

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Previous Work . . . . .	2
1.2	Project Goals . . . . .	2
<b>2</b>	<b>The Discrete Element Method</b>	<b>3</b>
2.1	Forces . . . . .	3
2.2	Collision Detection . . . . .	3
2.3	Implicit/Explicit . . . . .	3
2.4	Rotation/Quaternions . . . . .	3
<b>3</b>	<b>OpenCL and Graphics Processing Units</b>	<b>4</b>
<b>4</b>	<b>Python Implementation</b>	<b>5</b>
4.1	Overview . . . . .	5
4.2	Element Types . . . . .	5
4.2.1	Particle . . . . .	5
4.2.2	Axis-Aligned Simple Wall . . . . .	5
4.3	Collisions . . . . .	5
4.3.1	Collision Detection . . . . .	5
4.3.2	Collision Calculation . . . . .	6
4.4	Calculating Forces . . . . .	6
4.4.1	Drag . . . . .	6
4.4.2	Gravity . . . . .	6
4.4.3	DEM Forces . . . . .	6
4.5	Numerical Integration . . . . .	6
4.5.1	Velocity . . . . .	6
4.5.2	Position . . . . .	6
4.5.3	Method of Integrating Drag . . . . .	7
4.6	Verification . . . . .	9
4.6.1	Settling Overlap . . . . .	9
4.6.2	Terminal Velocity . . . . .	10
4.6.3	Timestep Stability . . . . .	10
<b>5</b>	<b>OpenCL Implementation</b>	<b>11</b>
<b>6</b>	<b>Results</b>	<b>12</b>
6.1	Comparison between CPU and GPU . . . . .	12
<b>7</b>	<b>Conclusion</b>	<b>13</b>
7.1	Further Work . . . . .	13
<b>A</b>	<b>Derivations</b>	<b>14</b>
A.1	Equation 4.9 . . . . .	14

# Chapter 1

## Introduction

1.1 Previous Work

1.2 Project Goals

## Chapter 2

# The Discrete Element Method

### 2.1 Forces

### 2.2 Collision Detection

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 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 this can decrease efficiency for statistical distributions of particle sizes.

### 2.3 Implicit/Explicit

### 2.4 Rotation/Quaternions

## Chapter 3

# OpenCL and Graphics Processing Units

# 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  $\Delta t$  to the particle object and iterates the velocity and position. The methods used for integrating these properties are discussed in section 4.5.

#### 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.

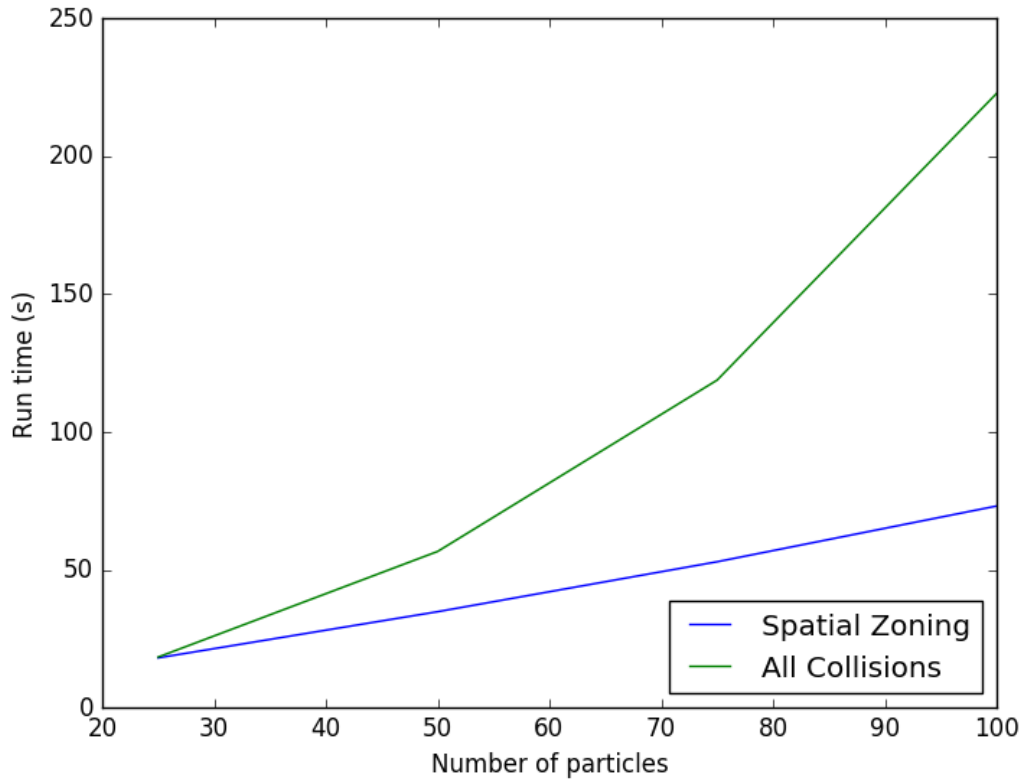
### 4.3 Collisions

#### 4.3.1 Collision Detection

Broad phase collision detection uses the simple spatial zoning technique described in Tuley[1]. 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 Calculation

Collisions are then resolved...

## 4.4 Calculating Forces

### 4.4.1 Drag

### 4.4.2 Gravity

### 4.4.3 DEM Forces

## 4.5 Numerical Integration

### 4.5.1 Velocity

Velocity is iterated with equation 4.1 where  $\dot{u}$  is the acceleration obtained from the function call 'Particle.get\_accel()'.  

$$u_{n+1} = u_n + \dot{u}\Delta t \quad (4.1)$$

### 4.5.2 Position

Position is iterated with equation 4.2.

$$x_{n+1} = x_n + \frac{u_{n+1} + u_n}{2} \Delta t \quad (4.2)$$

### 4.5.3 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 4.3.

$$\dot{u} = \frac{v - u}{\tau} \quad (4.3)$$

Equation 4.3 can be solved to show that the analytical solution for the particle speed,  $u$ , is that in equation 4.4.

$$u(t) = v(1 - e^{-\frac{t}{\tau}}) \quad (4.4)$$

The explicit numerical integration method for equation 4.3 is that in equation 4.5. Where  $u_n$  is the current speed and  $u_{n+1}$  is the speed after timestep  $\Delta t$ .

$$\frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_n}{\tau} \quad (4.5)$$

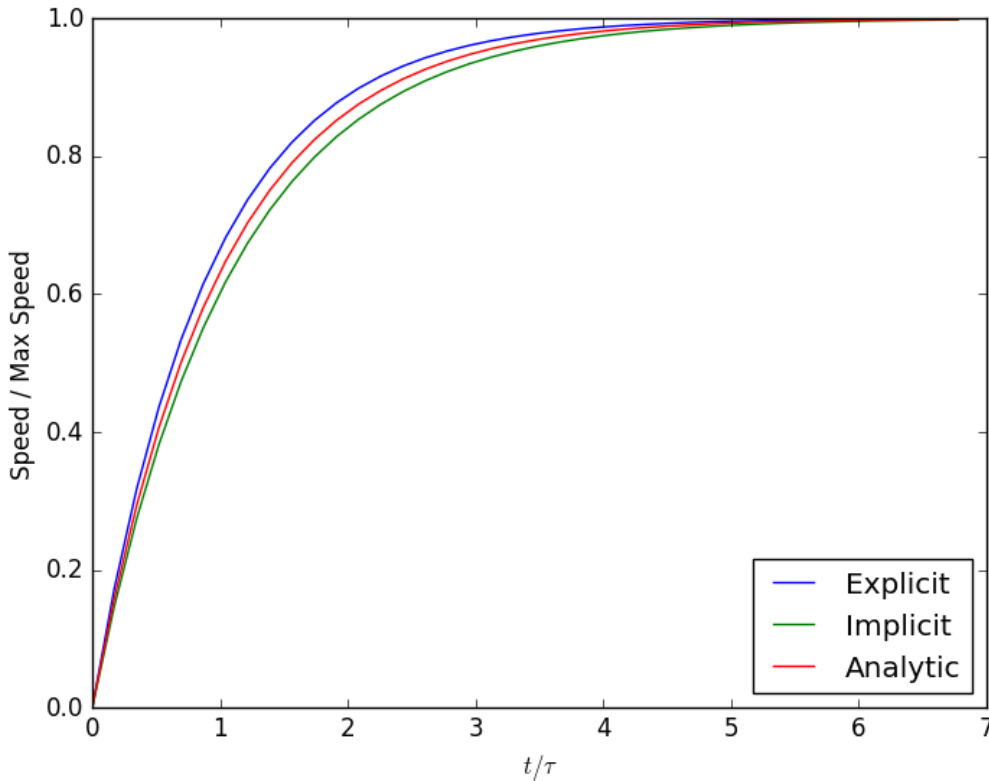
The implicit numerical integration method for equation 4.3 is that in equation 4.6. Where  $u_n$  is the current speed and  $u_{n+1}$  is the speed after timestep  $\Delta t$ .

$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} \quad (4.6)$$

Equation 4.6 can be rearranged to get an equation of the form  $\dot{u} = f(u_n)$  as shown in equation 4.7.

$$\dot{u} = \frac{v - u_n}{\tau + \Delta t} \quad (4.7)$$

When these three methods are applied to the system they produce the results in figure 4.2.

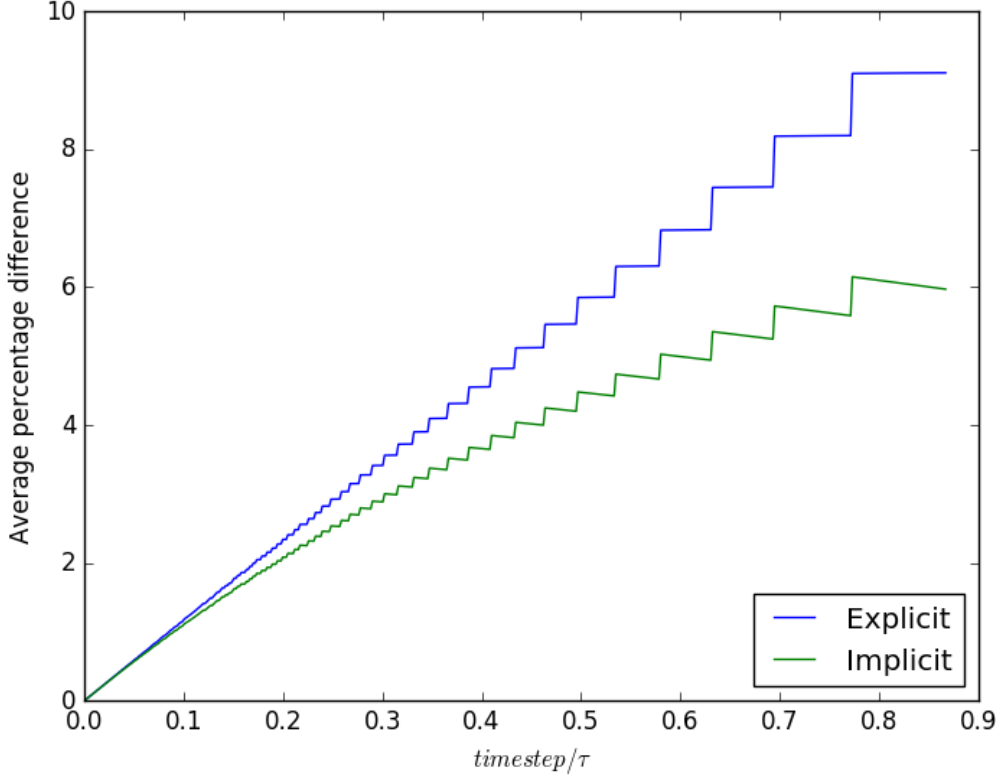


**Figure 4.2:** A graph of particle speed against time for the three methods of integration.

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 4.3. 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.



**Figure 4.3:** A graph of average percentage difference between the numerical method and analytical solution against varying timestep.

Unlike the explicit method, the implicit method depends on the other accelerations in the system. Equation 4.7 can be redefined to include these other accelerations as an extra  $\dot{u}_e$  term as shown in equation 4.8.

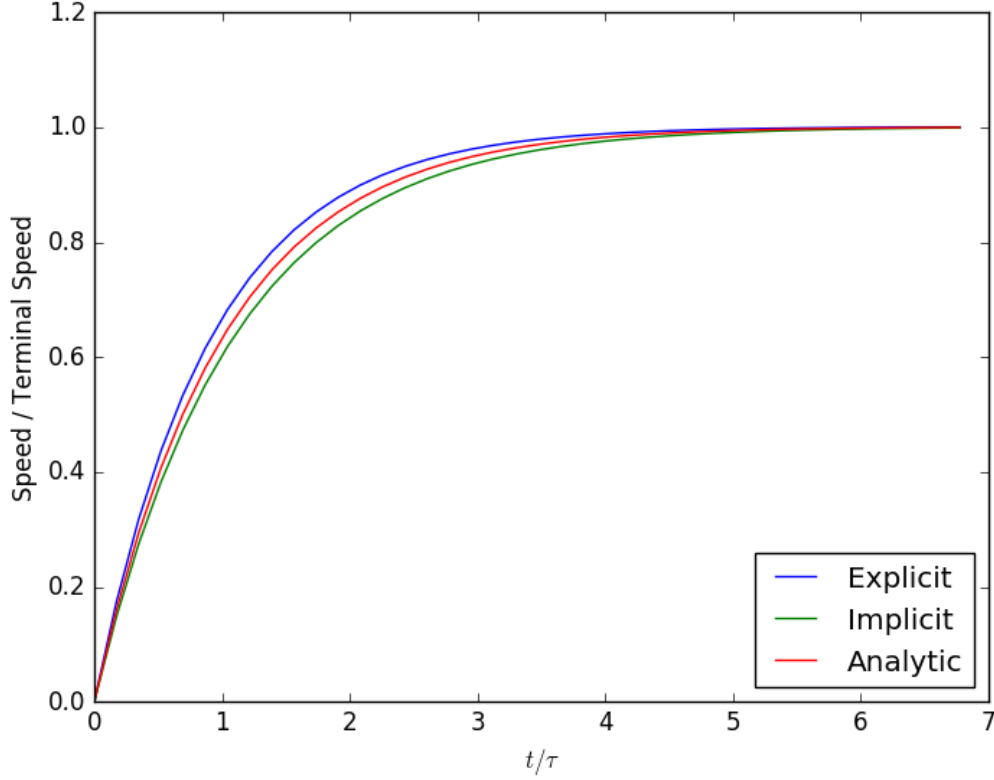
$$\dot{u} = \frac{u_{n+1} - u_n}{\Delta t} = \frac{v - u_{n+1}}{\tau} + \dot{u}_e \quad (4.8)$$

As before, equation 4.8 can be rearranged to be in the form  $\dot{u} = f(u_n)$  as shown in equation 4.9. The derivation of this can be found in appendix A.1.

$$\dot{u} = \frac{v - u_n + \tau \dot{u}_e}{\tau + \Delta t} \quad (4.9)$$

Equation 4.9 can be applied to a particle falling under the effect of gravity through a stationary fluid. The results are shown in figure 4.4. 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 4.7.





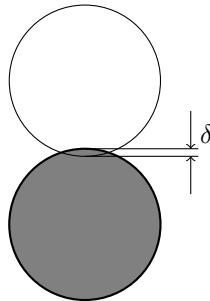
**Figure 4.4:** A graph of particle speed against time for each method of integration using equation 4.9.

## 4.6 Verification

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

### 4.6.1 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 4.5:** A particle, under the effect of gravity, resting upon a second particle of infinite density, unaffected by gravity. Overlap,  $\delta$ , 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.

$$F_g = F_n \quad (4.10)$$

$$mg = k_e \delta - \eta u \quad (4.11)$$

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

$$\delta = \frac{mg}{k_e} \quad (4.12)$$

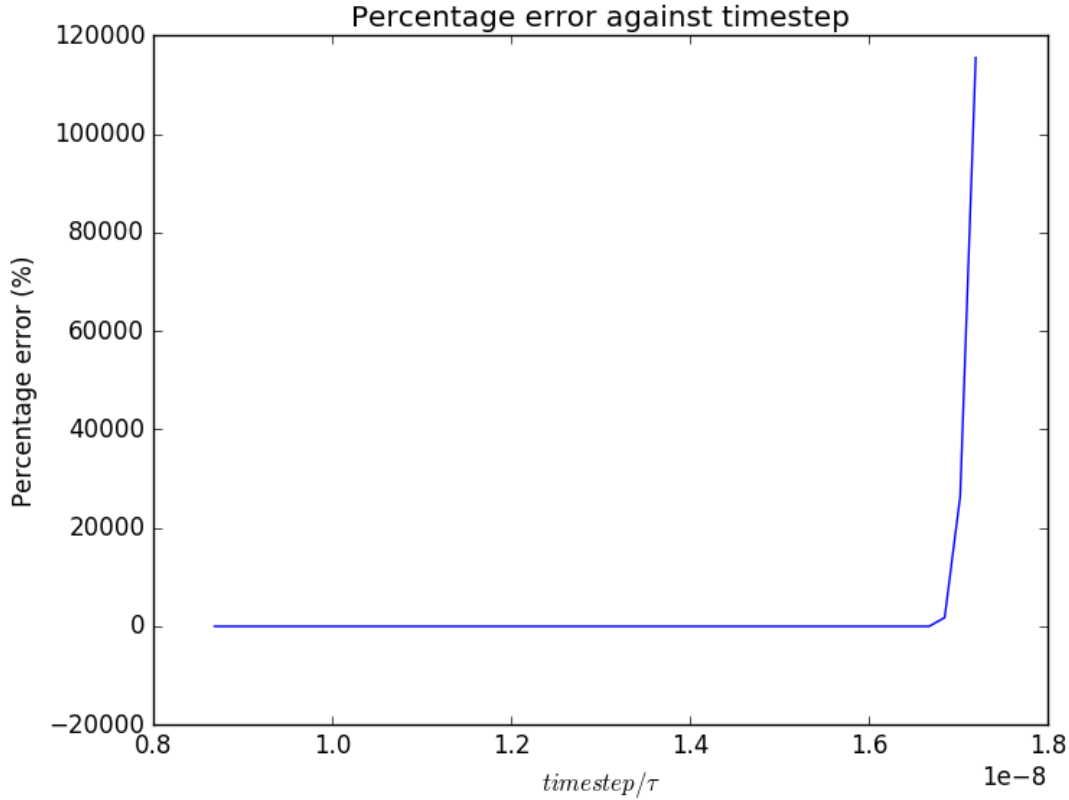
Taking particle diameter to be  $0.1 \text{ m}$  and particle density to be  $2000 \text{ kgm}^{-3}$ , the particle mass is  $1.047 \text{ kg}$ . Gravity is taken to be  $9.81 \text{ ms}^{-2}$  and model spring stiffness is  $10^5$ .

$$\delta = \frac{1.047 * 9.81}{10^5} = 1.027 * 10^{-4} \quad (4.13)$$

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 asses the stability of the implementation. Figure 4.6 shows very stable results up until a timestep of approximately 0.00097.



**Figure 4.6:** A graph of overlap percentage error against time step.

### 4.6.2 Terminal Velocity

As mentioned in [ref], integrate and compare with different timesteps.

### 4.6.3 Timestep Stability

Collisions go boom if timestep is too high etc. [Test against bouncing or something]. This can be spotted and logged if  $E_k$  is higher after collision than before.

## Chapter 5

# OpenCL Implementation

## Chapter 6

# Results

### 6.1 Comparison between CPU and GPU

## Chapter 7

# Conclusion

### 7.1 Further Work

# Appendix A

## Derivations

### A.1 Equation 4.9

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

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

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

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

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

# Bibliography

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