortices and are elongated by the flow then pulling them apart.

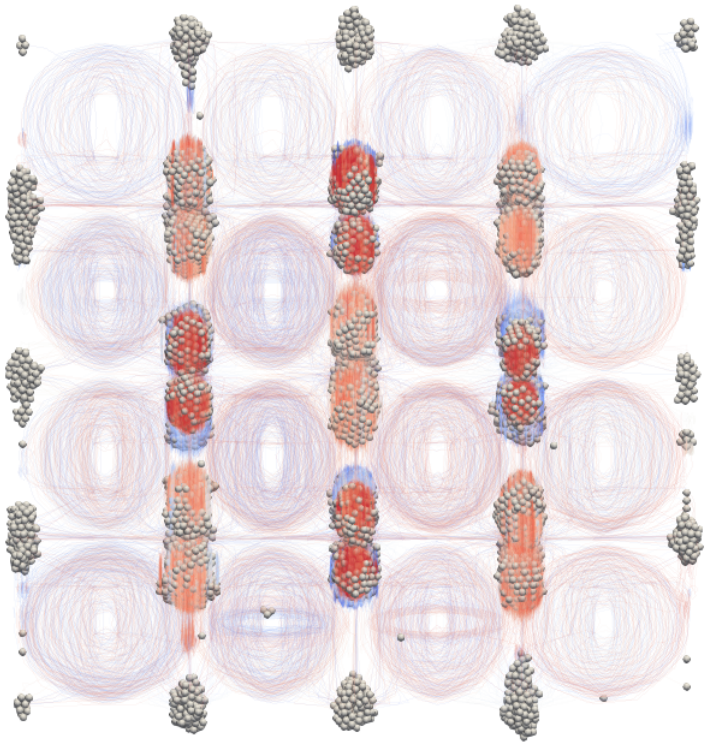


Figure 6.5: A 2D projection of agglomerates in a Taylor-Green Vortex Flow with particle traces to show the locations of the vortices.

To determine the effect of changes in properties the simulation must reach a statistically static state. Agglomerates may continue forming and breaking up, but the mean size and void fraction should tend to some value. This behaviour is shown in Figure 6.6.

For some simulations a fully statistically static state was not achieved. In some cases there was a rapid increase in mean size and void fraction at the end of the simulation and in others the statistics had not settled completely. These effects can be seen in Figure 6.7. To reduce the impact of this on the final values the overall value for the mean size and mean void fraction have been determined by finding an average of each property between $t = 500s$ and $t = 550s$. This provides more stable results for later analysis. Further analysis could be performed to ascertain the causes of these effects, however the results of these simulations are sufficient to determine the general trends in the simulations.

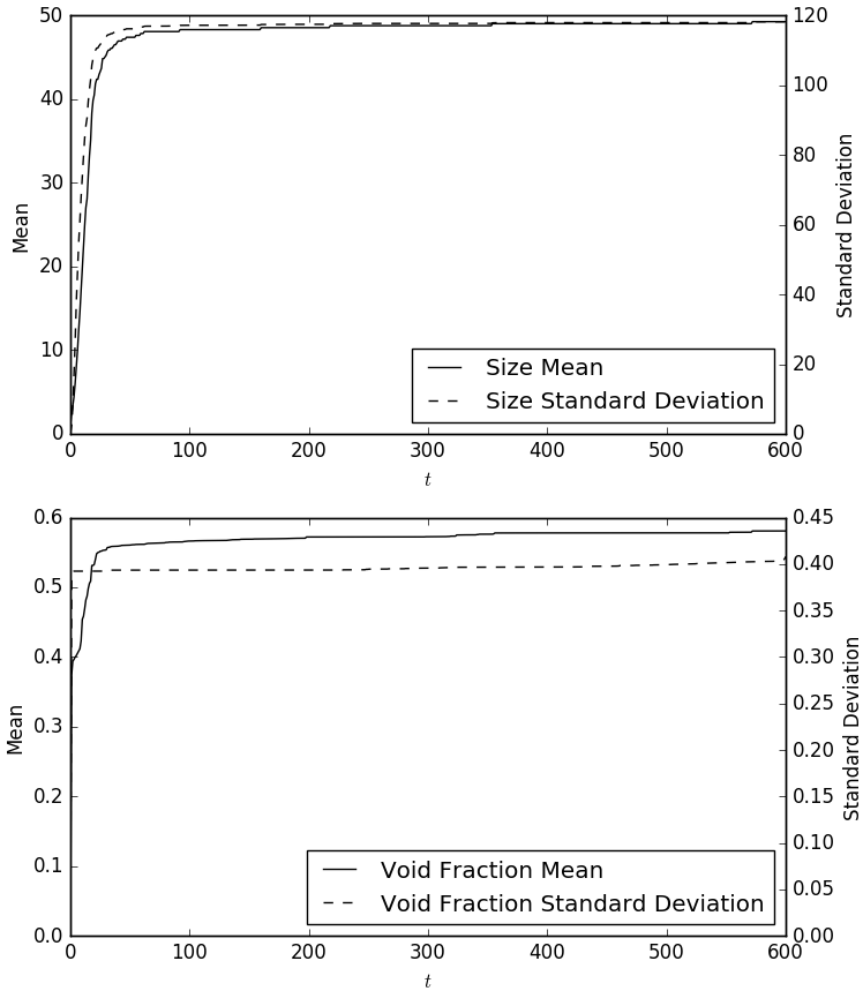


Figure 6.6: Mean and standard deviation graphs for agglomerate size and void fraction in a simulation with $Stk = 5$ and $Sy = 1.5$. These graphs show the simulation coming to a statistically static state.

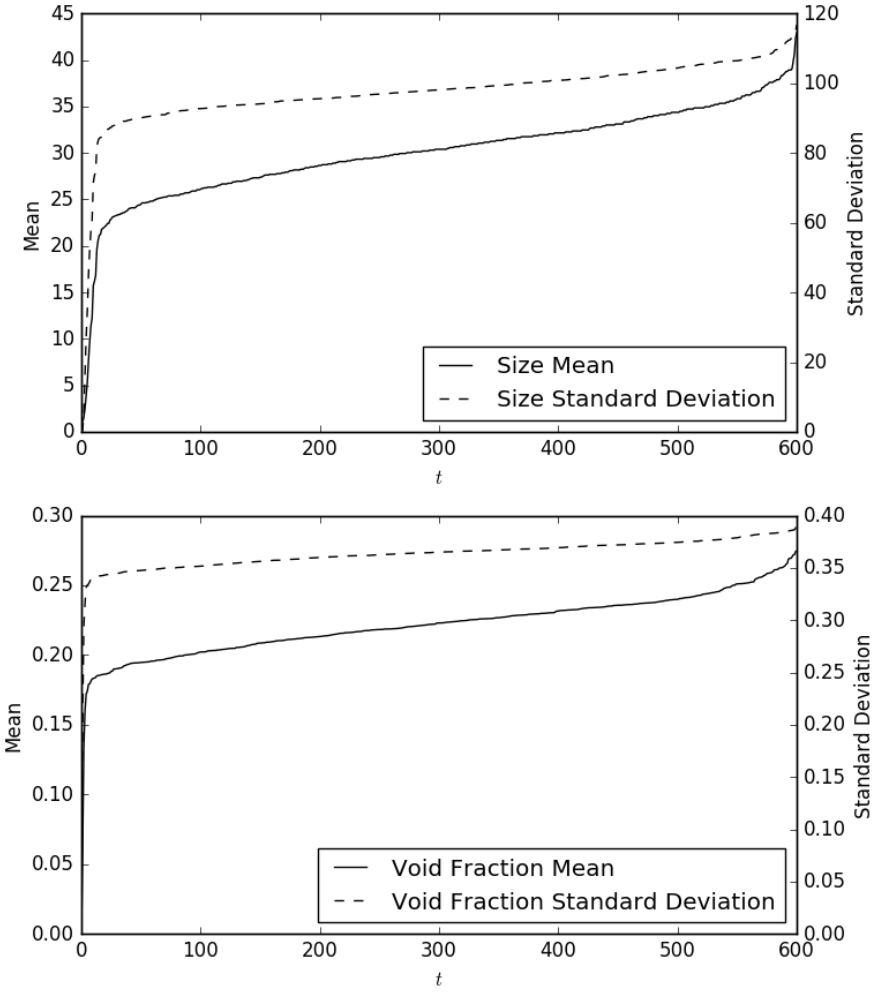


Figure 6.7: Mean and standard deviation graphs for agglomerate size and void fraction in a simulation with $Stk = 1$ and $Sy = 0.1$. These graphs show how a statistically static state is not always perfectly achieved.

6.2.3 Variation with Stokes and Stickiness

Simulations were performed with Stokes Numbers of 0.1, 1, 5, and 10 and Stickiness Numbers of 0.1, 0.6, 1, and 1.5. Figure 6.8 shows the results of these simulations, graphed as mean size and mean void fraction.

Size

The graph of mean size shows that for small Stokes Numbers the agglomerate size is very large. This is because the particles are very quickly brought together by the vortices and then stay together due to the Stickiness. The agglomerate size is thus influenced by the initial conditions, this effect is discussed further in Section 6.2.4. For the lowest Stickiness Number value the agglomerate size is very small, this is where very few agglomerates are formed because the cohesion force is not strong enough.

For larger Stokes Numbers the particles are more free to move around. The general trend is that the mean agglomerate size increases as the Stokes Number increases and also increases as the Stickiness Number increases.

Void Fraction

The graph of mean void fraction shows similar trends. The void fraction of agglomerates for low Stokes Numbers is quite high, except for very low Stickiness Numbers. For low Stickiness Numbers there are very few agglomerates and so the void fraction is decreased by the many particles that are on their own and thus have a void fraction of 0.

For higher Stokes Numbers the void fraction increases with Stokes Number, however the trend with Stickiness Numbers is less clear. More data points would be required to draw any clear conclusions from the Stickiness Number trends.

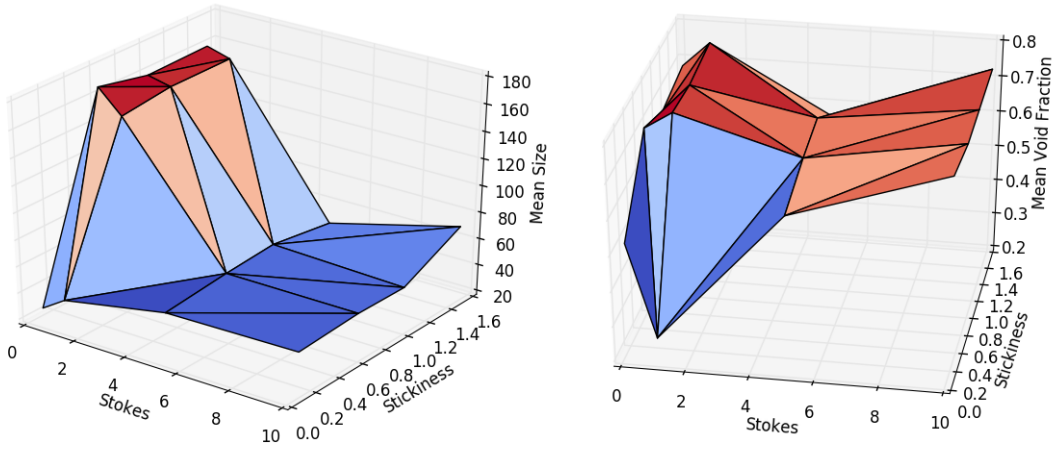


Figure 6.8: Graphs of mean size (left) and mean void fraction (right) against varying Stokes Number and Stickiness Number.

6.2.4 Variation with Initial Conditions

Some simulations with high Stickiness Numbers ($Sy = 1.5$) were performed with different initial distributions of particles. At high Stokes Numbers, where the particles were in the inertial regime, this had no effect on the results. However, at low Stokes Numbers, where the particles were in the viscous regime, this had a significant effect because the particles are very quickly drawn together by the flow and rapidly form stable agglomerates due to the high Stickiness Number. With a smaller initial spread of particles the agglomerates are much larger because the particles are drawn to the same vortex corners. With a larger initial spread of particles the agglomerates are much smaller because the particles are drawn to vortex corners further apart. This effect is clearly observable in Figure 6.9. This effect also

causes the void fraction to be lower because the particles form more spherical agglomerates due to the larger size.

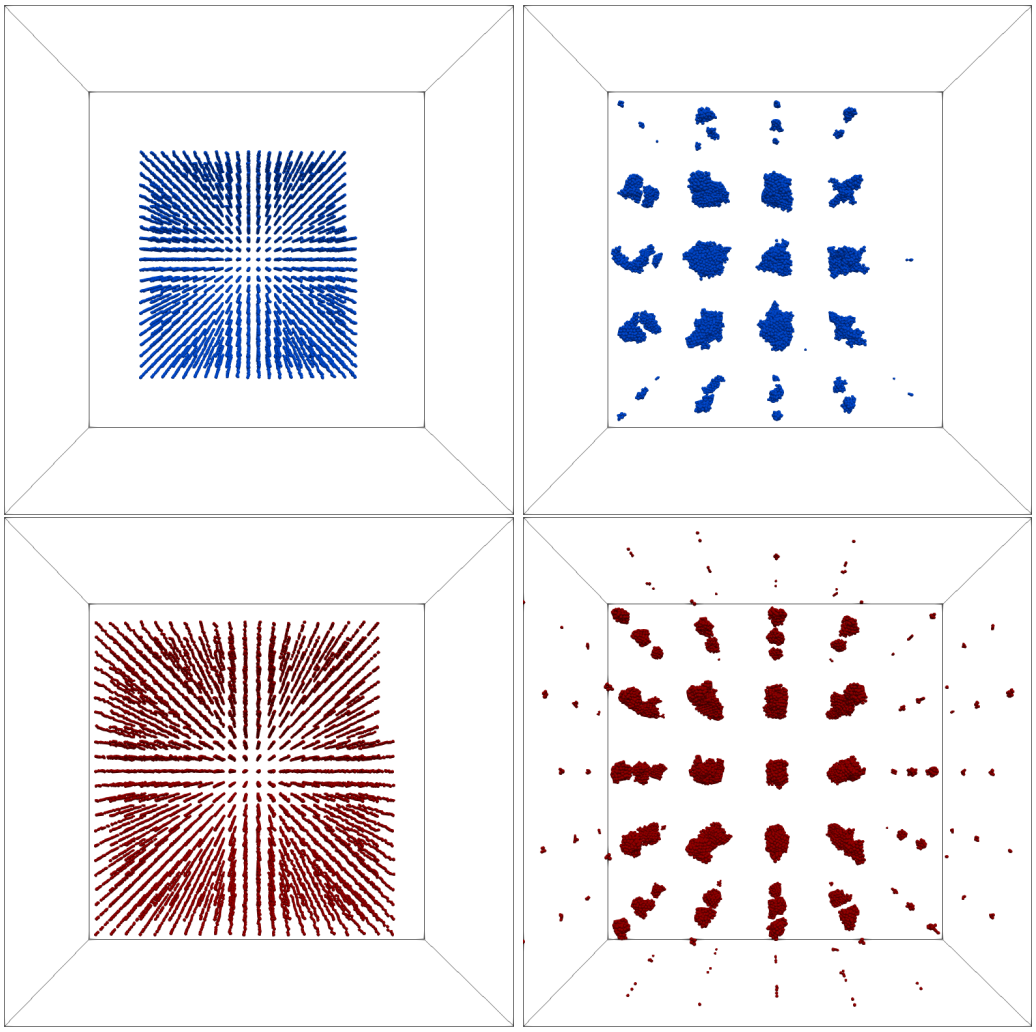


Figure 6.9: Two sets of simulations with different initial particle spreads. Left images are at $t = 0$, Right images are at $t = 60$. The top (blue) images are for a small initial spread and the bottom (red) images are for a larger initial spread. For these simulations $Stk = 1$ and $Sy = 1.5$.

6.3 Recommendations for Future Analysis

Section 6.2.4 shows that the results for low Stokes Numbers are very sensitive to initial conditions. Further analysis of this sensitivity would be valuable. It would likely be most useful to distribute particles evenly across the domain. Giving the particles the local flow velocity as their initial velocity could also improve the predictability of the results.

Further analysis of variation of agglomerate properties with a variation in Stokes Number and Stickiness Number would be especially interesting in the ranges $0 < Stk < 5$ and $0 < Sy < 0.6$. This is the area with the most variation and likely contains the most interesting insights into agglomerate formation.

Chapter 7

Conclusion

This project has successfully met all of its aims and objectives from Section 1.2. A simple particle simulation was developed with Python and provided some useful insights into how the DEM works and how best to implement it. A GPU based particle simulation was developed using OpenCL and was capable of running simulations with large numbers of particles. This simulation was then used to observe how agglomerates form with varying simulation properties. The analysis provides a strong basis for future analysis of agglomerate formation in Taylor-Green Vortex flow.

7.1 Further Work

7.1.1 Simulation Improvements

Particle Rotation

The DEM implementations in this project do not consider particle rotation. Future work could implement particle rotations to improve the simulation accuracy and give the simulation broader applicability. It is recommended that rotations be implemented using quaternions rather than Euler angles. Quaternions avoid problems such as gimbal lock and simplify rotation calculations.

Polyhedral Particles

This project only considers spherical particles, an implementation of polyhedral particles could give the simulation broader applicability. This would require particle rotation to be considered but would allow for analysis of more complex particle populations and more varied simulation parameters.

Collision Detection Algorithms

The spatial zoning algorithm in this project is simple but is not the most efficient algorithm available. Other algorithms, such as triangulation[13], could be implemented to improve the efficiency of the simulation. More advanced algorithms could also simplify the implementation of better models, particularly friction models, that require collision history.

Particle-Fluid Interaction

This project assumes that the particle does not significantly affect the fluid, this vastly simplifies the simulation. However, particle interaction with the fluid could affect how the particles move and how groups of particles affect each other through the fluid. This would require a computational fluid dynamics code to be built into the simulation.

7.1.2 Additional Analysis

Additional analysis of agglomeration behaviour in Taylor-Green Vortex flow could be of value. Observing how agglomerates form with other properties, for example, could lead to a greater understanding of the overall system. Some interesting properties to vary would be the size of the vortices, the standard deviation of particle masses, and variation of initial conditions.

Appendix A

Derivations

A.1 Normal Collision

$$m\ddot{x} = k_e\delta - \eta\dot{x}$$

$$\delta = d_b - x$$

$$m\ddot{x} = k_e d_b - k_e x - \eta\dot{x}$$

$$m\ddot{x} + \eta\dot{x} + k_e x = d_b k_e$$

Complementary Function

Auxilliary Equation: $mp^2 + \eta p + k_e = 0$

$$p_{1,2} = \frac{-\eta \pm \sqrt{\eta^2 - 4mk_e}}{2m}$$

For this case $\eta^2 > 4mk_e$ so p_1 and p_2 are complex.

Let: $a = \frac{-\eta}{2m}$ and $b = \frac{\sqrt{4mk_e - \eta^2}}{2m}$

$$x = e^{at}(A_1 sin(bt) + A_2 cos(bt))$$

Particular Integral

Ansatz: $x = B$

$$\dot{x} = \ddot{x} = 0$$

$$k_e B = d_b k_e$$

$$B = d_b$$

$$x = d_b$$

General Solution

$$x = e^{at}(A_1 sin(bt) + A_2 cos(bt)) + d_b$$

$$x(0) = d_b = A_2 + d_b$$
$$A_2 = 0$$

$$\dot{x}(0) = u_0 = A_1 b$$
$$A_1 = \frac{u_0}{b}$$

$$x = e^{at}\frac{u_0}{b} sin(bt) + d_b$$
$$\dot{x} = u_0 e^{at}(\frac{a}{b} sin(bt) + cos(bt))$$

A.1.1 Collision Duration

The duration of a collision is considered to be the time for which the particles' body surfaces are touching. For the normal collision used in section A.1 this is from $t = 0$ until some t_{col} when the moving particle returns to its starting position.

Let: $a = \frac{-\eta}{2m}$ and $b = \frac{\sqrt{4mk_e - \eta^2}}{2m}$

$$x = e^{at}\frac{u_0}{b} sin(bt) + d_b$$

At $t = t_{col}$, $x = d_b$

$$e^{at_{col}}\frac{u_0}{b} sin(bt_{col}) = 0$$

$$sin(bt_{col}) = 0$$

$$t_{col} = \frac{sin^{-1}(0)}{b}$$

$$t_{col} = \frac{0 + n\pi}{b} = \frac{n\pi}{b} \text{ , } n \in \mathbb{Z}$$

$n = 0$ Initial conditions.

$n = 1$ First crossing after collision, physical result for t_{col}.

$$t_{col} = \frac{\pi}{b} = \frac{2\pi m}{\sqrt{4mk_e - \eta^2}}$$

This can be rearranged to use the coefficient of restitution:

$$t_{col} = \sqrt{\frac{m}{k_e}}\sqrt{\pi^2 + ln(\epsilon)^2}$$

A.1.2 Coefficient of Restitution

The coefficient of restitution, ϵ, is the ratio of the speeds before (u_0) and after (u_1) the collision. Since the x velocity after the collision is in the opposite direction to the x velocity before the collision, $\epsilon = \frac{-u_1}{u_0}$. The velocity after the collision is determined by finding the

velocity at $t = t_{col}$ where t_{col} is the collision duration determined in section A.1.1.

From section A.1: $\dot{x} = u_0 e^{at}(\frac{a}{b}sin(bt) + cos(bt))$

From section A.1.1: $t_{col} = \frac{\pi}{b}$

$$u_1 = u_0 e^{at_{col}}(\frac{a}{b}sin(bt_{col}) + cos(bt_{col}))$$

$$u_1 = u_0 e^{a\pi/b}(\frac{a}{b}sin(\pi) + cos(\pi))$$

$$u_1 = -u_0 e^{a\pi/b}$$

$$\epsilon = \frac{-u_1}{u_0} = \frac{u_0 e^{a\pi/b}}{u_0} = e^{a\pi/b}$$

To determine the damping coefficient for a given ϵ:

$$ln(\epsilon) = \frac{a\pi}{b} = \frac{-\eta\pi}{\sqrt{4mk_e - \eta^2}}$$

$$\eta^2 = \frac{4mk_e ln(\epsilon)^2}{\pi^2 + ln(\epsilon)^2}$$

$$\eta = \pm\sqrt{\frac{4mk_e ln(\epsilon)^2}{\pi^2 + ln(\epsilon)^2}}$$

η must be +ve, $ln(\epsilon)$ is -ve so we take the -ve square root.

$$\eta = -\sqrt{\frac{4mk_e ln(\epsilon)^2}{\pi^2 + ln(\epsilon)^2}} = -2ln(\epsilon)\sqrt{\frac{mk_e}{\pi^2 + ln(\epsilon)^2}}$$

This equation matches the one found in Tuley[10].

A.2 Normal Collision with Cohesion

A normal collision with cohesion has three regimes.

$$m\frac{du}{dt} = 0 \text{ for } x > d_e$$

$$m\frac{du}{dt} = -k_c\delta_e \text{ for } d_e > x > d_b$$

$$m\frac{du}{dt} = m\ddot{x} = k_e\delta_b - \eta\dot{x} - k_c\delta_e \text{ for } x < d_b$$

Two important definitions are those of body overlap (δ_b) and effect overlap (δ_e).

$$\delta_b = d_b - x$$

$$\delta_e = d_e - x$$

A.2.1 Cohesion Only, Incoming ($d_e > x > d_b, u < 0$)

$$m\ddot{x} = -k_c\delta_e = -k_c(d_e - x)$$
$$m\ddot{x} - k_c x = -k_c d_e$$

Complementary Function

Auxilliary Equation: $mp^2 - k_c = 0$

$$p = \pm\sqrt{\frac{k_c}{m}}$$
$$x = A_1 e^{t\sqrt{k_c/m}} + A_2 e^{-t\sqrt{k_c/m}}$$

Particular Integral

Ansatz: $x = B$

$$\dot{x} = \ddot{x} = 0$$
$$-k_c x = -k_c d_e$$
$$-k_c B = -k_c d_e$$
$$B = d_e$$

General Solution

$$x = A_1 e^{t\sqrt{k_c/m}} + A_2 e^{-t\sqrt{k_c/m}} + d_e$$
$$\dot{x} = \sqrt{\frac{k_c}{m}} A_1 e^{t\sqrt{k_c/m}} - \sqrt{\frac{k_c}{m}} A_2 e^{-t\sqrt{k_c/m}}$$

$$x(0) = d_e = A_1 + A_2 + d_e$$
$$A_1 = -A_2$$

$$\dot{x}(0) = u_0 = \sqrt{\frac{k_c}{m}} A_1 - \sqrt{\frac{k_c}{m}} A_2$$
$$A_1 = \frac{u_0}{2}\sqrt{\frac{m}{k_c}}$$

$$x = \frac{u_0}{2}\sqrt{\frac{m}{k_c}} e^{t\sqrt{k_c/m}} - \frac{u_0}{2}\sqrt{\frac{m}{k_c}} e^{-t\sqrt{k_c/m}} + d_e$$
$$x = u_0\sqrt{\frac{m}{k_c}} sinh\Big(t\sqrt{\frac{k_c}{m}}\Big) + d_e$$

$$\dot{x} = \frac{u_0}{2} e^{t\sqrt{k_c/m}} + \frac{u_0}{2} e^{-t\sqrt{k_c/m}}$$
$$\dot{x} = u_0 cosh\Big(t\sqrt{\frac{k_c}{m}}\Big)$$

Two important results from this solution are the impact time (t_i) and impact velocity (u_i).

Impact Time (t_i)

$$\text{At impact: } x = d_b$$
$$d_b = \frac{u_0}{2}\sqrt{\frac{m}{k_c}}e^{t_i\sqrt{k_c/m}} - \frac{u_0}{2}\sqrt{\frac{m}{k_c}}e^{-t_i\sqrt{k_c/m}} + d_e$$
$$\frac{2(d_b - d_e)}{u_0}\sqrt{\frac{k_c}{m}} = e^{t\sqrt{k_c/m}} - e^{-t\sqrt{k_c/m}} = 2sinh(\sqrt{\frac{k_c}{m}}t)$$
$$t_i = \sqrt{\frac{m}{k_c}}sinh^{-1}\left(\frac{d_b - d_e}{u_0}\sqrt{\frac{k_c}{m}}\right)$$

Impact Velocity (u_i)

$$u_i = \frac{u_0}{2}e^{t_i\sqrt{k_c/m}} + \frac{u_0}{2}e^{-t_i\sqrt{k_c/m}}$$
$$u_i = u_0 cosh\left(t_i\sqrt{\frac{k_c}{m}}\right)$$

A.2.2 Full Contact ($x < d_b$)

$$m\frac{du}{dt} = m\ddot{x} = k_e\delta_b - \eta\dot{x} - k_c\delta_e$$
$$m\ddot{x} + \eta\dot{x} + (k_e - k_c)x = k_e d_b - k_c d_e$$

Complementary Function

Auxilliary Equation: $mp^2 + \eta p + k_e - k_c = 0$

$$p_{1,2} = \frac{-\eta \pm \sqrt{\eta^2 - 4m(k_e - k_c)}}{2m}$$

For this case $k_e > k_c$ and $\eta^2 > 4m(k_e - k_c)$ so p_1 and p_2 are complex.

Let: $a = \frac{-\eta}{2m}$ and $b = \frac{\sqrt{4mk_e - \eta^2}}{2m}$

$$x = e^{at}(A_1 sin(bt) + A_2 cos(bt))$$

Particular Integral

Ansatz: $x = B$

$$\dot{x} = \ddot{x} = 0$$
$$(k_e - k_c)B = k_e d_b - k_c d_e$$
$$B = \frac{k_e d_b - k_c d_e}{k_e - k_c}$$

General Solution

$$x = e^{at}(A_1 sin(bt) + A_2 cos(bt)) + \frac{k_e d_b - k_c d_e}{k_e - k_c}$$
$$\dot{x} = e^{at}((A_1 a - A_2 b)sin(bt) + (A_1 b + A_2 a)cos(bt))$$

To simplify the result the t here is $t - t_i$ in the overall collision.

$$x(0) = d_b$$
$$A_2 + \frac{k_e d_b - k_c d_e}{k_e - k_c} = d_b$$
$$A_2 = \frac{k_c(d_e - d_b)}{k_e - k_c}$$

Let: $c = \frac{k_c(d_e - d_b)}{k_e - k_c}$

$$\dot{x}(0) = aA_2 + bA_1 = u_i$$
$$A_1 = \frac{u_i - aA_2}{b} = \frac{u_i - ac}{b}$$

$$x = e^{at}(\frac{u_i - ac}{b} sin(bt) + ccos(bt)) + \frac{k_e d_b - k_c d_e}{k_e - k_c}$$
$$\dot{x} = e^{at}\Big(\Big(\frac{u_i - ac}{b}a - cb\Big)sin(bt) + u_i cos(bt)\Big)$$

The position cannot be analytically solved for t in order to find the initial return velocity. A reasonable estimation for this is to use $t_{col} = \frac{\pi}{b}$, and other results, from section A.1. Thus the initial return velocity, u_r, can be found to be $u_r = -\epsilon u_i$.

A.2.3 Cohesion Only, Returning ($d_e > x > d_b, u > 0$)

The General Solution to the returning equation is the same as the incoming equation.

General Solution

$$x = A_1 e^{t\sqrt{k_c/m}} + A_2 e^{-t\sqrt{k_c/m}} + d_e$$
$$\dot{x} = \sqrt{\frac{k_c}{m}} A_1 e^{t\sqrt{k_c/m}} - \sqrt{\frac{k_c}{m}} A_2 e^{-t\sqrt{k_c/m}}$$

To simplify the result the t here is $t - t_r$ in the overall collision.

Where t_r is the return time.

$$x(0) = d_b = A_1 + A_2 + d_e$$
$$A_1 + A_2 = (d_b - d_e)$$

$$\dot{x}(0) = u_r = \sqrt{\frac{k_c}{m}} A_1 - \sqrt{\frac{k_c}{m}} A_2$$
$$A_1 = \frac{1}{2}\Big(d_b - d_e + u_r \sqrt{\frac{m}{k_c}}\Big)$$
$$A_2 = \frac{1}{2}\Big(d_b - d_e - u_r \sqrt{\frac{m}{k_c}}\Big)$$
$$A_1 = \frac{u_0}{2}\sqrt{\frac{m}{k_c}}$$

$$x = (d_b - d_e) cosh\Big(\sqrt{\frac{k_c}{m}} t\Big) + u_r \sqrt{\frac{m}{k_c}} sinh\Big(\sqrt{\frac{k_c}{m}} t\Big) + d_e$$
$$\dot{x} = \sqrt{\frac{k_c}{m}} (d_b - d_e) sinh\Big(\sqrt{\frac{k_c}{m}} t\Big) + u_r cosh\Big(\sqrt{\frac{k_c}{m}} t\Big)$$

A.3 Stickiness Number

Taking the results from section A.2 it is known that if the particle is to escape the collision the position, x, will at some point return to d_e. Thus, using the solution for the returning

collision section (see Section A.2.3), the position equation can be solved for t:

$$d_e = (d_b - d_e)cosh\Big(\sqrt{\frac{k_c}{m}}t\Big) + u_r\sqrt{\frac{m}{k_c}}sinh\Big(\sqrt{\frac{k_c}{m}}t\Big) + d_e$$

$$(d_b - d_e)cosh\Big(\sqrt{\frac{k_c}{m}}t\Big) + u_r\sqrt{\frac{m}{k_c}}sinh\Big(\sqrt{\frac{k_c}{m}}t\Big) = 0$$

$$tanh(\sqrt{\frac{k_c}{m}}t) = \frac{d_e - d_b}{u_r\sqrt{\frac{m}{k_c}}}$$

This only has a solution when the RHS is less than 1.

Thus, for a particle to stick in a collision:

$$\frac{d_e - d_b}{u_r}\sqrt{\frac{k_c}{m}} > 1$$

u_r is the velocity of the particle as it returns from the contact collision. Although not precise, the non-cohesion restitution coefficient ϵ provides a good approximation for a relationship between the impact velocity u_i and the return velocity u_r.

$$u_r = -\epsilon u_i$$

u_i relates to u_0 by the relationship from Section A.2.1:

$$u_i = u_0 cosh\Big(t_i\sqrt{\frac{k_c}{m}}\Big) = u_0 cosh\Big(sinh^{-1}\Big(\frac{d_b - d_e}{u_0}\sqrt{\frac{k_c}{m}}\Big)\Big)$$

$$\frac{d_e - d_b}{u_r}\sqrt{\frac{k_c}{m}} = \frac{d_e - d_b}{-\epsilon u_0 cosh\Big(sinh^{-1}\Big(\frac{d_b - d_e}{u_0}\sqrt{\frac{k_c}{m}}\Big)\Big)}\sqrt{\frac{k_c}{m}}$$

$$\frac{d_e - d_b}{u_r}\sqrt{\frac{k_c}{m}} = \frac{(d_b - d_e)\sqrt{k_c}}{\epsilon\sqrt{(d_b - d_e)^2 k_c + u_0^2 m}}$$

Finding the dimensions of this number:

Dimensions of stiffness: $[M][L][T]^{-2}[L]^{-1} = [M][T]^{-2}$

$$\frac{[L]([M][T]^{-2})^{0.5}}{[1]([L]^2[M][T]^{-2} + [L]^2[T]^{-2}[M])^{0.5}}$$

$$= \frac{[L][M]^{0.5}[T]^{-1}}{[L][T]^{-1}[M]^{0.5}} = [1]$$

This shows that the number is dimensionless.

A.4 Dynamic Friction Sliding

$$
\begin{aligned}
F_n &= mg \\
F_t^{dynamic} &= -\mu |F_n| \\
\frac{du}{dt} = \ddot{x} &= \frac{-\mu mg}{m} = -\mu g \\
\dot{x} &= -\mu g t + u_0 \\
x &= \frac{-\mu g}{2} t^2 + u_0 t + x_0
\end{aligned}
$$

A.5 Particle Drag

$$m\frac{du}{dt} = \frac{m}{\tau}(u_f - u)$$
$$\ddot{x} + \frac{1}{\tau}\dot{x} = \frac{u_f}{\tau}$$

Complementary Function

Auxilliary Equation: $p^2 + \frac{1}{\tau}p = 0$

$$p_1 = 0, p_2 = -\frac{1}{\tau}$$
$$x = A_1 e^{0t} + A_2 e^{-t/\tau}$$

Particular Integral

Ansatz: $x = Bt$

$$\dot{x} = B$$
$$\ddot{x} = 0$$
$$\frac{B}{\tau} = \frac{u_f}{\tau}$$
$$B = u_f$$

General Solution

$$x = A_1 + A_2 e^{-t/\tau} + u_f t$$
$$\dot{x} = \frac{-A_2}{\tau}e^{-t/\tau} + u_f$$

$$x(0) = 0 = A_1 + A_2$$
$$\dot{x}(0) = 0 = \frac{-A_2}{\tau} + u_f$$
$$A_2 = u_f\tau$$
$$A_1 = -u_f\tau$$

$$x = u_f\tau(e^{-t/\tau} - 1) + u_f t$$
$$\dot{x} = u_f(1 - e^{-t/\tau})$$

Bibliography

[1] Andrew Chow. Programming gpu cards with opencl to predict the motion of billions of particles. 2017.

[2] Peter Alan Cundall. *The measurement and analysis of accelerations in rock slopes*. PhD thesis, Imperial College London, 1971.

[3] Nicolin Govender, Daniel N. Wilke, and Schalk Kok. Blaze-DEMGPU: Modular high performance DEM framework for the GPU architecture. *SoftwareX*, 5:62 – 66, 2016.

[4] J.Q. Gan, Z.Y. Zhou, and A.B. Yu. A GPU-based DEM approach for modelling of particulate systems. *Powder Technology*, 301(Supplement C):1172 – 1182, 2016.

[5] Ji Qi, Kuan-Ching Li, Hai Jiang, Qingguo Zhou, and Lei Yang. Gpu-accelerated dem implementation with cuda. volume 11, pages 330 – 337, 2015.

[6] Nicolin Govender, Daniel N. Wilke, and Schalk Kok. Collision detection of convex polyhedra on the nvidia gpu architecture for the discrete element method. *Applied Mathematics and Computation*, 267:810 – 829, 2015.

[7] In Soo Seo, Ju Hyeon Kim, Jae Ho Shin, Sang Woo Shin, and Sang Hwan Lee. Particle behaviors of printing system using gpu-based discrete element method. *Journal of Mechanical Science and Technology*, 28(12):5083 – 5087, 2014.

[8] Sebastian Kuckuk, Tobias Preclik, and Harald Köstler. Interactive particle dynamics using opencl and kinect. *International Journal of Parallel, Emergent and Distributed Systems*, 28(6):519–536, 2013.

[9] T. Washizawa and Y. Nakahara. "Parallel Computing of Discrete Element Method on GPU". *ArXiv e-prints*, jan 2013.

[10] Robert James Tuley. *Modelling dry powder inhaler operation with the discrete element method*. PhD thesis, Imperial College London, 2008.

[11] Arman Pazouki, Michał Kwarta, Kyle Williams, William Likos, Radu Serban, Paramsothy Jayakumar, and Dan Negrut. Compliant contact versus rigid contact: A comparison in the context of granular dynamics. *Phys. Rev. E*, 96:042905, Oct 2017.

[12] Elijah Andrews. DEMApples. https://github.com/Xorgon/DEMApples, 2018.

[13] J. . Ferrez. Dynamic triangulations for efficient 3D simulation of granular materials. *Dynamic Triangulations for Efficient 3D Simulation of Granular Materials*, 2001. Cited By :28.

[14] Elijah Andrews. DEMOranges. https://github.com/Xorgon/DEMOranges, 2018.

[15] Elijah Andrews. DEMOranges Wiki. https://github.com/Xorgon/DEMOranges/wiki, 2018.

[16] Anca Hamuraru. Atomic operations for floats in OpenCL - improved. https://streamhpc.com/blog/2016-02-09/atomic-operations-for-floats-in-opencl-improved/, 2016. Accessed: 2018-04-13.

[17] Nicolin Govender, Daniel N. Wilke, and Schalk Kok. Collision detection of convex polyhedra on the nvidia gpu architecture for the discrete element method. *Applied Mathematics and Computation*, 267:810 – 829, 2015. The Fourth European Seminar on Computing (ESCO 2014).