

Contents

Abstract	3
Nomenclature	4
1 Introduction	5
1.1 Previous Work	5
1.2 Single Particle Motion in Fluids	5
1.3 Project Goals	5
2 The Discrete Element Method	6
2.1 Particle Definition	6
2.2 DEM Forces	6
2.2.1 Normal Contact Force	6
2.2.2 Tangential Contact Force	7
2.2.3 Cohesion Force	7
2.3 Equations of Motion	7
2.3.1 Drag and Gravitational Forces	7
2.4 Rotation/Quaternions	7
2.5 Collision Detection	8
3 Numerical Methods	9
3.1 Implicit/Explicit	9
3.2 Numerical Integration	9
3.2.1 Velocity	9
3.2.2 Position	9
3.2.3 Method of Integrating Drag	9
4 Python Implementation	12
4.1 Overview	12
4.2 Element Types	12
4.2.1 Particle	12
4.2.2 Axis-Aligned Simple Wall	12
4.3 Collisions	12
4.3.1 Collision Detection	12
4.3.2 Collision Resolution	13
4.4 Calculating Forces	13
4.4.1 Drag	13
4.4.2 Gravity	13
4.4.3 DEM Forces	14
4.5 Verification	14
4.5.1 Settling Overlap	14
4.5.2 Timestep Stability	14
5 OpenCL and Graphics Processing Units	16

6	OpenCL Implementation	17
6.1	Data Structures	17
6.1.1	Particle	17
6.1.2	Collision	17
6.1.3	Buffers	17
6.2	Kernels	18
6.2.1	Collision Detection	18
6.2.2	Collision Resolution	19
6.2.3	Particle Iteration	19
6.3	Unit Tests	20
6.3.1	Testing Framework	20
6.4	Verification	20
6.4.1	Normal Force	20
6.4.2	Friction Force	20
7	Results	23
7.1	Comparison between CPU and GPU	23
8	Application	24
8.1	Modified Particle Structure	24
8.2	Fluid Velocity Field	24
8.3	Results	24
9	Conclusion	25
9.1	Further Work	25
A	Derivations	26
A.1	Equation 3.9	26

Abstract

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
μ	Dynamic Viscosity
μ	Friction Coefficient
ρ_p	Particle Density
τ_p	Particle Relaxation Time
ζ	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 Speed

Chapter 1

Introduction

1.1 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.

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.

< A lot of this was from interim report. I'm concerned that it may be too much detail on previous DEM implementations before DEM is explained properly. >

1.2 Single Particle Motion in Fluids

The focus of the previous project was simulating how a particle moves in a fluid. The governing equations from that project form the basis of this project. The governing equation of motion was equation 1.1 where \mathbf{u} is the particle velocity, \mathbf{u}_f is the fluid velocity at the particle location, τ_p is the particle relaxation time, and \mathbf{g} is the gravitational acceleration.

$$\frac{d\mathbf{u}}{dt} = \frac{1}{\tau_p}(\mathbf{u} - \mathbf{u}_f) + \mathbf{g} \quad (1.1)$$

The particle relaxation time, τ_p is an approximate timescale describing how the particle's velocity changes in a fluid due to drag. Assuming Stoke's Law drag on a spherical particle, τ_p can be defined by equation 1.2 where ρ_p is the particle density, d_p is the particle diameter, and μ is the dynamic viscosity of the fluid.

$$\tau_p = \frac{\rho_p d_p^2}{18\mu} \quad (1.2)$$

1.3 Project Goals

Thus the aim of this project is to implement the DEM for large numbers of particles on OpenCL. This will allow for an assessment of OpenCL performance with large numbers of particle that has not yet been tested.

< Currently just a copy-paste from interim report, this will be expanded on and more clearly defined later on. >

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.

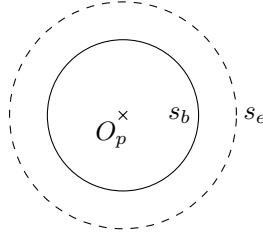


Figure 2.1: Definition of a particle.

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, 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.

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

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, ζ is the tangential displacement during the interaction, and $\hat{\mathbf{t}}$ is the unit vector tangential to the collision.

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

The dynamic regime friction force is calculated with equation 2.3 where μ is the coefficient of friction.

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

The final tangential friction force is defined in equation 2.4.

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

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 δ_e is the particle effect surface overlap.

$$\mathbf{F}_c = k_c \delta_e \hat{\mathbf{n}} \quad (2.5)$$

2.3 Equations of Motion

The motion of particles in the simulation is governed by equation 2.6. This is derived from Newton's Second Law of motion. The total force, \mathbf{F} is a combination of forces as shown in equation 2.7. \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.

$$\frac{d\mathbf{u}}{dt} = \frac{\mathbf{F}}{m} \quad (2.6)$$

$$\mathbf{F} = \mathbf{F}_n + \mathbf{F}_t + \mathbf{F}_c + \mathbf{F}_g + \mathbf{F}_d \quad (2.7)$$

2.3.1 Drag and Gravitational Forces

The drag force is the same as that used in the previous project. Equation 2.8 defines the drag force from the acceleration used in equation 1.1.

$$\mathbf{F}_d = \frac{m}{\tau_p} (\mathbf{u}_f - \mathbf{u}) \quad (2.8)$$

The gravity force is also the same as that used in the previous project and is defined in equation 2.9.

$$\mathbf{F}_g = m\mathbf{g} \quad (2.9)$$

2.4 Rotation/Quaternions

<This will be completed later as it is not immediately relevant to the progress of the project.>

2.5 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 4.3.1 and 6.2.1 provide further detail as to how this is implemented computationally in Python and with OpenCL, respectively.

Chapter 3

Numerical Methods

3.1 Implicit/Explicit

3.2 Numerical Integration

3.2.1 Velocity

Velocity is iterated with equation 3.1 where \dot{u} is the acceleration obtained from the function call 'Particle.get_accel()'.
`Particle.get_accel()`.

$$u_{n+1} = u_n + \dot{u}\Delta t \quad (3.1)$$

3.2.2 Position

Position is iterated with equation 3.2.

$$x_{n+1} = x_n + \frac{u_{n+1} + u_n}{2} \Delta t \quad (3.2)$$

3.2.3 Method of Integrating Drag

There are three different methods of integrating drag. Firstly, there is the analytical solution, this is the exact solution to the model but cannot be done easily computationally. Secondly, there is the explicit numerical solution, this takes the current state and estimates the future state. Thirdly, there is the implicit numerical solution, this assumes the future state and integrates accordingly. For a simple system with constant flow speed and no gravity, the system acceleration is described by equation 3.3.

$$\dot{u} = \frac{v - u}{\tau} \quad (3.3)$$

Equation 3.3 can be solved to show that the analytical solution for the particle speed, u , is that in equation 3.4.

$$u(t) = v(1 - e^{-\frac{t}{\tau}}) \quad (3.4)$$

The explicit numerical integration method for equation 3.3 is that in equation 3.5. Where u_n is the current speed and u_{n+1} is the speed after timestep Δt .

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_n}{\tau} \quad (3.5)$$

The implicit numerical integration method for equation 3.3 is that in equation 3.6. Where u_n is the current speed and u_{n+1} is the speed after timestep Δt .

$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} \quad (3.6)$$

Equation 3.6 can be rearranged to get an equation of the form $\dot{u} = f(u_n)$ as shown in equation 3.7.

$$\dot{u} = \frac{v - u_n}{\tau + \Delta t} \quad (3.7)$$

When these three methods are applied to the system they produce the results in figure 3.1.

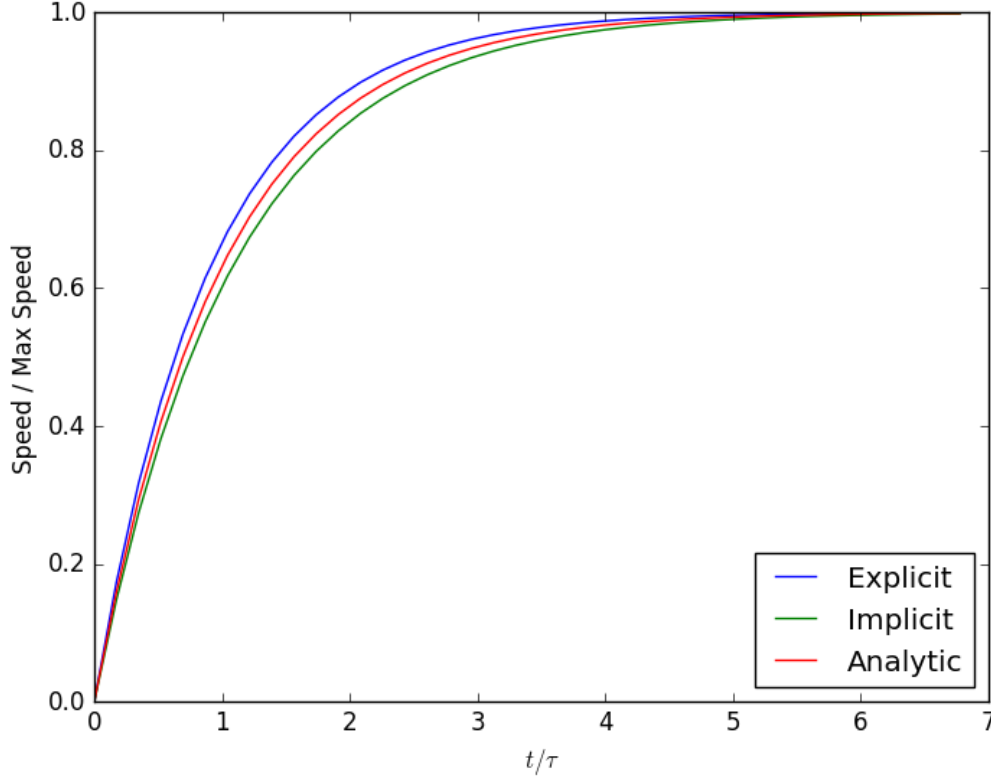


Figure 3.1: A graph of particle speed against time for the three methods of integration.

Comparing the explicit and implicit numerical integration to the analytical solution shows that the explicit method has an average percentage difference of 2.05% and the implicit method has an average percentage difference of 1.85% when the timestep is 0.1s.

The average percentage difference can be compared between the two methods with varying timesteps as shown in figure 3.2. This graph shows that the explicit method increases its average percentage difference approximately linearly with increasing timestep. The implicit method increases its average percentage difference non-linearly at a slower rate than the explicit method. This implies that the implicit method is more accurate than the explicit method, especially for higher timesteps.

Unlike the explicit method, the implicit method depends on the other accelerations in the system. Equation 3.7 can be redefined to include these other accelerations as an extra \dot{u}_e term as shown in equation 3.8.

$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} + \dot{u}_e \quad (3.8)$$

As before, equation 3.8 can be rearranged to be in the form $\dot{u} = f(u_n)$ as shown in equation 3.9. The derivation of this can be found in appendix A.1.

$$\dot{u} = \frac{v - u_n + \tau \dot{u}_e}{\tau + \Delta t} \quad (3.9)$$

Equation 3.9 can be applied to a particle falling under the effect of gravity through a stationary fluid. The results are shown in figure 3.3. As with the previous system, the explicit method has an average percentage difference of 2.05% and the implicit method has an average percentage difference of 1.85% when the timestep is 0.1s. This shows that the accuracy of the modified equation is consistent with that of equation 3.7.

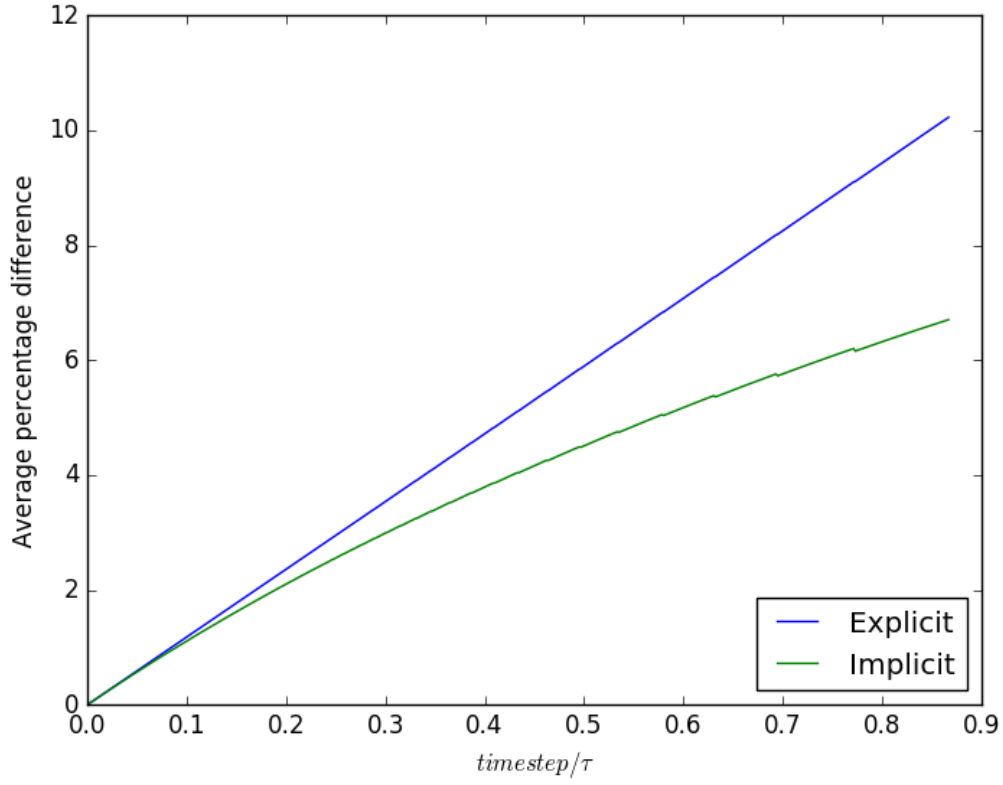


Figure 3.2: A graph of average percentage difference between the numerical method and analytical solution against varying timestep. The jaggedness of the lines is due to the discrete changes in timestep causing small rounding errors.

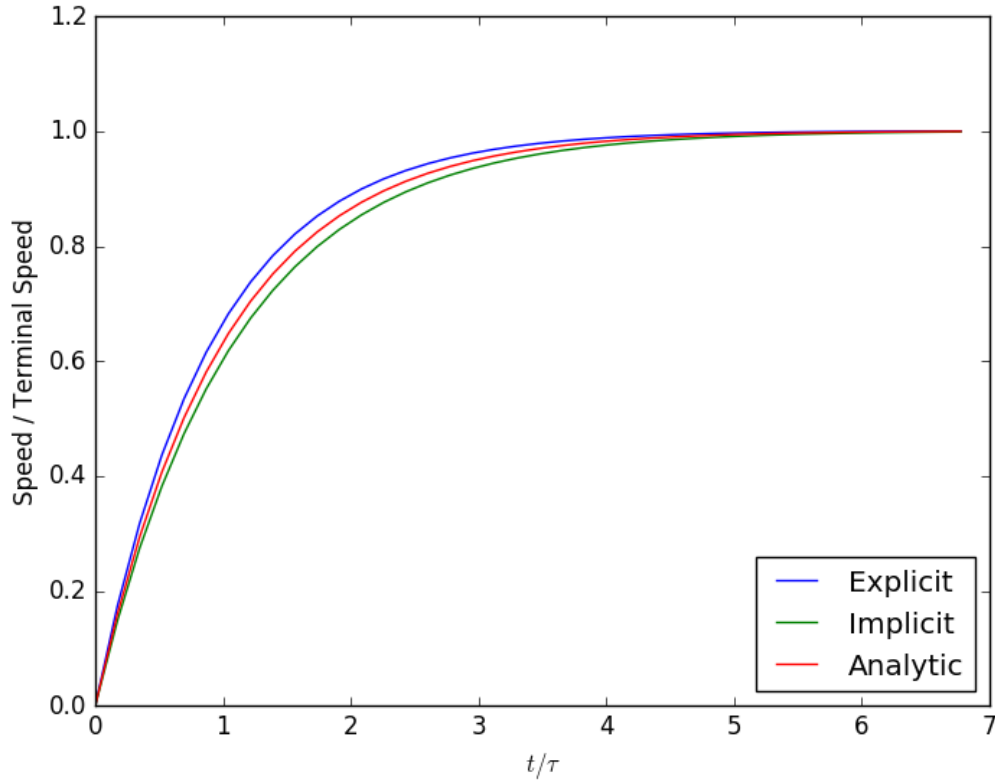


Figure 3.3: A graph of particle speed against time for each method of integration using equation 3.9.

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.

4.2 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.

4.2.1 Particle

The basic particle element is a spherical particle with pre-determined properties. These properties include initial position, initial velocity, diameter, density, fluid viscosity, and functions for fluid velocity and gravity. 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 Δt to the particle object and iterates the velocity and position. The methods used for integrating these properties are discussed in section 3.2.

4.2.2 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.

4.3 Collisions

4.3.1 Collision Detection

Broad phase collision detection uses the simple spatial zoning technique described in Tuley[10]. This approach has been chosen because it is quick and simple to implement. Other options were considered for this implementation but the benefits of using them were far outweighed by the complexity that using them would add to the overall algorithm. 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 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.

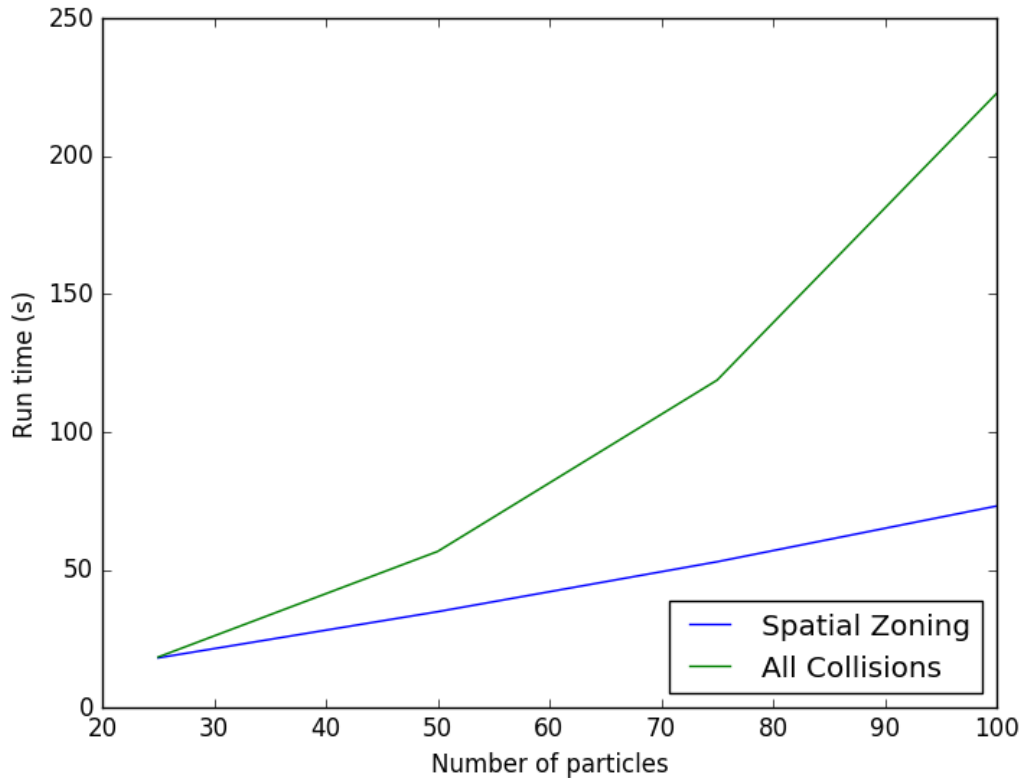


Figure 4.1: This graph shows that the simple spatial zoning technique reduces the problem from $O(N^2)$ down to almost $O(N)$.

4.3.2 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 this initial Python implementation only the simple normal and tangential contact forces are calculated. These are enough to run sufficient initial test cases.

4.4 Calculating Forces

There are three categories of forces used in this implementation. These are drag (particle-fluid interaction), gravity, and DEM forces.

4.4.1 Drag

The drag forces are determined using Stokes' law as discussed in This is calculated using a flow field calculation function that is passed into the Particle object upon instantiation. The particle then calls this function as part of its `get_accel()` function. 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.

4.4.2 Gravity

Gravity is treated in a similar manner to drag. A gravity function can be passed into the Particle object upon instantiation. Although this defaults to a simple $-9.81ms^{-1}$, 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.

4.4.3 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 added together and used in the integration 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.

4.5 Verification

To assess the accuracy of the implementation a series of cases have been tested and compared to analytical solutions of the model.

4.5.1 Settling Overlap

In this case two particles are used. The first particle is an ordinary particle acting under the effects of gravity. The second particle, placed below the first particle, is a particle with quasi-infinite density without being affected by gravity. As time increases the first particle bounces on the second particle until it eventually comes to rest with some overlap with the second particle.

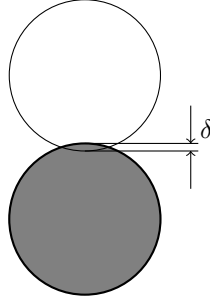


Figure 4.2: A particle, under the effect of gravity, resting upon a second particle of infinite density, unaffected by gravity. Overlap, δ , is labelled and exaggerated for clarity.

For the particle to be at rest the particle's gravity must be equal to the normal force from the DEM.

$$F_g = F_n \quad (4.1)$$

$$mg = k_e \delta - \eta u \quad (4.2)$$

To get an equation for overlap, δ , the equation is simplified and rearranged. The speed, u , is 0 at rest.

$$\delta = \frac{mg}{k_e} \quad (4.3)$$

Taking particle diameter to be 0.1 m and particle density to be 2000 kgm^{-3} , the particle mass is 1.047 kg. Gravity is taken to be 9.81 ms^{-2} and model spring stiffness is 10^5 .

$$\delta = \frac{1.047 * 9.81}{10^5} = 1.027 * 10^{-4} \quad (4.4)$$

Running a simulation with these parameters also yields an overlap of $1.027 * 10^{-4}$. Comparing simulation results with high precision overlap prediction shows that the simulation result is within $2.1 * 10^{-11}\%$ of the predicted value.

Timestep stability

This result can also be compared for varying timesteps to assess the stability of the implementation. Figure 4.3 shows very stable results up until a timestep of approximately 0.00097.

4.5.2 Timestep Stability

Collisions go boom if timestep is too high etc. The stiffer the collisions the smaller the timestep must be to maintain physical results. This can be spotted and logged if E_k is higher after collision than before.

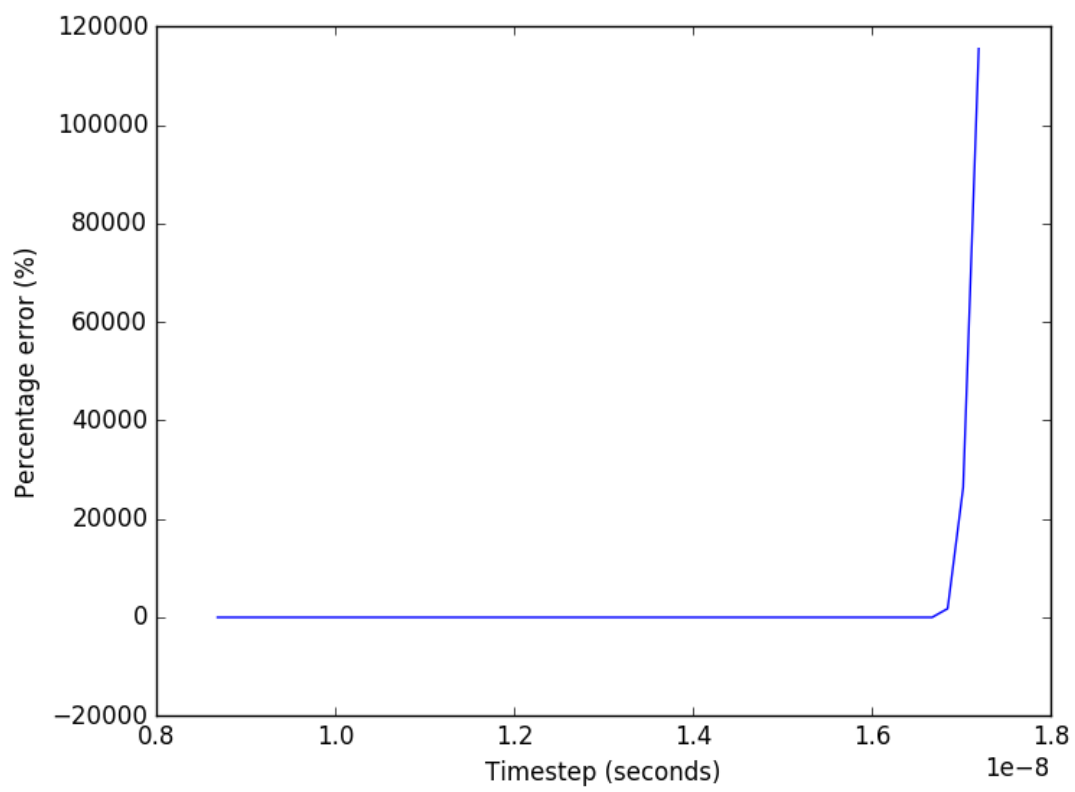


Figure 4.3: A graph of overlap percentage error against time step.

Chapter 5

OpenCL and Graphics Processing Units

Chapter 6

OpenCL Implementation

6.1 Data Structures

As the C language does not support objects, the objects used in the Python implementation must be simplified into data structures and separate functions. Most of the calculations that were done in the object functions will now be done within the kernels.

6.1.1 Particle

The particle structure contains very similar information as the Python implementation particle object. Notable differences are the lack of 'next_<property>' variables since these are just used within the iterate_particle kernel and do not need to be stored.

The structure is aligned to the nearest 128 bytes of memory to make accessing it easier. 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.

6.1.2 Collision

The collision structure contains two particle IDs and collision properties. This could be reduced to just particle IDs if collisions were taken as having the same properties throughout the simulation. Alternatively, if the collision changed based on particle properties, the collision resolution kernel could calculate the collision properties when they are needed.

6.1.3 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 the same definition of the structure as the host. 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. In addition padding has been added to the host code since the alignment specifier is apparently insufficient to ensure correct alignment. The alignment attribute specifier is also not the same between different compilers so “if defined” statements are implemented for both Visual Studio and GCC compilers. The code for these definitions is shown in figure 6.1.

Host:

```
typedef struct {
    cl_float3 pos;
    cl_float3 vel;
    cl_float3 forces;
    cl_ulong particle_id;
    cl_float particle_diameter;
    cl_float density;
    cl_float fluid_viscosity;
    cl_char padding[56];
    // Structure memory alignment for Visual Studio and GCC compilers.
#ifdef defined(_MSC_VER)
} __declspec(align(128)) particle;
#elif defined(__GNUC) || defined(__GNUG__)
} __attribute__((aligned (128))) particle;
#endif
```

Device:

```
typedef struct {
    float3 pos;
    float3 vel;
    float3 forces;
    ulong particle_id;
    float particle_diameter;
    float density;
    float fluid_viscosity;
} __attribute__((aligned (128))) particle;
```

Figure 6.1: Host and device definitions of the particle structure.

6.2 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.

6.2.1 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 single 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. When creating collisions this assumption will be checked and a warning issued. This arrangement of arrays is shown in figure 6.2.

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.

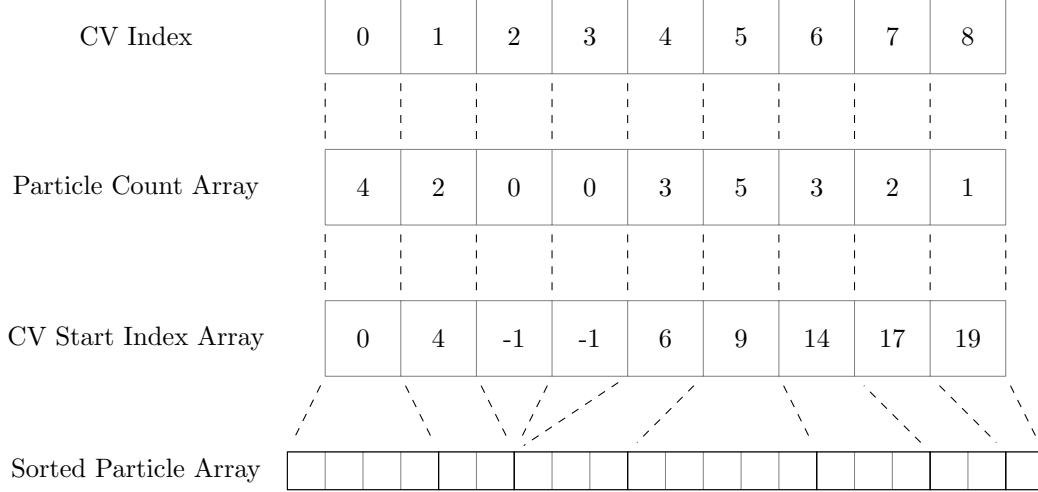


Figure 6.2: Diagram showing the structure and relationship between arrays representing Control Volumes.

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. This [could be]/[is] benchmarked in a later part of this project.

6.2.2 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 presented in The code is shown in figure 6.3. 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.

6.2.3 Particle Iteration

Particle iteration is performed almost identically to the Python implementation. The implicit integration scheme, as outlined in section 3.2, is used. The main difference, as discussed in section 6.2.2, 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.

```

_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 6.3: Kernel code for a solution to atomic addition with floats.

6.3 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.

6.3.1 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 paramter, '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 can be executed at runtime from a simulation to ensure that all tests are passed before starting a simulation run.

6.4 Verification

6.4.1 Normal Force

6.4.2 Friction Force

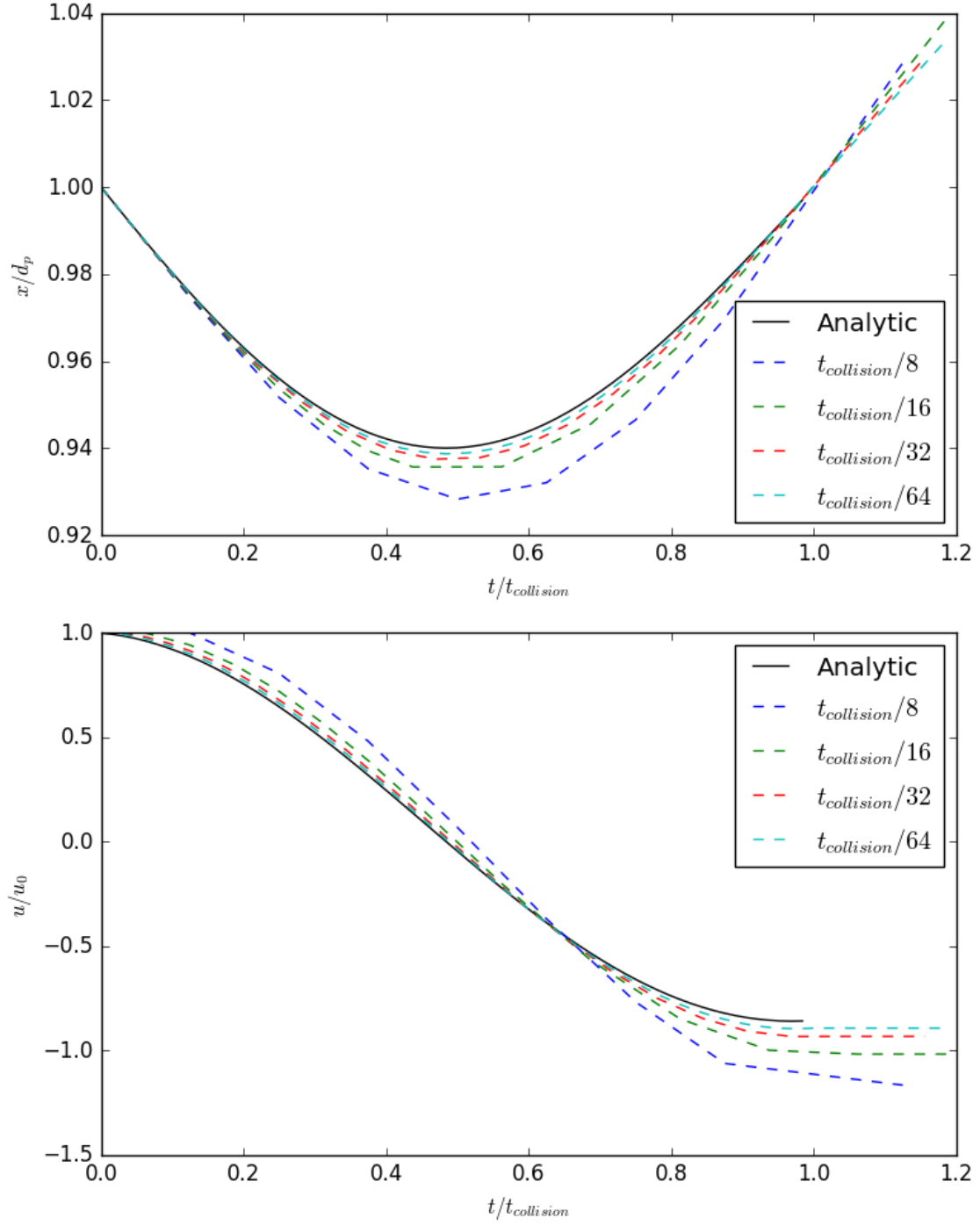


Figure 6.4: Normalized position and speed against time during a normal collision.

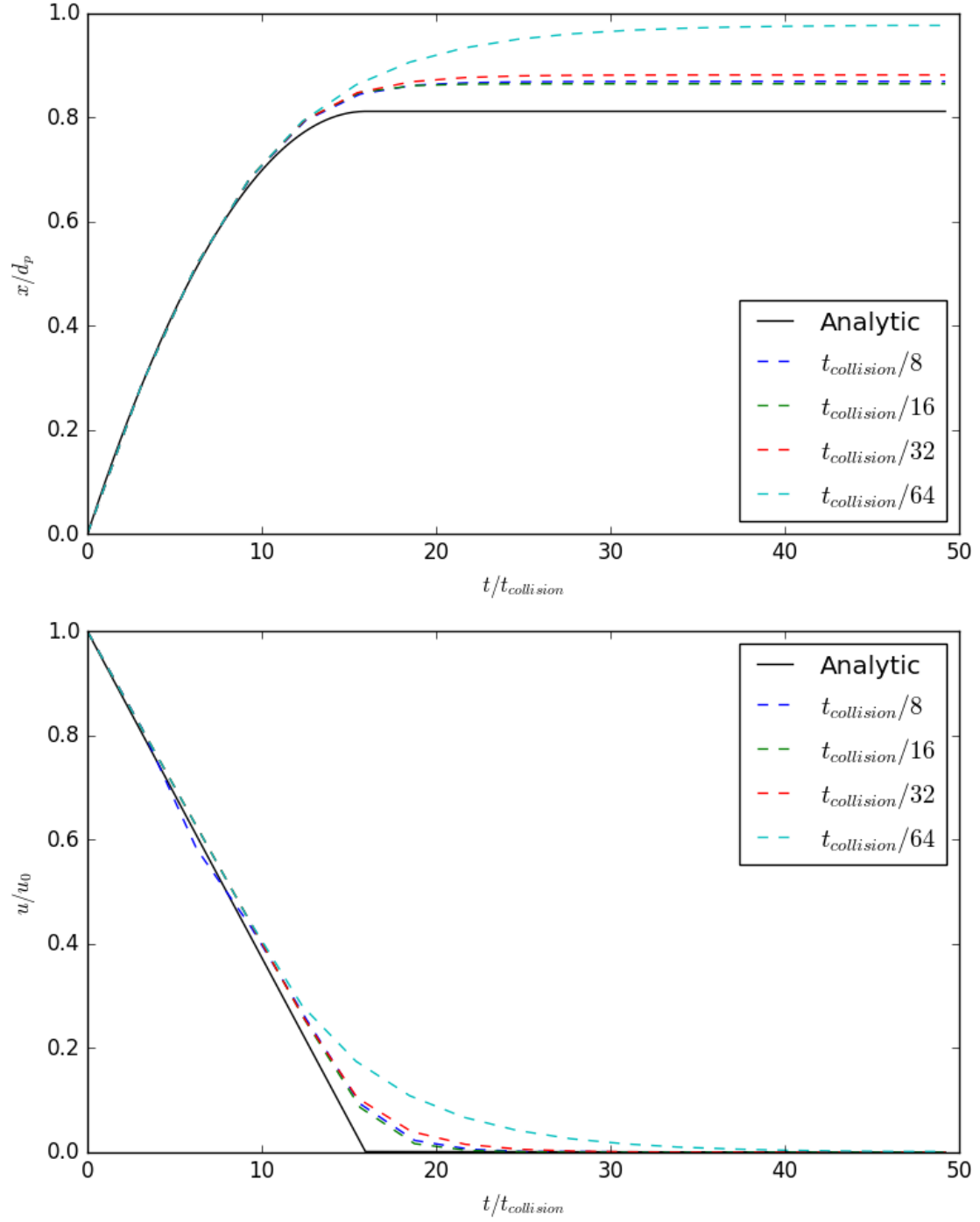


Figure 6.5: Normalized position and speed against time during a friction event.

Chapter 7

Results

7.1 Comparison between CPU and GPU

Chapter 8

Application

Modelling agglomerates...

8.1 Modified Particle Structure

Including effect diameter...

8.2 Fluid Velocity Field

get_vel_fluid... Taylor-Green Vortex...

8.3 Results

Pretty pictures etc...

Chapter 9

Conclusion

9.1 Further Work

Appendix A

Derivations

A.1 Equation 3.9

$$\dot{u} = \frac{v - u}{\tau} + \dot{u}_e \quad (\text{A.1})$$

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} + \dot{u}_e \quad (\text{A.2})$$

$$u_{n+1} = \frac{\frac{v}{\tau} + \frac{u_n}{\Delta t} + \dot{u}_e}{\frac{1}{\Delta t} + \frac{1}{\tau}} \quad (\text{A.3})$$

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{\left(\frac{\frac{v}{\tau} + \frac{u_n}{\Delta t} + \dot{u}_e}{\frac{1}{\Delta t} + \frac{1}{\tau}} \right) - u_n}{\Delta t} \quad (\text{A.4})$$

$$\dot{u} = \frac{v - u_n + \tau \dot{u}_e}{\tau + \Delta t} \quad (\text{A.5})$$

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.