# Contents

# Abstract

# Nomenclature

$\delta$         Particle Body Surface Overlap Distance

$\delta_e$         Particle Effect Surface Overlap Distance

$\eta$         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

$\mathbf{x}$         Particle Position Vector

$\mu$         Dynamic Viscosity

$\mu$         Friction Coefficient

$\rho_p$         Particle Density

$\tau_p$         Particle Relaxation Time

$\zeta$         Tangential Displacement During Interaction

$d_b$         Particle Body Surface Diameter

$d_e$         Particle Effect Surface Diameter

$d_p$         Particle Diameter

$k_c$         Cohesion Stiffness

$k_e$         Normal Collision Stiffness

$k_f$         Friction Stiffness

$N$         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

# Chapter 1

# Introduction

## 1.1 Project Overview

*What the project is about, why it is significant etc.*

## 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]. Improvements in the algorithm have been proposed and implemented, and it has proven to be a reliable technique.

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 is restricted to running on NVIDIA GPUs. This means that 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 includes CPUs, GPUs and other, more novel, processing units. 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 testing, decision, and implementation 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 of the key aspects of the DEM as well as its application in powder simulations.

# Chapter 2

# The Discrete Element Method

*< I'm considering adding an overall algorithm flowchart somewhere in this chapter. >*
The Discrete Element Method (DEM) is a numerical method for simulating how particles move and interact. The general principle is that 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. The DEM can be implemented with either soft or hard collision models. Soft models allow for overlap and treat collisions as sustained events whereas hard models treat collisions as an instantaneous event and model forces as an impulse. Soft collision models have broader applicability as they model sustained contact and multiple simultaneous collisions as well as some other complex phenomena[11]. For this reason a soft contact model has been chosen for this project.

## 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.
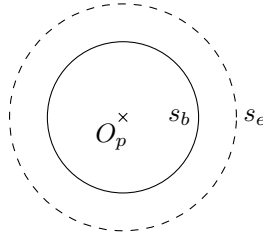


**Figure 2.1:** Definition of a particle.

## 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 collision stiffness, $\delta$ is the particle overlap, $\hat{\mathbf{n}}$ is the unit vector normal to the collision, $\eta$ 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]

The static regime friction force is calculated with equation 2.2 where $k_f$ is the friction stiffness, $\zeta$ is the tangential displacement during the interaction, and $\hat{\mathbf{t}}$ is the unit vector tangential to the collision.

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

The dynamic regime friction force is calculated with equation 2.3 where $\mu$ is the coefficient of friction.

$$\mathbf{F}_t^{dynamic} = -\mu |\mathbf{F}_n| \hat{\mathbf{t}} \tag{2.3}$$

The final tangential friction force is defined in equation 2.4.

$$\mathbf{F}_t = -\hat{\mathbf{t}} min(|\mathbf{F}_t^{static}|, |\mathbf{F}_t^{dynamic}|) \tag{2.4}$$

### 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 2.5 where $k_c$ is the cohesion stiffness and $\delta_e$ is the particle effect surface overlap.

$$\mathbf{F}_c = k_c \delta_e \hat{\mathbf{n}} \tag{2.5}$$

## 2.3 Equations of Motion

The motion of particles in the simulation is governed by equation 2.7. This is derived from Newton's Second Law of motion. The total force, $\mathbf{F}$ is a combination of forces as shown in equation 2.8. $\mathbf{F}_n$, $\mathbf{F}_t$, and $\mathbf{F}_c$ are the forces defined in section 2.2. $\mathbf{F}_g$ and $\mathbf{F}_d$ are the gravitational force and drag force, respectively, and are defined in section 2.3.1.

$$\frac{d\mathbf{x}}{dt} = \mathbf{u} \tag{2.6}$$

$$\frac{du}{dt} = \frac{\mathbf{F}}{m} \tag{2.7}$$

$$\mathbf{F} = \mathbf{F}_n + \mathbf{F}_t + \mathbf{F}_c + \mathbf{F}_g + \mathbf{F}_d \tag{2.8}$$

### 2.3.1 Drag and Gravitational Forces

Assuming low Reynolds numbers, Stokes flow relationships can be used. This includes Stokes drag, defined in Equation 2.9, where $m$ is particle mass, $\tau_p$ is particle relaxation time, $\mathbf{u}_f$ is fluid velocity, and $\mathbf{u}$ is particle velocity.

$$\mathbf{F}_d = \frac{m}{\tau_p}(\mathbf{u}_f - \mathbf{u}) \tag{2.9}$$

The particle relaxation time, $\tau_p$ is an approximate timescale describing how the particle's velocity changes in a fluid due to drag. Assuming Stokes Law drag on a spherical particle, $\tau_p$ can be defined by equation 2.10 where $\rho_p$ is the particle density, $d_p$ is the particle diameter, and $\mu$ is the dynamic viscosity of the fluid.

$$\tau_p = \frac{\rho_p d_p^2}{18\mu} \tag{2.10}$$

The gravity force is simple and defined in equation 2.11.

$$\mathbf{F}_g = m\mathbf{g} \tag{2.11}$$

## 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. Consider 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 2.12. This provides a relationship between the force in a collision and the relative acceleration as a result of that collision. 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 2.13). This reduced mass can be used instead of particle mass in equation 2.7 when considering relative motion of particles.

$$F_1 = m_1 a_1$$
$$F_2 = m_2 a_2$$
$$F_1 = -F_2$$
$$a_{rel} = a_2 - a_1$$
$$= \frac{F_2}{m_2} - \frac{F_1}{m_1}$$
$$= \frac{F_2}{m_2} + \frac{F_2}{m_1}$$
$$= F_2 \Big( \frac{1}{m_2} + \frac{1}{m_1} \Big)$$
$$= F_2 \Big( \frac{m_1 + m_2}{m_1 m_2} \Big)$$
$$F_2 = \Big( \frac{m_1 m_2}{m_1 + m_2} \Big) a_{rel} \tag{2.12}$$
$$m_{reduced} = \frac{m_1 m_2}{m_1 + m_2} \tag{2.13}$$

It should be noted that equation 2.13 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 3.2).

### 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 won't 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 2.14.

$$t_{col} = \sqrt{\frac{m}{k_e}} \sqrt{\pi^2 + ln(\epsilon)^2} \tag{2.14}$$

### 2.4.3 Coefficient of Restitution

The coefficient of restitution is 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 Appendix A and is related to the damping coefficient, $\eta$, by equation 2.15. This is derived in section A.1.2.

$$\eta = -2ln(\epsilon) \sqrt{\frac{m k_e}{\pi^2 + ln(\epsilon)^2}} \tag{2.15}$$

## 2.5 Particle Rotation

## 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 determines which particles could collide with which other particles. This reduces the number of particle collisions that have to be calculated. The second phase, collision resolution, resolves the collisions by measuring overlap. The goal of the broad phase is to allow the simulation

to run in $O(N)$ time.

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 statistical distributions of particle 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.

# 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 2.7 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 2.8). 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 and so equation 2.8 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.

$$F(u_{n+1}) = F_{fixed} + \frac{m}{\tau_p}(u_f - u_{n+1})$$

$$u_{n+1} = u_n + \frac{F_{fixed} + \frac{m}{\tau_p}(u_f - u_{n+1})}{m}\Delta t$$

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

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

$$u_{n+1} = \frac{\tau u_n + F_{fixed}\tau\Delta t/m + u_f\Delta t}{\tau + \Delta t} \tag{3.7}$$

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

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

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

### 3.1.2  Position

Equation 2.6 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 a bit higher than first order accurate. This means that it is better to use the implicit method than the explicit method.
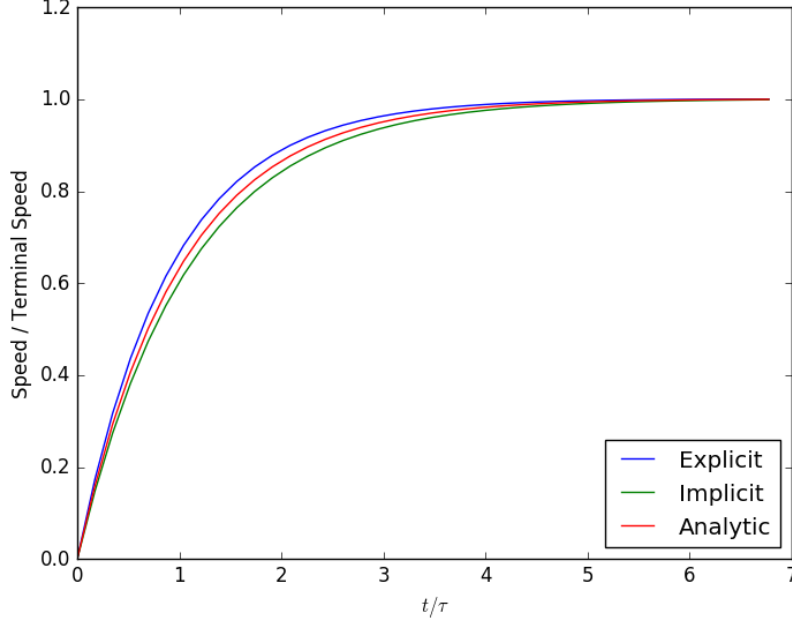
**Figure 3.1:** Particle speed against time with different integration schemes.
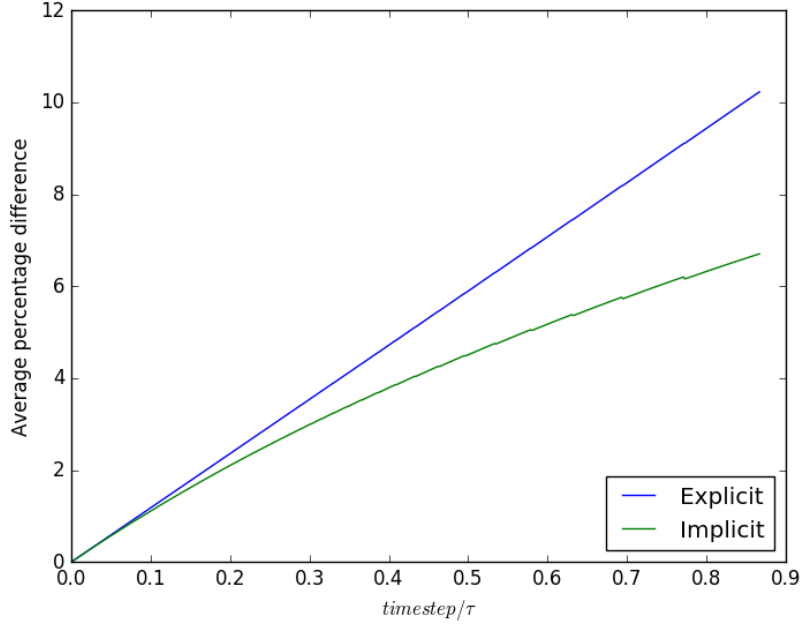


**Figure 3.2:** Average percentage difference between the numerical method and analytical solution against varying timestep. The jaggedness of the lines is due to the discrete changes in timestep causing small rounding errors.

## 3.2 Verification

### 3.2.1 Drag

The drag verification case is a particle in a fluid. The fluid has a constant velocity in the x direction denoted $u_f$. 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.

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

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

$$\tag{3.12}$$

### 3.2.2 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 3.3.

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.

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

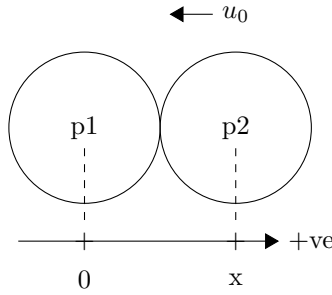$$\dot{x} = u_0 e^{at} (\frac{a}{b} sin(bt) + cos(bt)) \tag{3.14}$$



**Figure 3.3:** The initial setup of the normal collision verification case.

### 3.2.3 Friction Sliding

The friction sliding test case is ...
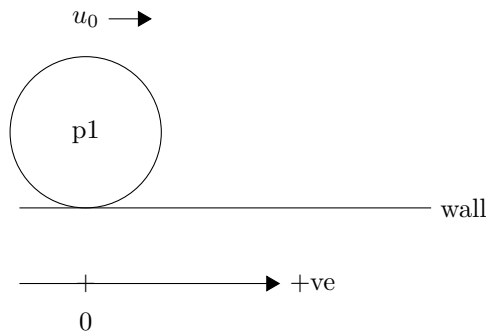


**Figure 3.4:** The initial setup of the friction sliding verification case.

### 3.2.4 Normal Collision with Cohesion

The cohesion collision verification case is very similar to the normal collision verification case (section 3.2.2. 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 3.5.

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.2.

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

$$x = u_0 \sqrt{\frac{m}{k_c}} sinh\left(t\sqrt{\frac{k_c}{m}}\right) + d_e \tag{3.15}$$

$$\dot{x} = u_0 cosh\left(t\sqrt{\frac{k_c}{m}}\right) \tag{3.16}$$

When $x < d_b$:

$$x = e^{at}\left(\frac{u_i - ac}{b}sin(bt) + ccos(bt)\right) + \frac{k_e d_b - k_c d_e}{k_e - k_c} \tag{3.17}$$

$$\dot{x} = e^{at}\left(\left(\frac{u_i - ac}{b}a - cb\right)sin(bt) + u_i cos(bt)\right) \tag{3.18}$$

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

$$x = (d_b - d_e)cosh\left(\sqrt{\frac{k_c}{m}}t\right) + u_r\sqrt{\frac{m}{k_c}}sinh\left(\sqrt{\frac{k_c}{m}}t\right) + d_e \tag{3.19}$$

$$\dot{x} = \sqrt{\frac{k_c}{m}}(d_b - d_e)sinh\left(\sqrt{\frac{k_c}{m}}t\right) + u_r cosh\left(\sqrt{\frac{k_c}{m}}t\right) + d_e \tag{3.20}$$

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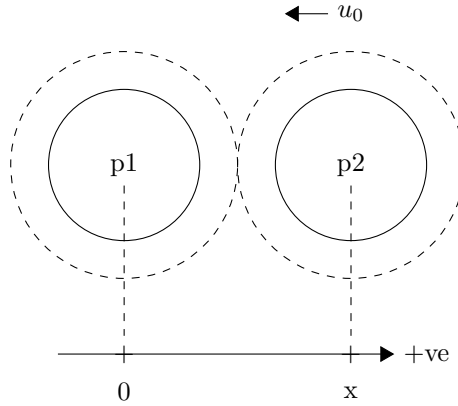


**Figure 3.5:** The initial setup of the cohesion collision verification case.

16

# Chapter 4

# Python Implementation

## 4.1 Overview

An initial implementation of the Discrete Element Method has been done 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 keeping track of everything very easy. Objects have been used extensively throughout the Python implementation.

#### 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 a variety of 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.
The particle is iterated using the function call 'Particle.iterate()'. This passes a $\Delta t$ to the particle object and iterates the velocity and position. The iterative functions used are discussed in Chapter 3.

#### 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.

### 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 is 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.
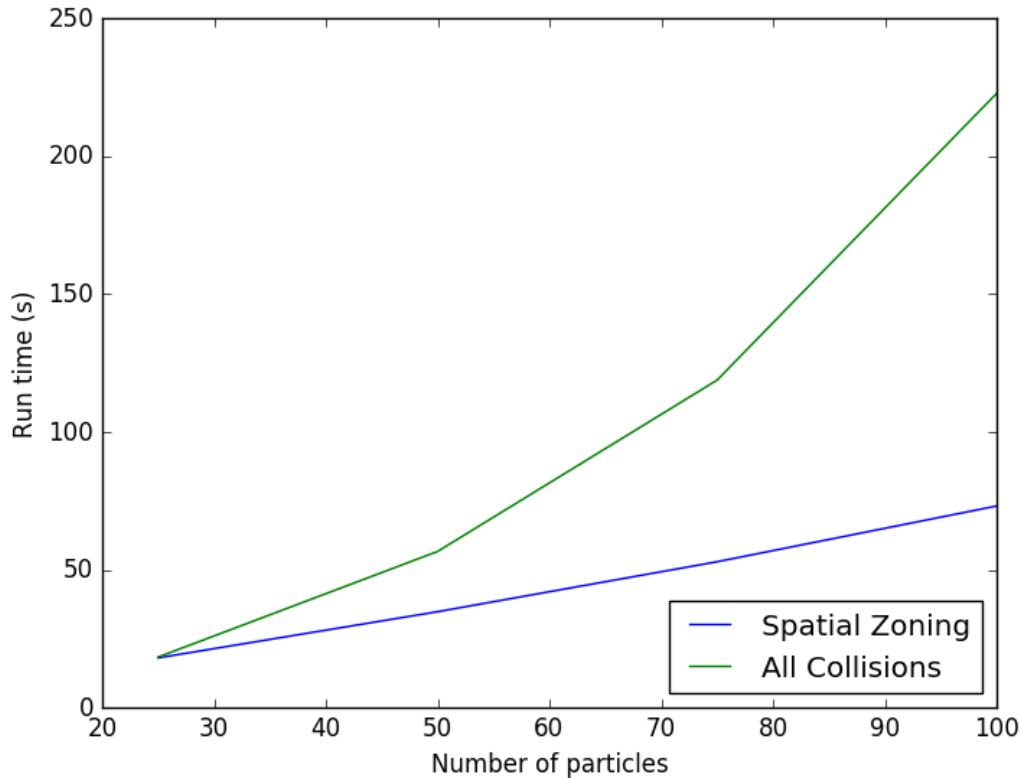


**Figure 4.1:** This graph shows that the simple spatial zoning technique reduces the problem from $O(N^2)$ down to almost $O(N)$.

### 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 -9.81$ms^{-2}$ in the negative 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 don't 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 Optimization and Performance
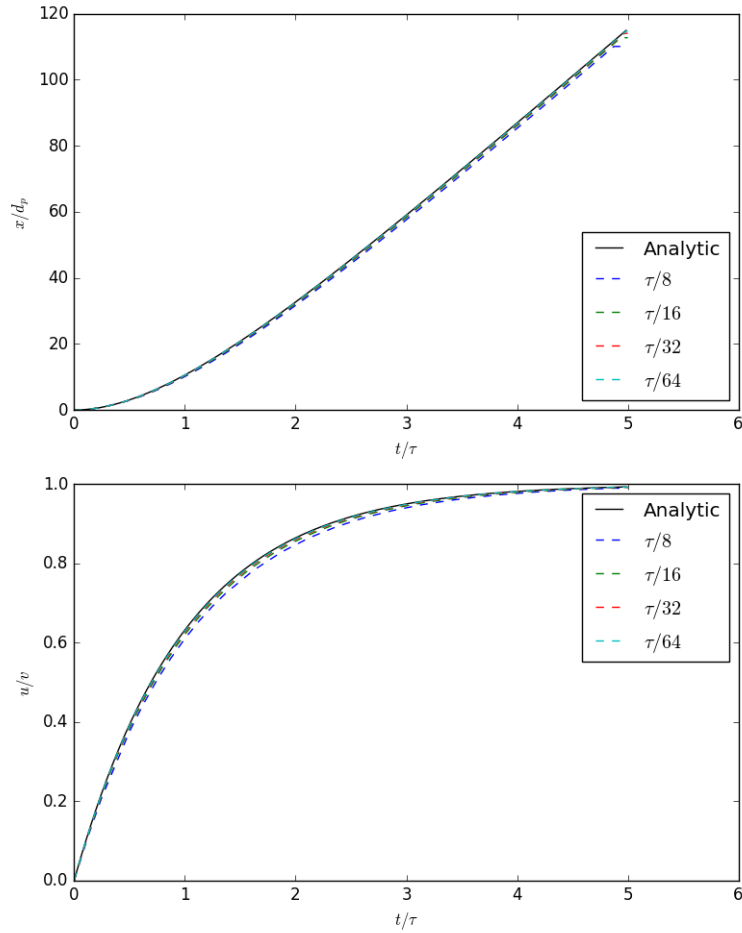
## 4.4 Verification

### 4.4.1 Drag



**Figure 4.2:** Normalized position and speed against time of a particle in a moving fluid.
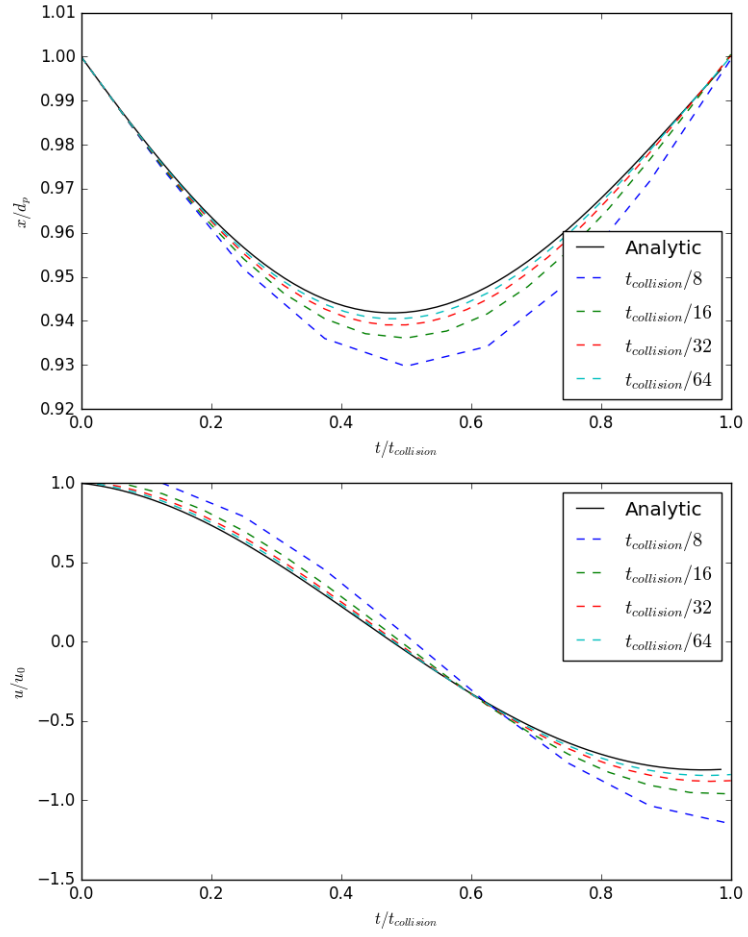
### 4.4.2 Normal Collision

**Figure 4.3:** Normalized position and speed against time during a normal collision.
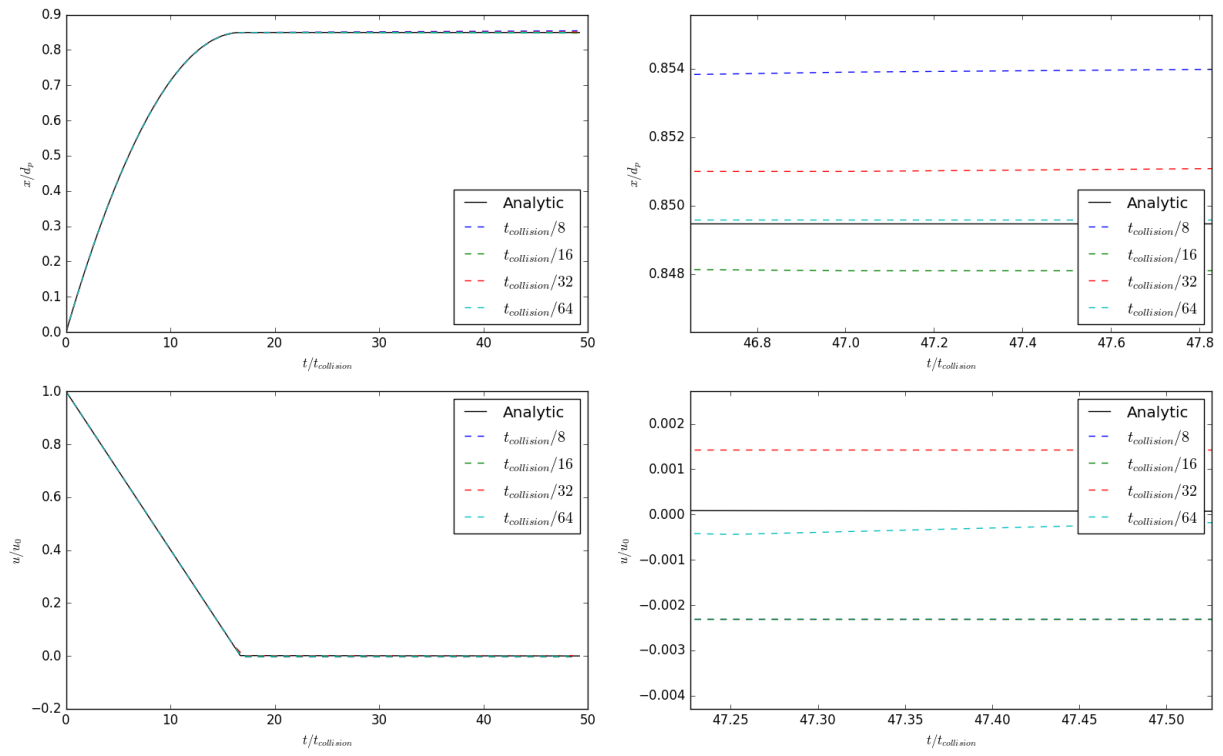
### 4.4.3 Friction Sliding

**Figure 4.4:** Normalized position and speed against time during a normal collision.

# 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 requires 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.

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. Often the device would be a GPU or other, more novel, processing units. 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.

## 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 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. 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 likely 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.

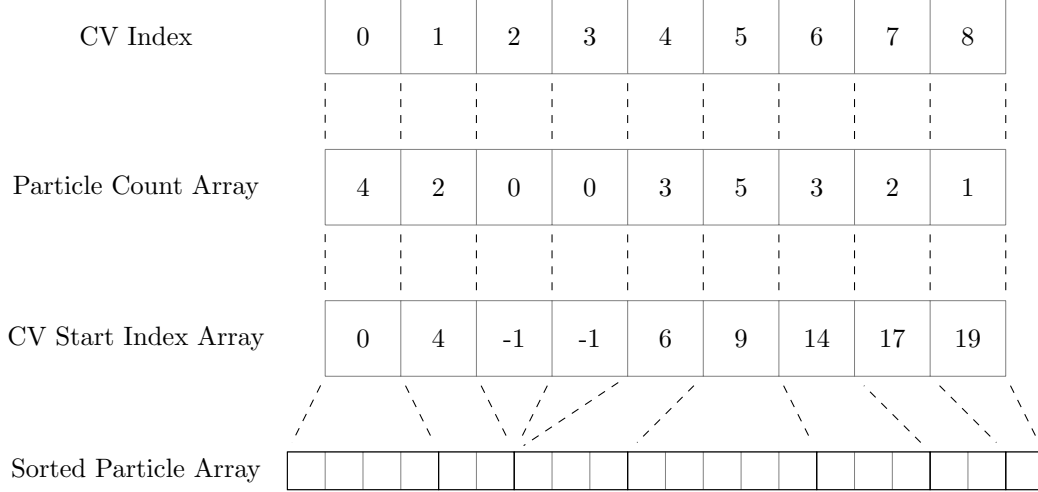| CV Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| Particle Count Array | 4 | 2 | 0 | 0 | 3 | 5 | 3 | 2 | 1 |
| CV Start Index Array | 0 | 4 | -1 | -1 | 6 | 9 | 14 | 17 | 19 |

Sorted Particle Array

**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. A solution to this weakness could be to have each control volume add up the number of particles in all preceding control volumes in order to determine its own start index in the sorted particle array. However, this would repeat a lot of maths and could end up being slower. The function consists of adding to the start index count and then assigning the count to memory in the CV start index array. If it were to be done in parallel, the adding would have to be repeated for each CV but the memory assignment would be done in parallel. Thus, whether it would be faster sequentially or in parallel depends on which operation is slower. 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. 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[14]. The code is shown in figure 5.2. 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 doesn't 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.

```
_INLINE_ void atomicAdd_g_f(volatile __global float *addr, float val)
{
    union{
        unsigned int u32;
        float        f32;
    } next, expected, current;
     current.f32    = *addr;
    do{
        expected.f32 = current.f32;
        next.f32     = expected.f32 + val;
            current.u32  = atomic_cmpxchg( (volatile __global unsigned int *)addr,
                            expected.u32, next.u32);
    } while( current.u32 != expected.u32 );
}
```

**Figure 5.2:** Kernel code for a solution to atomic addition with floats.

**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.

**Testing Framework**

Often a framework is used for unit testing, however there isn't 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

*Each of the subsections in this section will have details of the equations used and the significance of the results.*
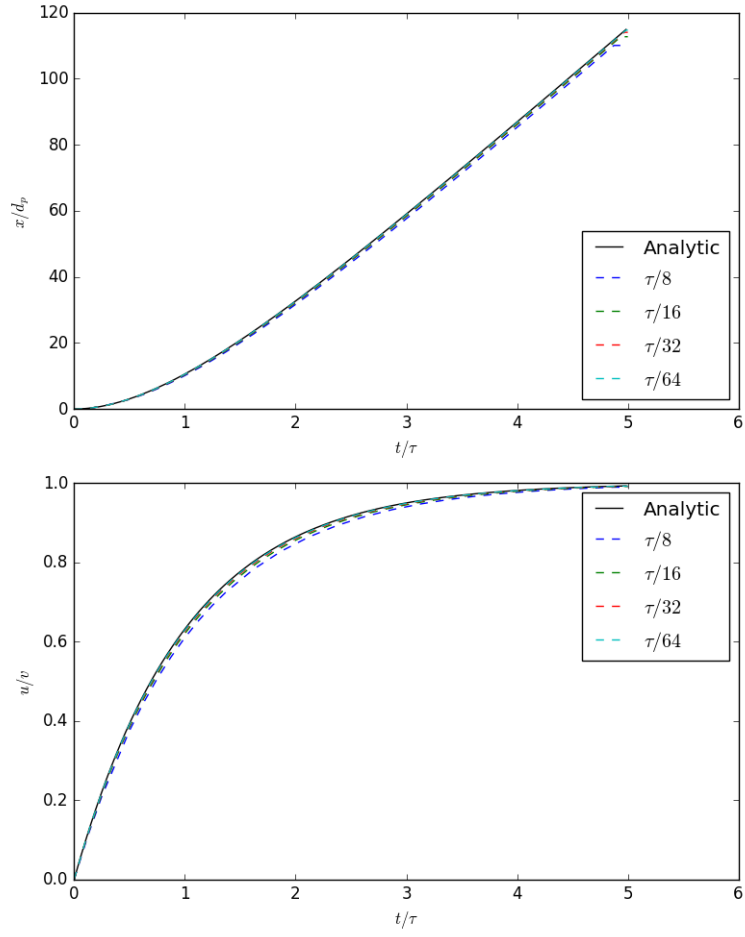
### 5.4.1   Drag

**Figure 5.3:** Normalized position and speed against time of a particle in a moving fluid.
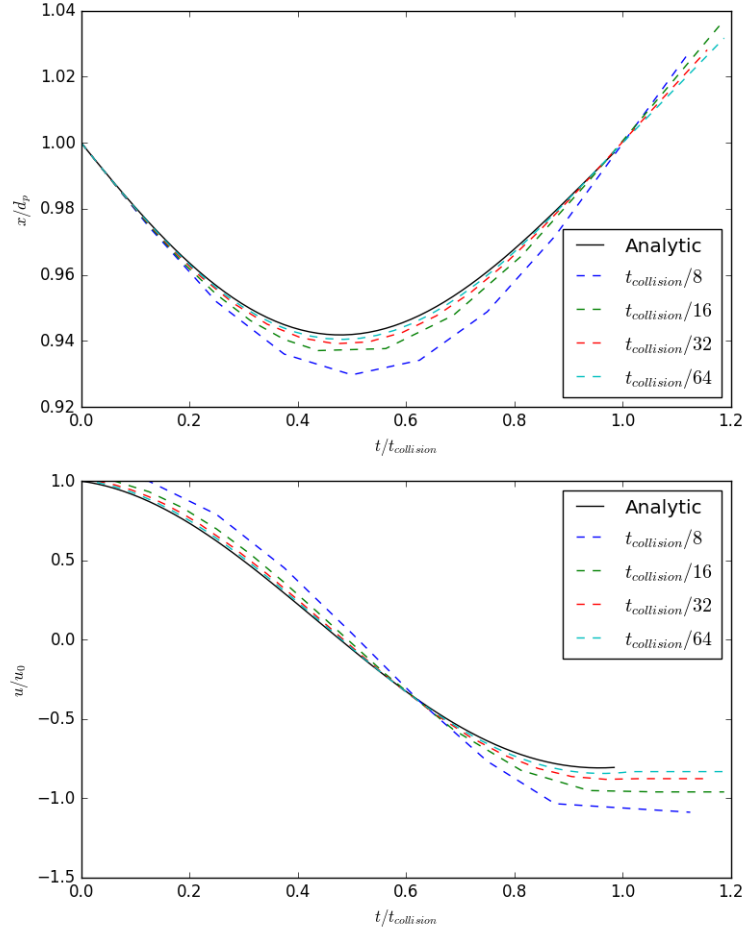
## 5.4.2 Normal Collision

**Figure 5.4:** Normalized position and speed against time during a normal collision.

### 5.4.3   Friction Sliding

The analytic solution used in 5.5 is a dynamic Coulomb model friction only. Note that the error increases as the timestep decreases. This is caused by the static friction being lower than the dynamic friction and so not slowing the particle down as much. If the static friction part of the model is removed then the results are those shown in 5.6. This very closely matches the analytic solution but is less stable.
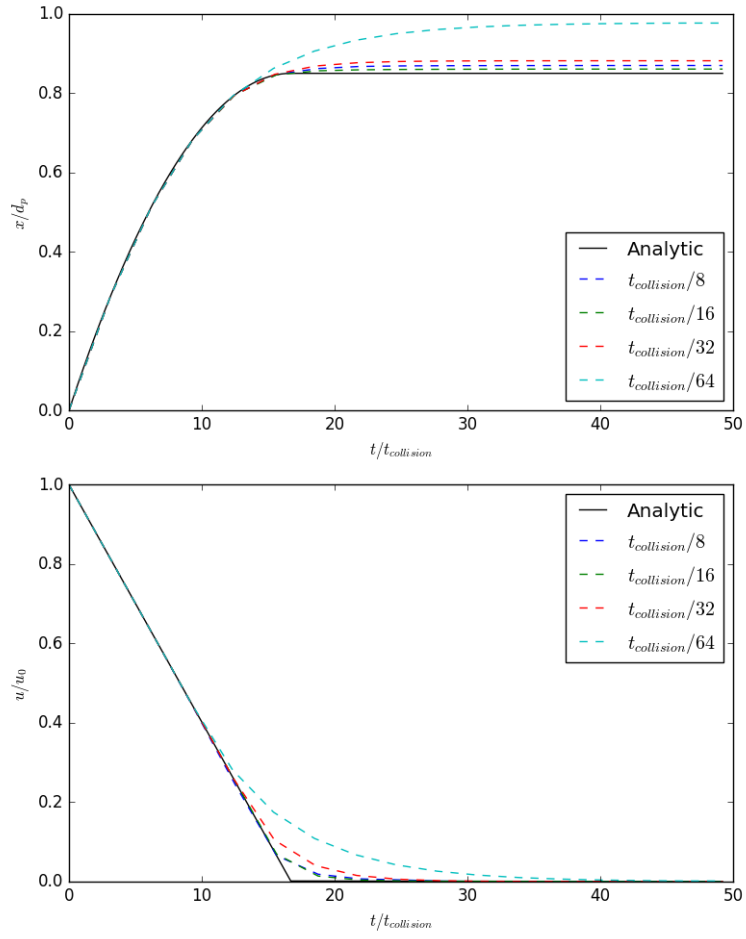
**Figure 5.5:** Normalized position and speed against time during a friction event.
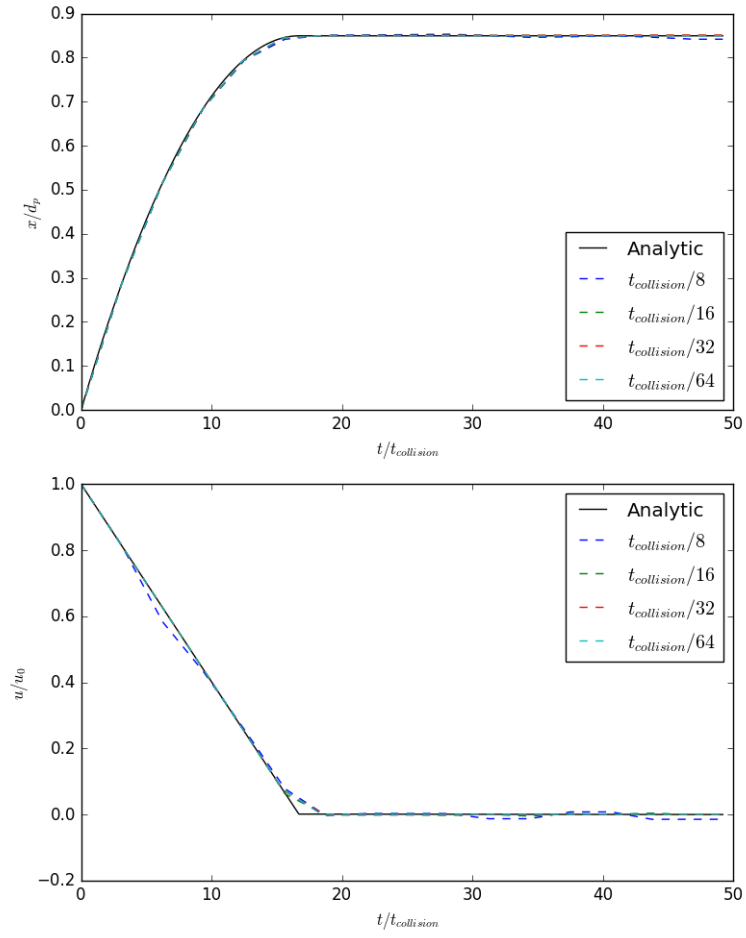
**Figure 5.6:** Normalized position and speed against time during a friction event with only dynamic friction enabled.
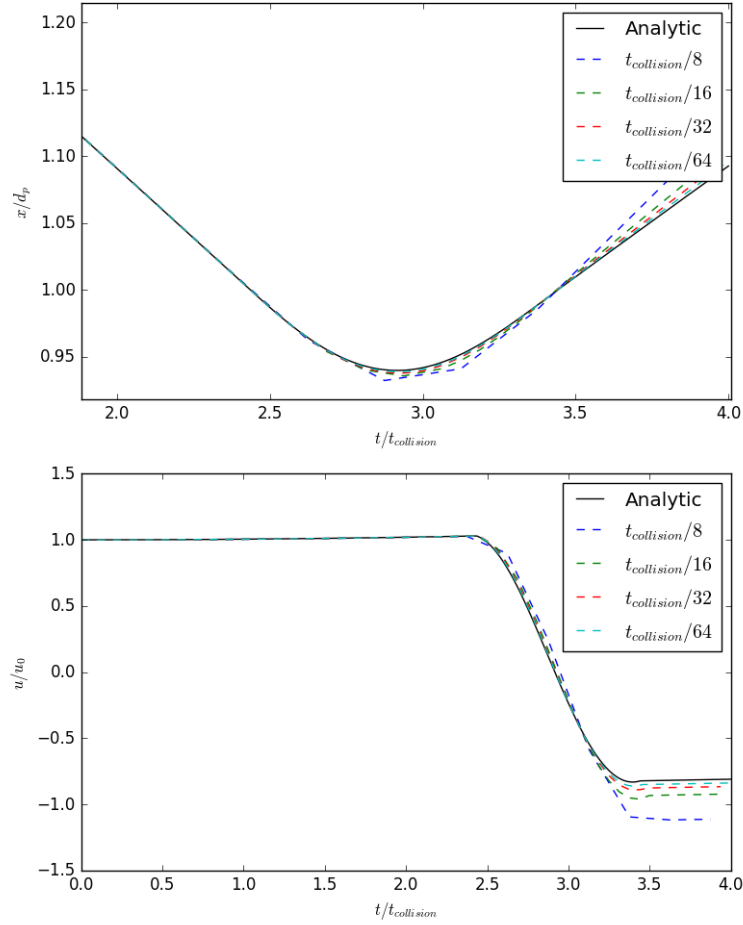
### 5.4.4 Normal Collision with Cohesion



**Figure 5.7:** Normalized position and speed against time during a normal collision with cohesion.

# 5.5 Optimization and Performance

### 5.5.1 Optimizations

### 5.5.2 Performance

Govender et al. have also developed GPU based DEM implementations and have provided performance measures in one of their papers[15]. 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.[15] 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 $11.5 \times 10^6$.

The table in Figure 5.8 compares this project with Govender et al.[15]. 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 | $10 \times 10^6$ | $11.5 \times 10^6$ |
| Govender et al.[15] | Sphere | $50 \times 10^6$ | $55 \times 10^6$ |
| BLAZE-DEM[3] | Sphere | $32 \times 10^6$ | $100 \times 10^6$ |

**Figure 5.8:** 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 assess the algorithm performance only. Figure 5.9 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.
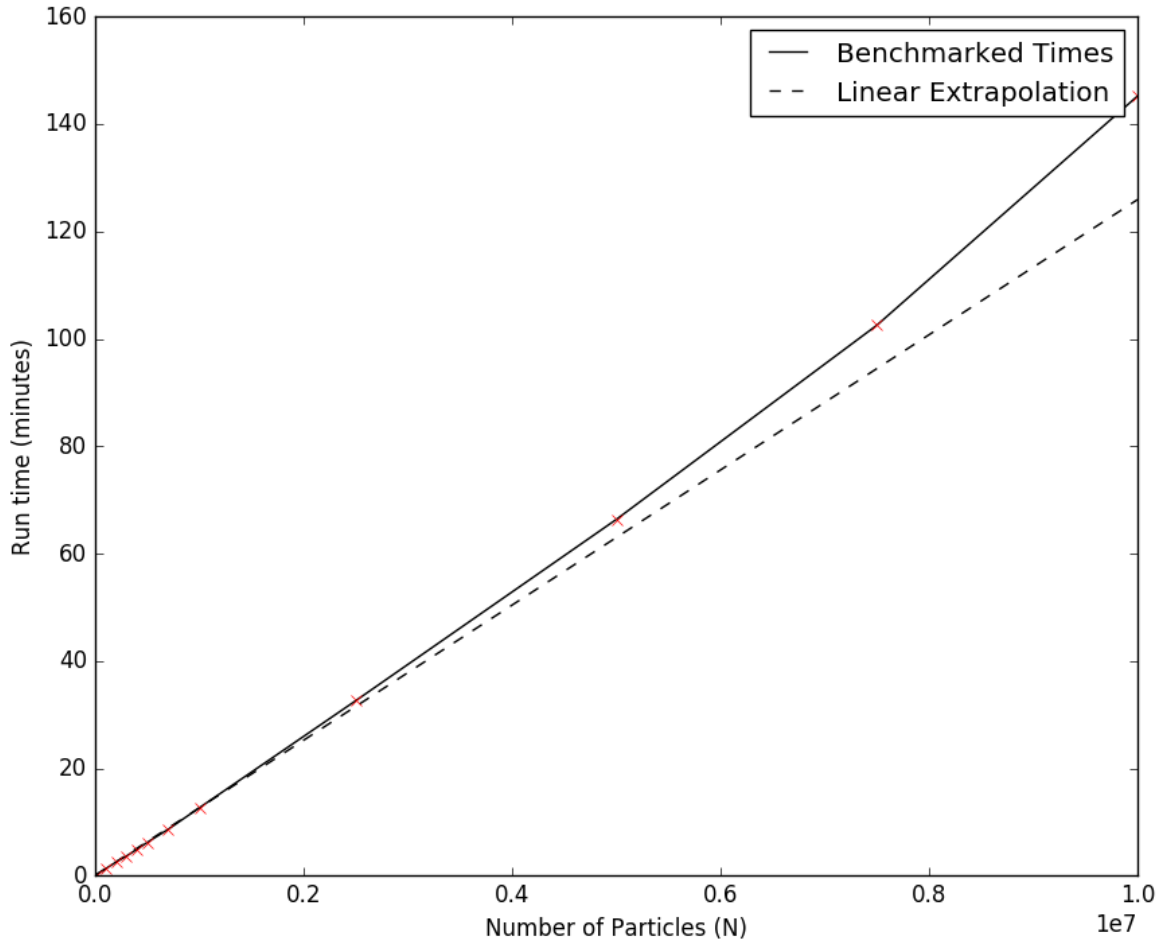


**Figure 5.9:** Run time against number of particles for the OpenCL simulation compared to a linear extrapolation through $N = 1e4$ and $N = 1e6$.

# Chapter 6

# Application

*General overview of agglomerates and why different properties are interesting.*

## 6.1 Simulation Setup

*Initial conditions, properties to be varied, how this is defined in the program.*

### 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 $\pi$. The flow is plotted in Figure 6.1.

$$u = A\cos\left(a\left(x + \frac{\pi}{2a}\right)\right)\sin\left(a\left(y + \frac{\pi}{2a}\right)\right)\sin\left(a\left(z + \frac{\pi}{2a}\right)\right) \tag{6.1}$$

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

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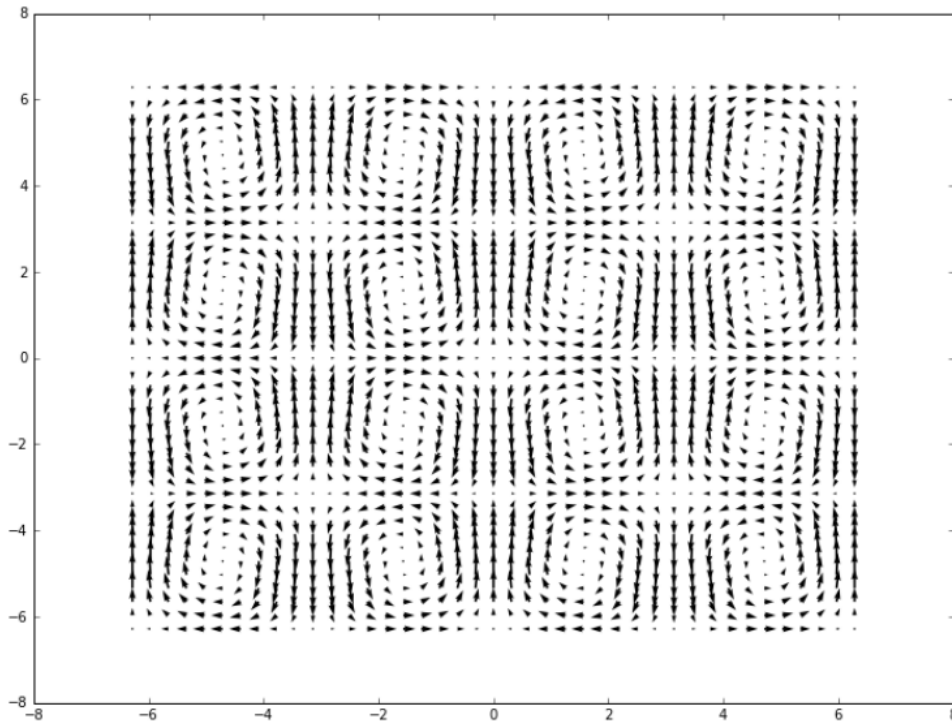


**Figure 6.1:** A z-y view of a Taylor-Green Vorted flow field. (Figure 5a from Chow[1]).

### 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 $u_{f,avg} = 0.7839A$.
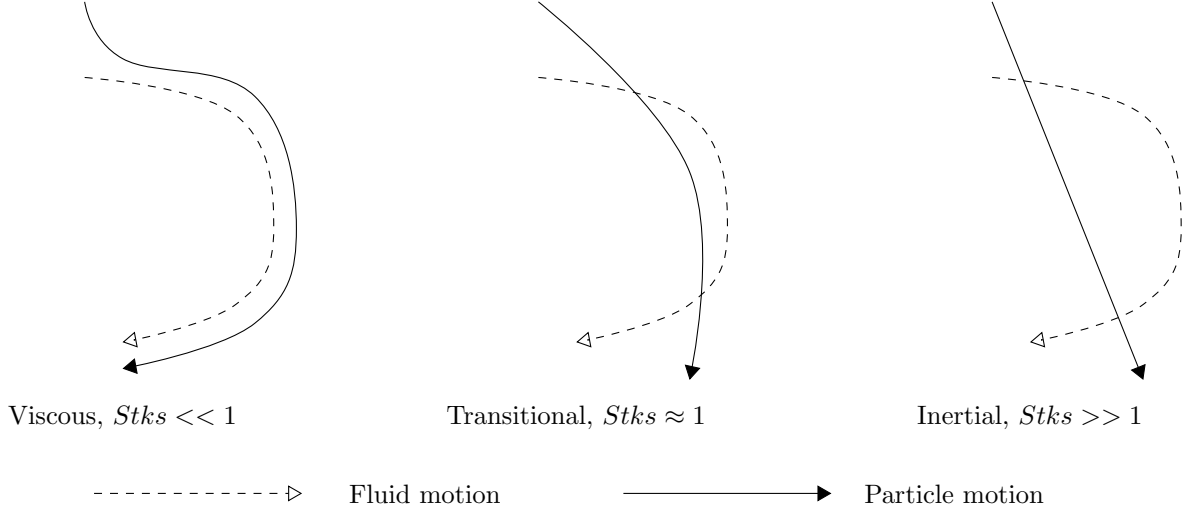
$$Stk = \frac{\rho_p d_p^2 u_0}{18\mu l_0} \tag{6.4}$$



Viscous, $Stks << 1$       Transitional, $Stks \approx 1$       Inertial, $Stks >> 1$

- - - - - - - - - - ▷   Fluid motion       ───────────▶   Particle motion

**Figure 6.2:** Stoke's Number regimes.

### 6.1.3 Stickyness Number

The Stickyness 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 2.5. $\epsilon$ 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 *<TODO: Add section for this in appendix or verification>*. When the effect length is $d_e - d_b$ and $u$ is the initial collision speed, the Stickyness Number determines whether the particles will stick after the collision.

$$Sy < 1 \ Does \ not \ stick$$
$$Sy > 1 \ Sticks$$

## 6.2 Results

*Presentation of results, graphs etc.*

## 6.3 Discussion

*Discussion of results including key relationships and conclusions.*

# Chapter 7

# Conclusion

*Overall conclusions from the project. This will likely include some discussion of the successes of the programs as well as a summary of the application results.*

## 7.1 Further Work

*Recommendations for possible further work from this project.*

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

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

**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.

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

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

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

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

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

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

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

$n = 0$ 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, $\epsilon$, 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 $\epsilon$:

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

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

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

$\eta$ 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 ($\delta_b$) and effect overlap ($\delta_e$).

$$\delta_b = d_b - x$$
$$\delta_e = d_e - x$$

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

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

### Complementary Function

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

$$p = \pm\sqrt{\frac{k_c}{m}}$$

$$x = A_1e^{t\sqrt{k_c/m}} + A_2e^{-t\sqrt{k_c/m}}$$

### Particular Integral

Ansatz: $x = B$

$$\dot{x} = \ddot{x} = 0$$
$$-k_cx = -k_cd_e$$
$$-k_cB = -k_cd_e$$
$$B = d_e$$

### General Solution

$$x = A_1e^{t\sqrt{k_c/m}} + A_2e^{-t\sqrt{k_c/m}} + d_e$$

$$\dot{x} = \sqrt{\frac{k_c}{m}}A_1e^{t\sqrt{k_c/m}} - \sqrt{\frac{k_c}{m}}A_2e^{-t\sqrt{k_c/m}}$$

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

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

$$A_1 = \frac{u_0}{2}\sqrt{\frac{m}{k_c}}$$

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

$$x = u_0\sqrt{\frac{m}{k_c}}sinh\left(t\sqrt{\frac{k_c}{m}}\right) + d_e$$

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

$$\dot{x} = u_0cosh\left(t\sqrt{\frac{k_c}{m}}\right)$$

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

**Impact Time ($t_i$)**

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 ($u_i$)**

$$u_i = \frac{u_0}{2}e^{t_i\sqrt{k_c/m}} + \frac{u_0}{2}e^{-t_i\sqrt{k_c/m}}$$

$$u_i = u_0cosh\left(t_i\sqrt{\frac{k_c}{m}}\right)$$

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

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

**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: $c = \dfrac{k_c(d_e - d_b)}{k_e - k_c}$

$\dot{x}(0) = aA_2 + bA_1 = u_i$

$$A_1 = \frac{u_i - aA_2}{b} = \frac{u_i - ac}{b}$$

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

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

The position cannot be analytically solved for $t$ in order to find the initial return velocity. A reasonable estimation for this is to use $t_{col} = \dfrac{\pi}{b}$, 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.

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

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

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

$$A_1 = \frac{1}{2}\left(d_b - d_e + u_r\sqrt{\frac{m}{k_c}}\right)$$

$$A_2 = \frac{1}{2}\left(d_b - d_e - u_r\sqrt{\frac{m}{k_c}}\right)$$

$$A_1 = \frac{u_0}{2}\sqrt{\frac{m}{k_c}}$$

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

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

## A.3   Stickyness 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:

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

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

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

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

Thus, for a particle to stick in a collision:

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

$$u_r = -\epsilon u_i$$

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

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

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

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

$$F_n = mg$$

$$F_t^{dynamic} = -\mu|F_n|$$

$$\frac{du}{dt} = \ddot{x} = \frac{-\mu mg}{m} = -\mu g$$

$$\dot{x} = -\mu gt + u_0$$

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

## 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, 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] Anca Hamuraru. Atomic operations for floats in opencl - improved, 2016.

[15] 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).