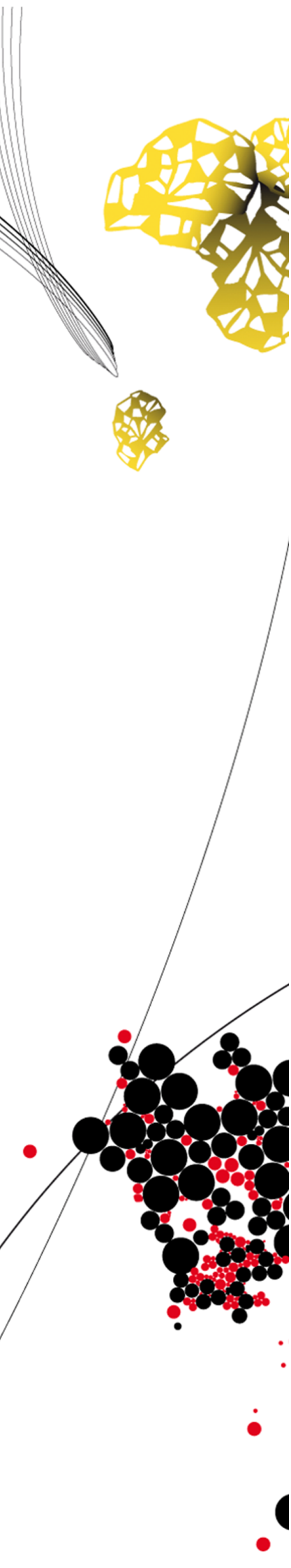


UNIVERSITY OF TWENTE.

Faculty of Thermal and Fluid Engineering

Applying discrete particle simulations to industrial-sized problems using MercuryDPM

J.W. Bisschop — s1597116

Internship

April-June 2019

Supervisors:
dr. D. Pinson (BSL)
prof. dr. A. R. Thornton (UT)
dr. T. Weinhart (UT)

Coke and Ironmaking Technology
BlueScope Steel Limited
Port Kembla, Australia

Multi-Scale Mechanics Group
Faculty of Thermal and Fluid Engineering
University of Twente
Enschede, The Netherlands

Preface

During my study Mechanical Engineering at the University of Twente in Enschede, the Netherlands I got more and more interested in the research done by the Multi-Scale Mechanics group. It all started with my bachelor thesis on the failure line in rotating drums with granular materials and Deepak Tunuguntla and Anthony Thornton were the ones who actively helped and motivated me throughout, which led me to choose a master study in this field. Once chatting with Anthony he mentioned David Pinson, a guy in Australia who is always interested in having some master students around. This once in a lifetime opportunity was handed to me just like that and so I ended up at BlueScope Steel in Port Kembla, Australia. Here David was a great supporter and motivator for the work I carried out and in the mean time taught me some life lessons as well. He and his colleagues showed me around on site, which was an impressive and fun experience. A short description on BlueScope Steel and a reflection on my position can be found in Appendix A. I hereby want to thank David very much for this opportunity and the help he gave me, as well as Anthony and Thomas Weinhart who were my contacts from back home.

Summary

Steelmaking, as well as many other industrial and natural processes, involves granular materials and one way to study their behaviour is by making discrete particle simulations, for which in this work MercuryDPM, a code developed at the University of Twente, is used. The main project involves a waste gas cleaning plant, in which activated carbon particles slowly descend, while waste gas from the sintering process is blown through them and the SO_x molecules are absorbed. A few other projects are looked at as well, the first being about the difference between using solid walls versus periodic boundaries in a 2D model of a blast furnace. Another project uses a profile scanner and belt weigher to try and calculate the bulk density of a material on a conveyor belt and the last project repeats the simulation of a simple experiment done before to find the influence of the particle-particle sliding and rolling friction coefficient on the angle of repose.

To properly resemble reality certain simulations parameters have to be decided on, which is done by comparing simulations with experiments. These experiments are a box test, a cubic box filled with particles and after removing one side the angle of repose of the remainder is measured; a tube lift, where a tube filled with particles is slowly lifted from a flat surface and the angle of repose of the pile that forms is measured; and a bridging test, a hopper filled with particles with different sized orifices to see whether or not bridging (particles blocking the outflow) is happening. Simulations of the box test and tube lift are varied with different sliding and rolling friction coefficients and after comparing the angle of repose with the experiments their values are chosen to be respectively 0.25 and 0.4. Simulations of the bridging test are mainly used for a quick comparison and are in agreement with the experiment. Experience has taught that a particle stiffness of $40000N/m^2$ and a restitution coefficient of 0.25 are proper values and furthermore a timestep of $1e-5s$ is used.

Particles are inserted by a rotary valve at the top of the waste gas cleaning plant and land on centering rings, which center the stream of particles, after which they land on the distributor and split into four directions, namely two long and two short arms in respectively the positive and negative x- and z-direction. The rotary valve is imitated by considering four cases, namely particles dropped at the outer most positive x-, xz- and z- direction as well as the exact center. Areas in the system

containing stationary particles are filled as much as possible beforehand to reduce computation time. The simulations show the particles are not equally distributed, mainly because the centering rings bounce the particles to the opposite side of which they are dropped instead of centering them. Extending the cylindrical part of the centering rings might help prevent this. Furthermore, the long arms barely have any particles rolling down on them.

A 2D slot model and 3D model with periodic boundaries of a blast furnace are simulated and are kept completely filled at all times, while being discharged with a certain mass flow rate at the bottom sides. Once the system is in steady state the data is coarse grained in order to get continuum fields from the discrete particle data. By plotting the velocity magnitude it is easy to see if a deadman will form and, in agreement with previous studies, this does happen for the 2D slot model, but not for the 3D model with periodic boundaries. In addition a higher discharge mass flow rate decreases the deadman size.

To calculate the bulk density of the material on a conveyor belt the mass flow rate and volume flow rate are used, of which the former is given by a belt weigher and the latter is calculated using a profile scanner. The profile scanner gives the outline of the material, which can be thought of being the cross-section of the material and the area can be approximated by rectangles by assuming the belt is fully pressed on the conveyor belt rollers, which themselves are assumed to have a fixed known position. After multiplying the area with the known belt velocity the volume flow rate is obtained and the bulk density can be calculated. Plotting this shows clusters of points with a lot of variation within clusters as well as variations between clusters. The former is expected to be because of uncertainties in measurements and small calculation errors, while the latter is expected to be from a change in belt velocity or possibly an actual change in bulk density. Integrating the actual belt velocity in the volume flow rate calculation will likely get rid of the big variations.

The influence of the particle-particle sliding and rolling friction coefficient on the angle of repose of the box test is studied in more detail by setting their range to respectively $0.05-0.5$ and $0.01-0.28$, both divided in 10 equal increments. Simulations are run 10 times with their random seed changed, where in the end the average is taken, as well as for 3 different materials sinter, pellet and coke. A few scripts are written to easily run the 3000 simulations, get a screenshot of their last frame and use that to get the angle of repose. A 3D plot of the sliding and rolling friction coefficient and the angle of repose shows little difference between the different materials used. A higher sliding or rolling friction coefficient results in a higher angle of repose, however at some point the curve flattens, and really low values of one could cancel out the influence of the other. The influence of other factors, e.g. the coefficient of restitution, remain a subject for future research.

Contents

Appendices

List of acronyms

AoR	angle of repose
BF	blast furnace
SFC	particle-particle sliding friction coefficient
RFC	particle-particle rolling friction coefficient

Chapter 1

Introduction

1.1 Motivation

Many industrial and natural processes involve granular materials and there is still much to learn about their behaviour. Making simulations using the discrete element method can be very time consuming and in need of lots of computation power, however it is often the only way to get insight in the processes studied. At the University of Twente a code for discrete particle simulations has been developed called MercuryDPM [1], which is the main tool used for the projects described in this report. All simulations in this report use the LinearViscoelasticFrictionSpecies, which consists of the LinearViscoelasticNormalSpecies and the FrictionSpecies. The former uses a linear spring dashpot model and the latter considers the sliding friction, rolling resistance and torque resistance to describe the linear tangential contact forces.

1.2 Framework

In the steelmaking process at BlueScope Steel in Port Kembla, Australia the waste gas from the iron ore sinter plant is cleaned in a waste gas cleaning plant, in which the waste gas is blown through a bed of activated carbon particles absorbing the SO_x molecules from the gas stream. These particles are initially cylindrical, but end up smaller and more spherical after multiple uses. The particles are added at the top of the system and slowly descend, while being removed at the bottom. In the past a few temperature excursions have occurred, which heavily implicate the performance of the feed system and it is suspected that this is a free surface and segregation problem.

The plant itself is continuously running and cannot be put on halt without affecting the production process. Even if it was possible it would still be a challenge to find a way to have a proper look inside. Making simulations of the plant is a good way to get

around this problem and will give more insight of what might be happening. Since the particle-equipment ratio is very large it is not feasible to make simulations of the plant as a whole, therefore the first step in understanding the particle behaviour inside the plant is by looking at the insertion of particles at the top of the system and studying how the distribution of particles is happening.

Although the waste gas cleaning plant is the main project in this report, once the simulations are running other projects are worked on as well. These projects include simulations of a blast furnace, data analysis of a conveyor belt used in coke making and extending simulations of a basic experiment to find the influence of the rolling and sliding friction on the angle of repose.

In a blast furnace solid particles are continuously added at the top and slowly descend, while being discharged at the bottom sides. At the bottom center a deadman will form, which consists of stationary particles. Our model is a small cross section of a blast furnace and we are interested in the difference in the formation of the deadman between using solid walls versus periodic boundaries for the front and back wall, i.e. the difference between a 2D slot model and a 3D model with periodic boundaries.

The fuel used in a blast furnace is called coke and it is produced by baking coal in an coke furnace in the absence of air. The main property of interest is the bulk density of the coal before entering the furnace. A section of the conveyor belt transporting the coal contains a belt weigher, which returns the mass flow rate, and at the same section a profile scanner is placed, which measures the outline of coal on the belt. This, combined with a known belt velocity, is used to calculate the volume flow rate and together with the mass flow rate the bulk density is easily calculated.

The box test as described in Section 2.1 is simulated again, but more extensively in order to find a more detailed answer as to what the influence of the particle-particle sliding and rolling friction coefficient is on the angle of repose.

1.3 Research questions

A research question is formulated for each of the projects addressed in this report, respectively the waste gas cleaning plant, blast furnace, conveyor belt and box test:

- Is the distribution of particles at the top of the waste gas cleaning plant done in a equal manner?
- What is the difference between using solid walls versus periodic boundaries for the formation of a deadman in a cross section of a blast furnace and how does this compare to previous studies?

- Can a profile scanner and belt weigher be used to accurately describe the bulk density through time?
- How do the particle-particle sliding and rolling friction coefficient influence the angle of repose of the box test?

1.4 Report organization

The main project of this report is described in two chapters, namely Chapter 2, which describes the experiments done to find certain simulation parameters, and Chapter 3, in which the actual simulation of the top part of the waste gas cleaning plant is elaborated. Following up are Chapter 4, which describes the simulation of the blast furnace, Chapter 5, which is about calculating the bulk density from a belt weigher and profile scanner, and Chapter 6, on the influence of the particle-particle sliding and rolling friction coefficient on the angle of repose in a box test. Finally, Chapter 7 gives conclusions drawn from this report and recommendations for future research. All relevant codes made during the research can be found in Appendices B to F.

Chapter 2

Experiments

Before starting any simulations it is important to find the correct parameters to mimic the behaviour of the actual particles. This is done by comparing a few simple experiments with simulations of these experiments. The activated carbon particles are initially cylindrical, but through reuse become more spherical once they start breaking down and thus the general mix consists of cylindrical and roughly spherical particles, with sizes ranges from $11 - 1.2mm$. For simplicity the simulations use perfectly spherical particles as well as their average diameter of $5 - 7mm$. Experience has taught that a particle stiffness of $40000N/m$ and a restitution coefficient of 0.25 for both particle-particle and particle-wall are good values, as well as a time step of $1e - 5s$. The (sliding and rolling) friction coefficients for particle-particle and particle-wall are taken to be the same and, by comparing simulations with the experiments described below, their values are found to be 0.25 and 0.4 for respectively the sliding and rolling friction. In Appendices B.1, B.2 and B.3 the drivers code for respectively the box test, tube lift and bridging test can be found.

2.1 Box Test

The first experiment involves a simple cubic box of size $150mm$ filled with activated carbon particles. The box is placed at the edge of a flat surface, after which the side facing the edge is removed and the particles spill out over the edge. The slope formed by the particles left in the box, called the angle of repose (AoR), is measured for multiple experiments and their average is taken. Because of friction contact with the front and back wall the AoR is a little bit bigger at the sides compared to the middle of the box. The latter is used for simulations the compare with, since in the simulations periodic boundaries are used for these walls, which, with a distance of 5 times the biggest particle diameter between them, decreases the computation time by about half. Figure 2.1 shows the result of a simulation for a sliding and rolling friction coefficient of respectively 0.25 and 0.4.

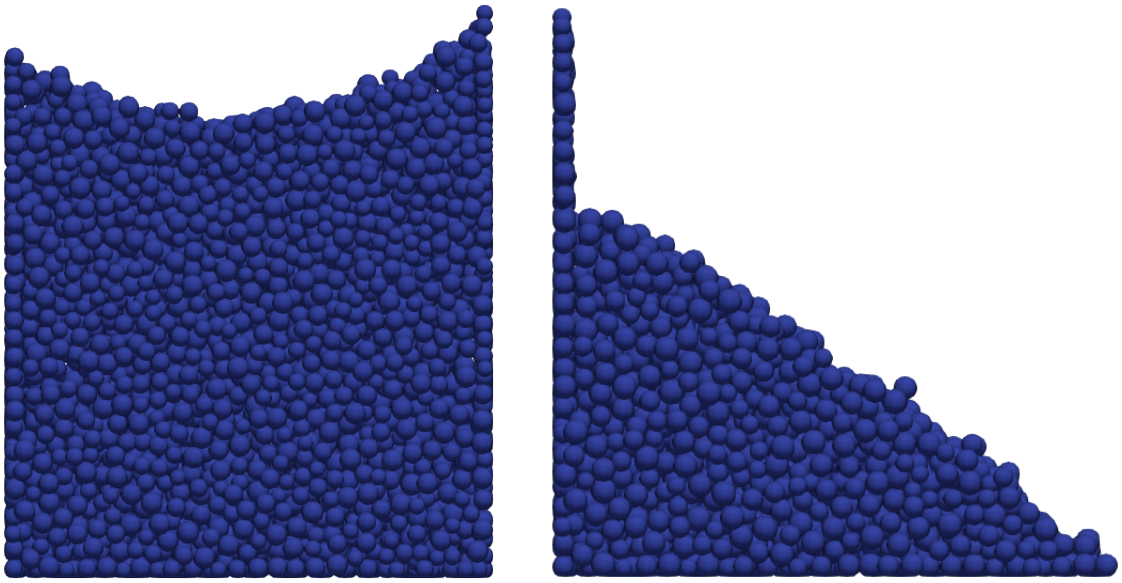


Figure 2.1: Box test simulation result for a sliding and rolling friction coefficient of respectively 0.25 and 0.4

2.2 Tube Lift

For the second experiment a tube standing on a flat surface is filled with activated carbon particles and then slowly lifted. The particles slowly spill from underneath the tube and, once the tube is completely lifted, a pile is formed for which the AoR is measured at three different places around the pile and after multiple repetitions of the experiment their average is taken. For the simulations the tube has an upward velocity of $1mm/s$ and a tube diameter and height of $100mm$. Figure 2.2 shows the result of a simulation for a sliding and rolling friction coefficient of respectively 0.25 and 0.4.

Table 2.1 shows the result for the box test and tube lift simulation. More simulations have been done, but they are not all shown here. From experiments the average value of the AoR were found to be 38.0 and 30.9 for respectively the box test and tube lift. For now it is not important to be really exact and therefore a sliding and rolling friction coefficient of respectively 0.25 and 0.4 are chosen to be used for all future simulations.

2.3 Bridging Test

A third experiment involves a hopper where at the bottom different sizes orifices can be attached, see Figure 2.3 for a schematic drawing. Once the hopper is filled with activated carbon particles the orifice is opened. For smaller sized orifices bridging

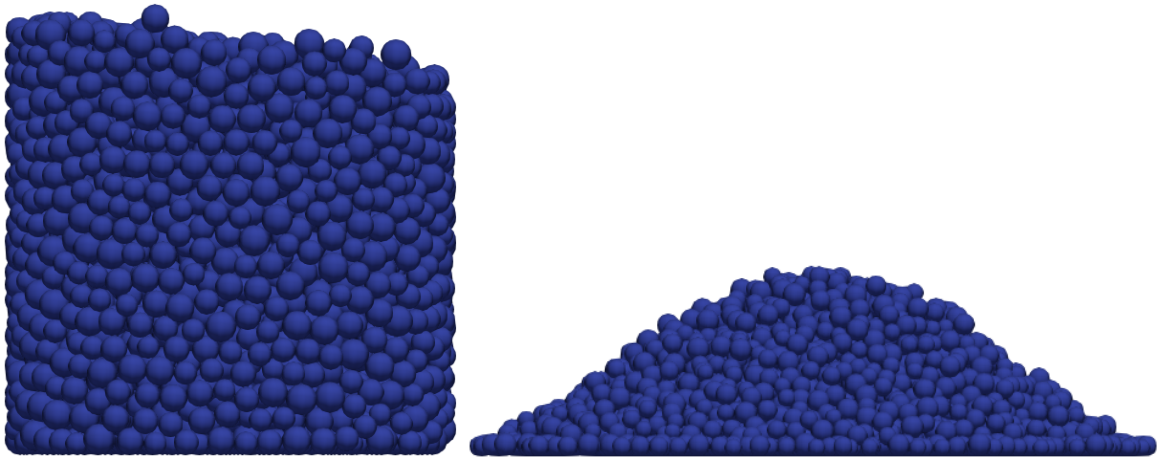


Figure 2.2: Tube lift simulation result for a sliding and rolling friction coefficient of respectively 0.25 and 0.4

sliding	rolling	AoR Box	AoR Tube
0.2	0.2	28	-
0.3	0.2	34	34
0.3	0.3	35	38
0.25	0.4	35	33
0.25	0.5	33	34

Table 2.1: Box test and tube lift simulation results

occurs, which means the particles block the orifice and the flow stops. Table 2.2 shows the results from experiments and simulations, where only the smallest three diameters are shown, since the bigger diameters did not show bridging and were therefore not the starting point of the simulations. Notice that for an orifice diameter of $45mm$ tapping was required during the experiment to maintain flow. Only the smallest diameter orifice showed bridging happening and the remaining simulations kept running for a long time, but after not showing any sign of bridging they were stopped. For these cases bridging was not expected to happen, since the particles were perfect spheres compared to the actual shape of the particles used in the experiments. Since the result of this experiment was mainly used for a quick comparison and because the focus of this paper is not on bridging it was decided to continue without looking into this any further.

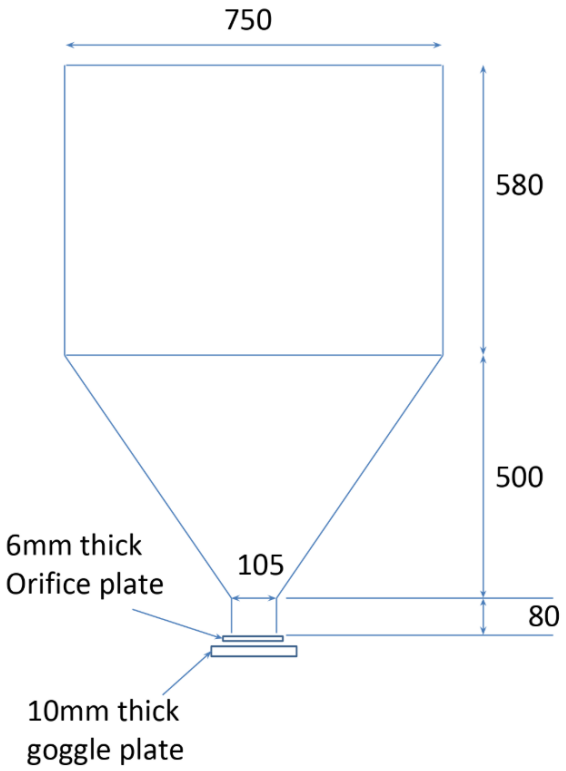


Figure 2.3: Schematic drawing of the hopper in mm

Diameter (mm)	Flow experiment	Flow simulation
55	Yes	Yes
45	Yes/No	Yes
30	No	No

Table 2.2: Bridging test experiment and simulation results

Chapter 3

Waste Gas Cleaning Plant

As mentioned in the introduction, our interest lies in the distribution of particles at the top of the waste gas cleaning plant, of which a 3D model is shown in Figure 3.1. Particles are inserted by a rotary valve and land on a distributor, which splits the stream of particles into four directions, namely negative and positive x- and z-direction as shown in Figure 3.2. Since the rotary valve does not drop the particles exactly in the middle, the distribution would not be done equally were it not for the centering rings located above the distributor and although it is easy to assume they help a great deal, it is not actually certain how well they work. At this stage the rotary valve itself is ignored and its effect is imitated by dropping the particles at different points above the centering rings. Four cases are considered: a drop at the exact center and drops at the outer most positive x-, xz- and z-position. The drivers code of the simulation can be found in Appendix C.1.

3.1 Particle insertion

Figure 3.3 shows the three different parts of the system. The centering rings and long arms have sections, which are naturally filled once the process is running and the velocity of other particles rolling on top of them is greatly reduced compared to rolling on a flat surface. To reduce computation time the sections are filled as much as possible beforehand. In general the number of particles needed to completely fill a certain section is approximated by dividing the volume of that section by the volume of an average sized particle and multiplying that by a fill fraction, since there is always a bit of space in between stacked particles. The fill fraction will always be between 0 and 1, is found by trial and error and is dependent on the particle diameter and shape of the section filled. Particles are given a random radius between their minimum and maximum value and are placed randomly within the boundaries of a section. It is preferred to have as little particle interaction as possible, because

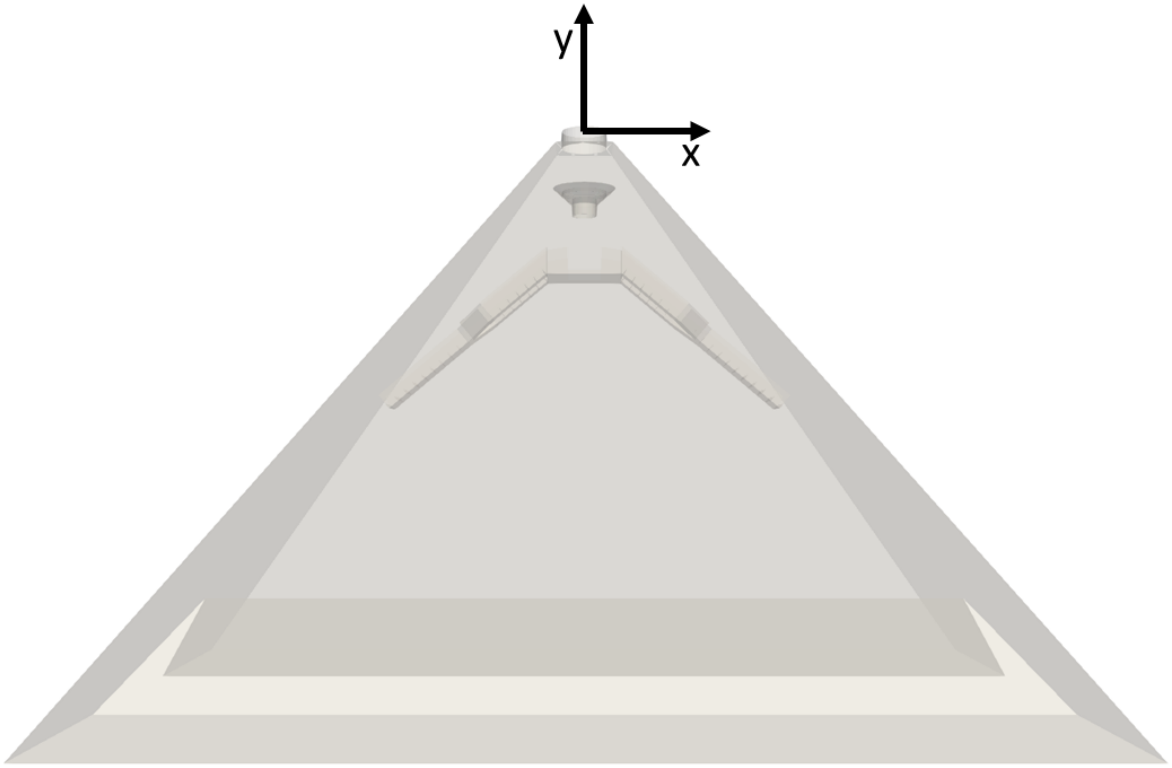


Figure 3.1: Top part of the waste gas cleaning plant

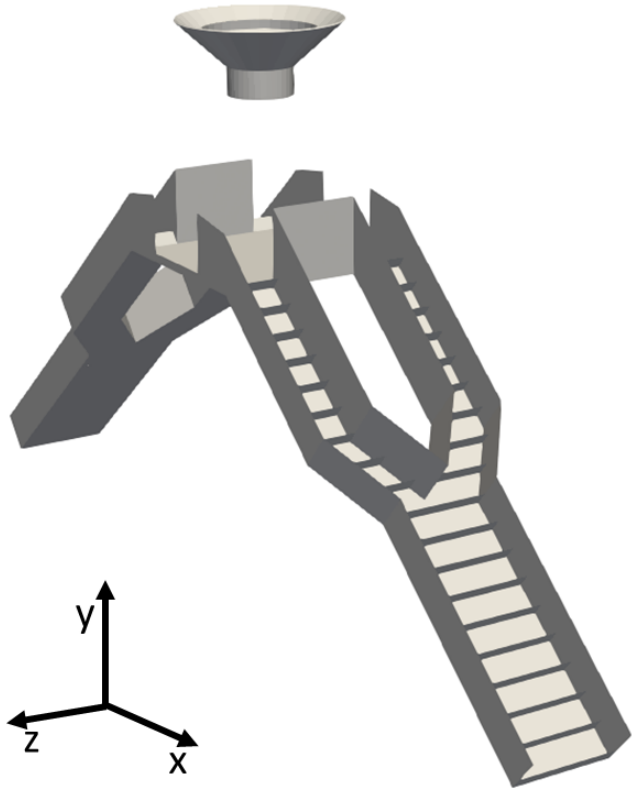


Figure 3.2: Distributor and centering rings

otherwise particles may shoot off, will not settle as well and in general make it less predictable what will happen. Every time a particle is placed it is checked if it has an interaction and if so another position is tried. This is done as many times as needed, up to a certain limit after which the particle is added anyway even though it does have an interaction. To help with placing the particle without interactions and to reduce computation time the upper boundary of a section is set a bit higher, but should not be unnecessarily high, because that will increase settling time. Usually multiplying the height of a section by a value of $1-2$ does the trick. Besides that, the boundaries between which the particles are placed are the actual boundaries plus or minus the particle radius.

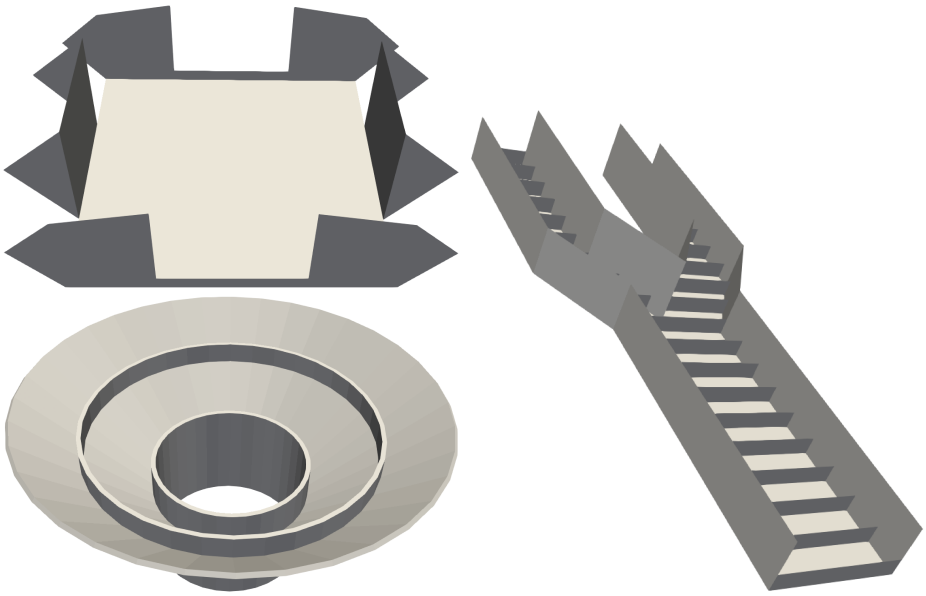


Figure 3.3: *Top left:* Box section. *Bottom left:* Centering rings. *Right:* long arm.

The two long arms are placed symmetrically in the yz-plane an are symmetrical themselves in the xy-plane. Besides that each section within them is the same, expect for the angled part, which in itself has three similar sections. To reduce computation time, for only one section the particle positions are set and these positions are copied to consecutive sections. This means that the particle positions for all sections are similar, however their radiuses are all set at random. The particle positions of the first section are set in the rectangular shape of the section at the origin first, after which they are rotated by $40°$, the angle at which the long arms are placed, and translated to the right position.

The centering rings are rotationally symmetric and have two sections, the inner-middle and the middle-outer ring section. For the particle positions a random position radius and rotation angle is chosen, while the y-position is dependent on the position radius, because the bottom boundary is angled. This angle should be known

from the 2D drawings and 3D model available, but the exact angle is not very clear. Instead the 3D model dimensions are used to calculate a ratio as is shown now. A schematic cross-section of part of the middle-outer ring section can be seen in Figure 3.4a. In this figure t stands for the top position in y-coordinates, h for the height, R for the radius and the subscript p for particle, t_p being the only unknown. The ratio of the big triangle is:

$$ratio = \frac{|t_{middle} - t_{outer}| + h_{middle}}{R_{outer} - R_{middle}} \tag{3.1}$$

Therefore the lowest possible position of the particle is:

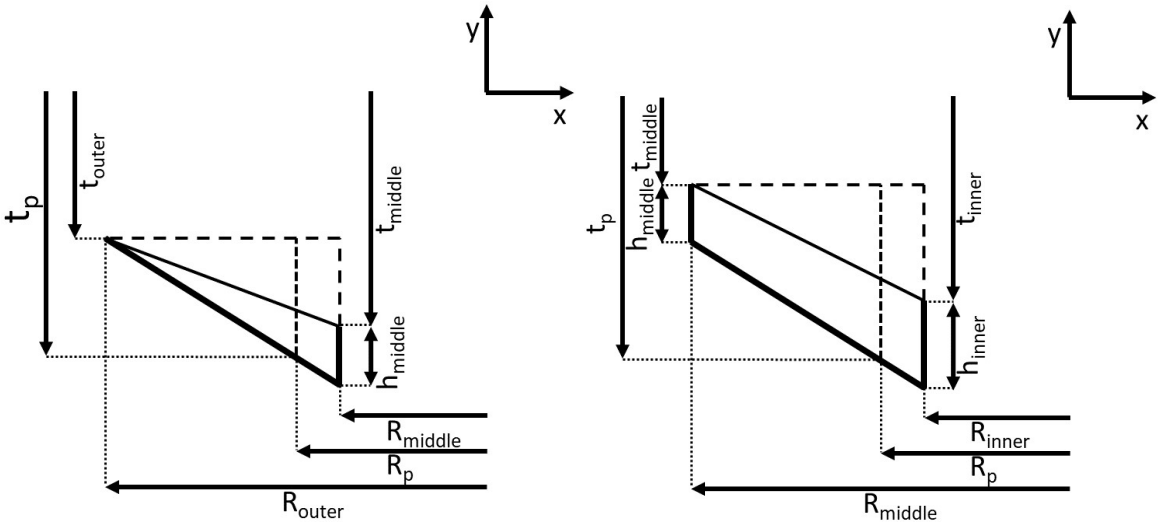
$$t_p = t_{outer} - ratio \cdot (R_{outer} - R_p) \tag{3.2}$$

The same can be done for the inner-middle ring section and although the ratio is (and should be) pretty much the same, it is shown here again for completeness. Figure 3.4b shows a schematic cross section and the ratio of the big triangle is:

$$ratio = \frac{|t_{inner} - t_{middle}| - h_{middle} + h_{inner}}{R_{middle} - R_{inner}} \tag{3.3}$$

And the lowest possible particle position:

$$t_p = t_{middle} - h_{middle} - ratio \cdot (R_{middle} - R_p) \tag{3.4}$$



(a) Middle-outer ring section (b) Inner-middle ring section

Figure 3.4: Schematic cross sections of (part of) the centering rings

The box section is filled by a bed of particles at the bottom, on top of that a cone of particles and at two sides in the positive and negative x-direction another layer of particles. The particles position in the cone have a similar dependency as the

centering rings, but here a random angle of $40°$ is chosen and the maximum particle position is easily calculated to be $h_{cone} = (R_{cone} - R_p) \cdot tan(40)$.

The rotary valve is imitated by inserting particles in sets of roughly $1kg$ (~ 800 particles) in intervals of $0.75s$ and in a cylindrical shape at the top of the system. In reality these intervals take longer, however to use the computation time more efficient the next set is inserted once the previous set reaches the box section and while this set is settling down or rolling away the next set can already start its descend. The particles have a density of $900kg/m^3$ and a diameter of $13 - 15mm$, which is larger than the actual/preferred size to decrease computation time. When all parts of the system are initially filled there are little over 40000 particles, shown in Figure 3.5. Since the simulations are very large they are run on a cluster with 36 processors. An existing MercuryDPM python script has been modified to easily visualize the cluster data in Paraview and can be found in Appendix C.2. Since running the four cases still is very slow, another simulation is started with the particle sizes doubled ($26 - 30mm$), which means the total number of particles is about eight times smaller and this, of course, reduces computation times drastically.

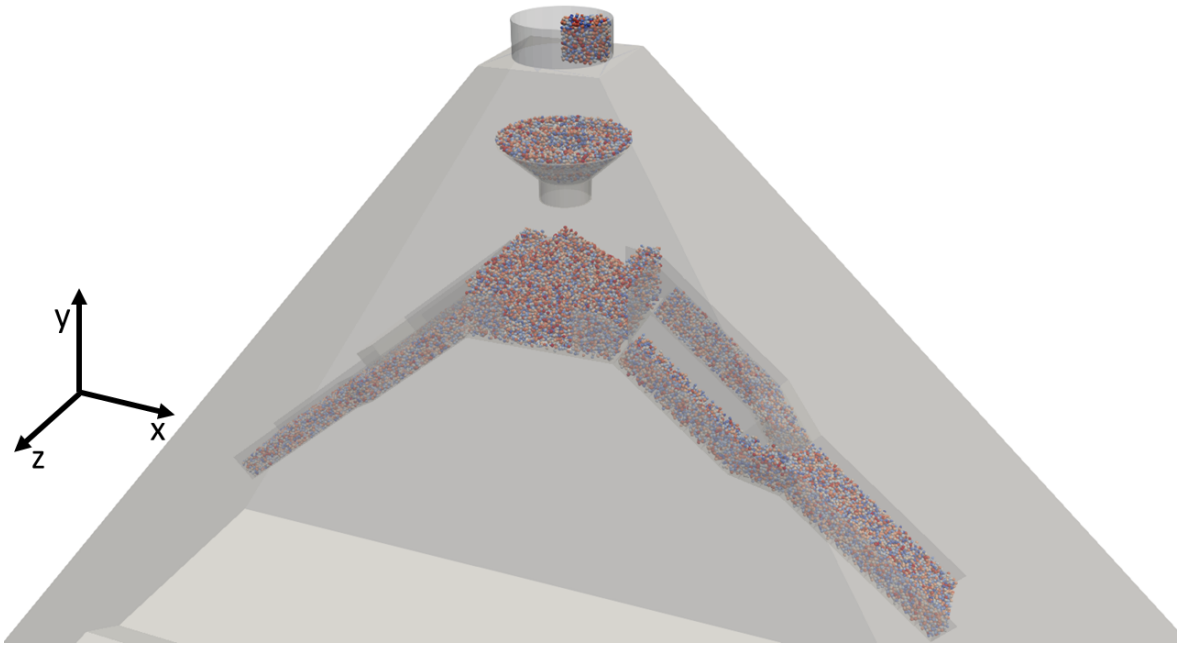


Figure 3.5: System when initially filled, with the first set inserted at the outer most x-position, coloured by particle radius

3.2 Results

The simulations have been running for about 7 months with a simulation time of a mere $60 - 70s$, with about $80 - 93$ sets of particles inserted. It has become clear that the centering rings are not centering the particles, but rather bouncing them to the opposite side of which they fell. This is clearly visible in Figure 3.6, where in case of the particles being dropped at the center or the positive x-direction on both sides of the distributor the amount of particles are roughly the same. In case of dropping the particles at the positive xz-direction or positive z-direction, however, both piles are definitely not equal and the particles end up a lot more in the negative z-direction. A set of particles falling through the centering rings is shown frame by frame in Figure 3.7, which shows this very clear as well. The figure suggests that extending the cylindrical bottom part of the centering rings will bounce the particles back to the center or at least keep them closer to it, however this has not been tested. In a highly unlikely scenario the average dropping position of the rotary valve is at the exact center and the effect of bouncing to the opposite side cancels each other out, however this defeats the whole purpose of the centering rings and should clearly not be a starting point.

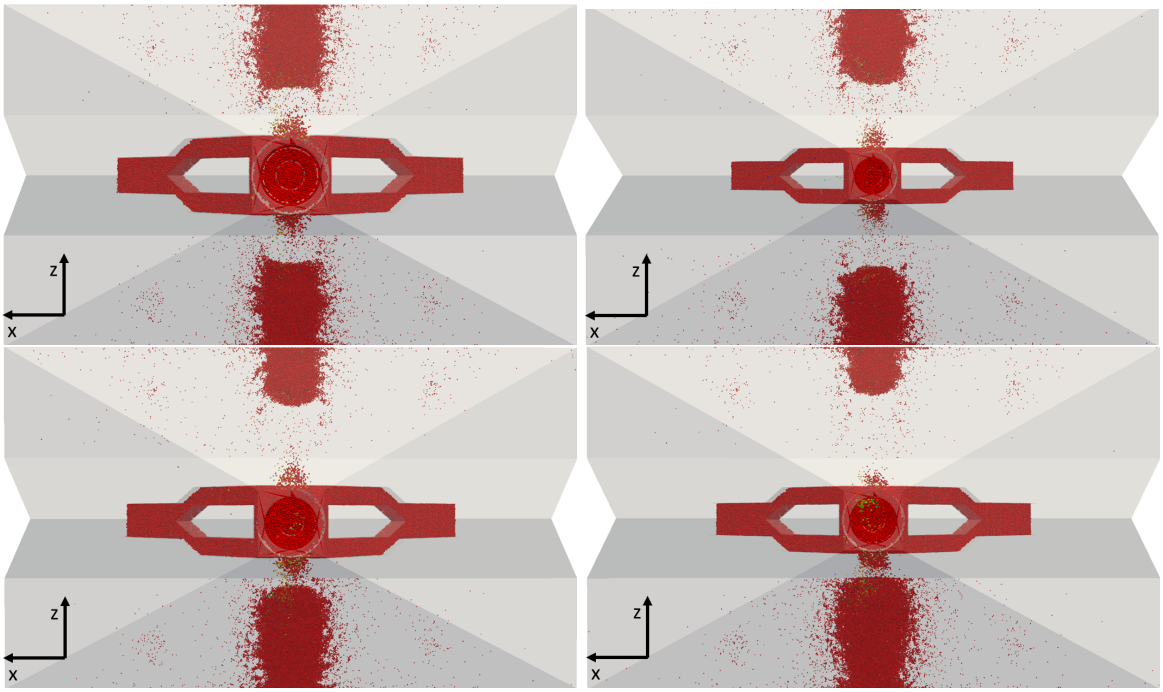


Figure 3.6: From left to right, top to bottom: particles dropped at the center, positive x-direction, positive xz-direction and positive z-direction at $t \approx 50s$

Most particles fall down the short arms of the distributor and it seems barely any particles are rolling down the long arms. For all four cases considered the part of the long arm located at the negative xz-direction is most likely to have particles

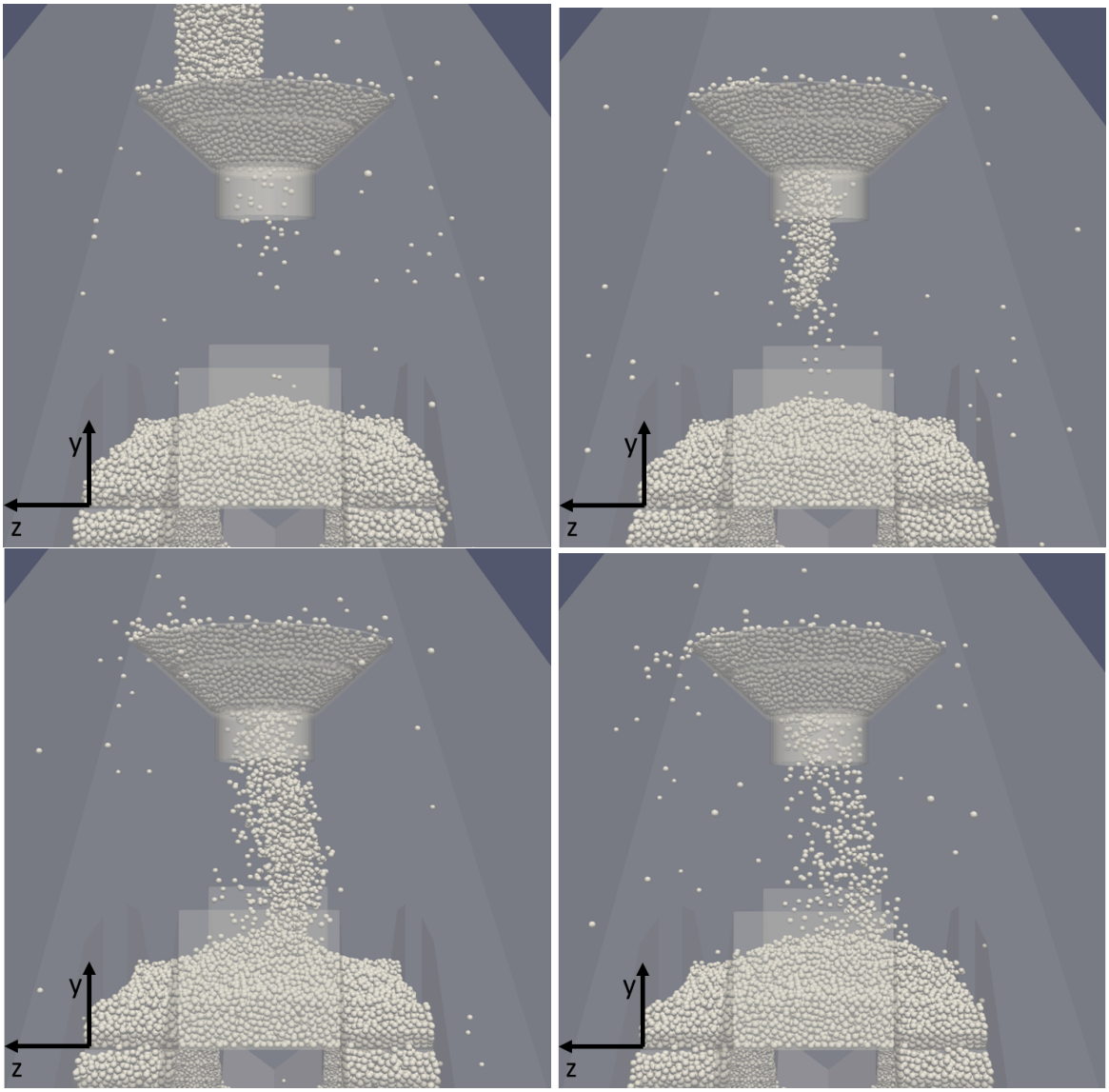


Figure 3.7: Frame by frame falling of one set of particles through the centering rings

rolling down on it and is therefore looked at for comparison. It turns out a drop at the positive xz-direction is the most likely case to have particles rolling down the long arm, see Figure 3.8. The least likely case is a drop at the positive z-direction, as shown in Figure 3.9, where most particles fall down the short arm. It is clear that even for the most likely scenario very little is happening, which means in general the long arms barely play a role. It is hard to say if anything significant would have happened if the simulations were kept running for even longer or that there would just be a pile forming at the top of the long arms preventing any possible flow along them. The simulations with bigger particles show the same result and do not give more insight.

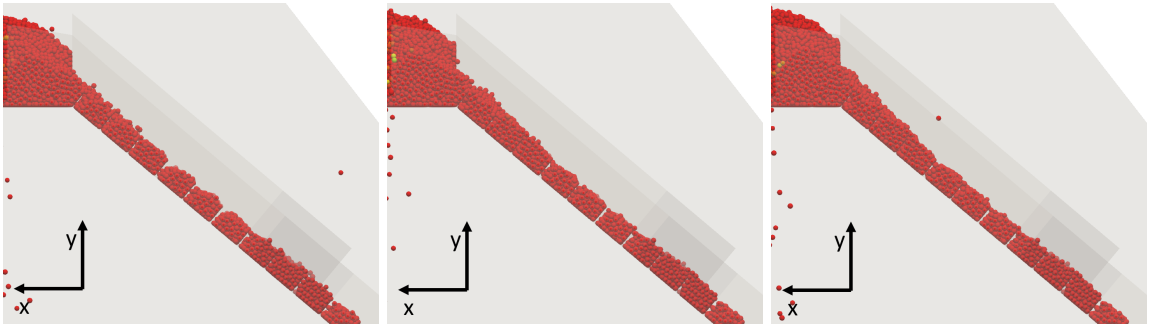


Figure 3.8: Particles dropped at the positive xz-direction, from left to right $t = 22.0s$, $t = 46.5s$, $t = 64.3s$ (positive z-axis pointing into paper)

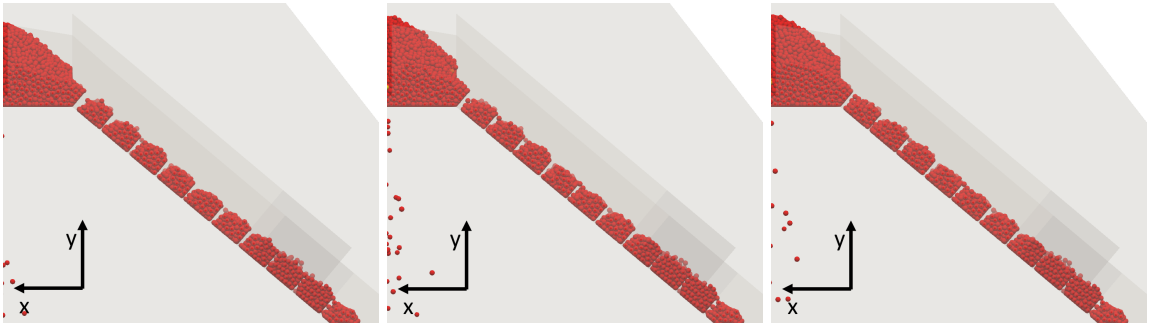


Figure 3.9: Particles dropped at the positive z-direction, from left to right $t = 22.2s$, $t = 44.6s$, $t = 61.2s$ (positive z-axis pointing into paper)

Chapter 4

Blast Furnace

In a blast furnace (BF) solid particles are slowly descending and discharged at the sides at the bottom, while continuously being added at the top. At the bottom center a pile of stationary particles will form, called a deadman. We want to simulate this using a 2D slot model with solid walls and a 3D model with periodic boundaries. Our model is the same as that of Zhang et al [2] and shown in Figure 4.1.

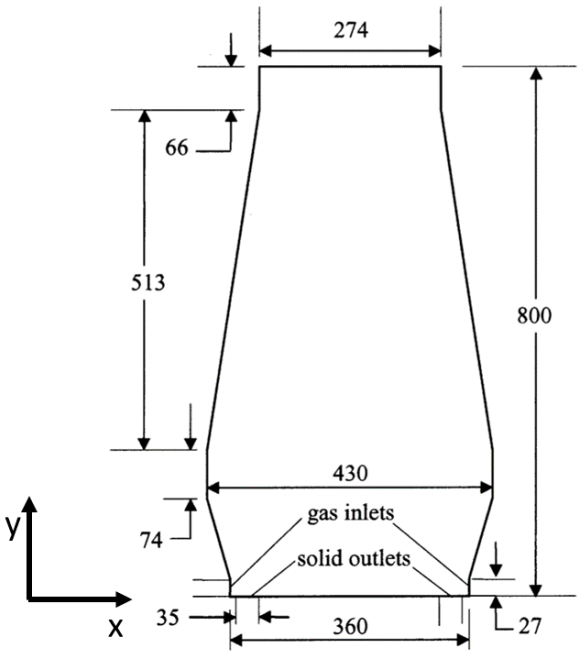


Figure 4.1: Schematic drawing of the BF model in mm [2]

In our simulation the particle discharge is created by calculating the mass that has to be removed every time step, using the discharge mass flow rate, and calculating to how many particles this corresponds. From the particles touching the bottom at the discharge regions the correct amount is removed and any remainder of the mass that has to be removed is saved for the next time step. Initially the BF is filled as much as possible and at the top an insertion boundary is placed, which

inserts particles with a higher rate than the discharge rate and in this way makes sure the BF is completely filled at all times. We try to keep as much parameters the same as that of Zhang's paper for the case of only glass beads, however to decrease computation time the particle size is increased as well as the discharge mass flow rate, which is taken the same as that of Zhou et al [3]. To summarize: a diameter of $6mm$, density of $2500kg/m^3$ and a discharge mass flow rate of $0.82kg/s$ are used. Two more cases are considered, namely the discharge mass flow rate halved and doubled. The drivers code of the simulation can be found in Appendix D.1, where the general simulation parameters are taken the same as for the waste gas cleaning plant.

In order to compare the different simulations in a simple and clear way, the data is coarse grained, which translates the discrete particle data to continuum fields, for more information see Tunuguntla et al [4] and Weinhart et al [5]. To find the right coarse grain width it is good practise to plot the density, velocity or stress at different points in the system as function of the coarse grain width. Regions in the plot which show a plateau for all points in the system indicate a good choice for the coarse grain width. For now the coarse grained data is only used for a quick visual comparison and therefore it is not important to focus too much on the details. Usually taking the coarse grain width equal to the mean particle diameter is a good choice and is therefore used here.

Coarse graining is done by time-averaging over a time interval where the system is in steady state. To find out when the system is in steady state the energy is plotted as function of time, as shown in Figure 4.2 for the case of solid walls and a discharge mass flow rate of $0.82kg/s$. A constant energy indicates a steady state, in this example starting around $t = 3s$. We are interested in data in the xy-plane, which means data in the z-direction is averaged. Furthermore a 100×100 grid is used.

The coarse grained velocity in x-, y- and z-direction is used to calculate the velocity magnitude, which is then plotted and can be found in Figure 4.3. The boundary of the deadman, which is plotted as a black dotted line, is where the velocity falls below $1/9$ particle diameter per minute, which is considered the critical solid velocity by Zhang et al [2]. In case of periodic boundaries the deadman boundary does not show up and from the plotted velocity magnitude it is also clear that the average velocity is higher than that of the case with solid walls. This makes perfect sense, since the particles have a lot of friction with the solid walls, while this is not the case for periodic boundaries. Previous studies, such as that of Zhou et al [3], agree with this observation and also mention a decrease is deadman size with a increase of discharge mass flow rate. This effect , although less pronounced as in Zhou's paper, is noticeable here as well.

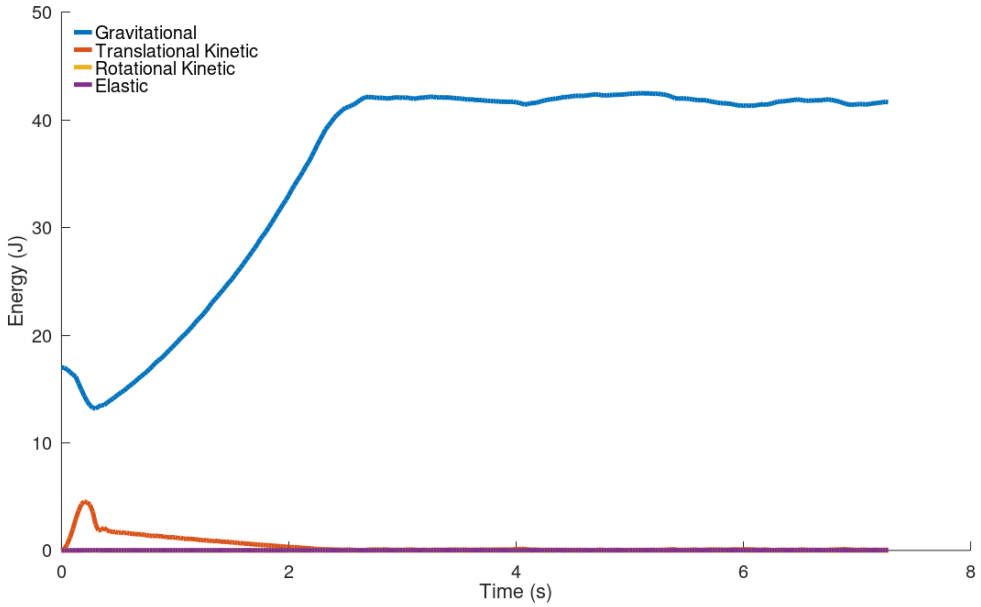


Figure 4.2: Energy of the BF plotted over time

The velocity at the sides and at the discharge regions is higher in case of solid walls compared to periodic boundaries and one reason for this is the way the discharge works. Only particles that touch the bottom are removed and because of the solid walls the effective area at the discharge regions is smaller, since the area is decreased by a particle radius from both walls. Having the same discharge mass flow rate with a smaller area results in a higher velocity. Another reason is the deadman itself, which narrows the channel through which the particles flow down.

In case of periodic boundaries the particles spill over the top of the BF, because somehow the insertion boundary placed at the top of the BF acts a little weird when combined with periodic boundaries. It seems not to have any influence on the final result and is definitely not important for us, since this is not an in-dept study, and has therefore been ignored.

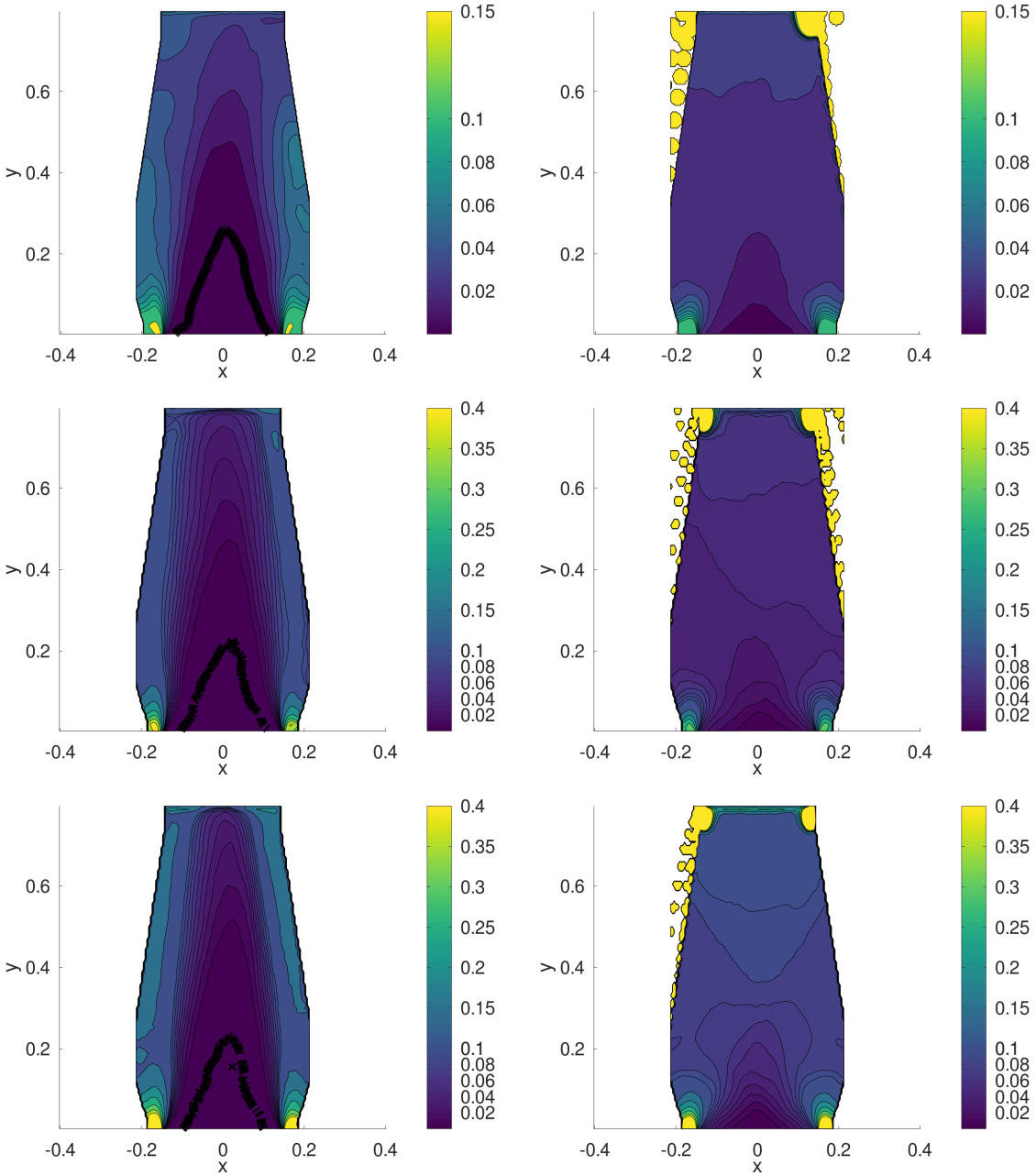


Figure 4.3: Coarse grained velocity magnitude for using solid walls (left) and periodic boundaries (right) and a discharge mass flow rate of $0.41kg/s$, $0.82kg/s$ and $1.64kg/s$ for respectively the top, middle and bottom plots.

Chapter 5

Conveyor Belt

In coke making the bulk density of the coal before entering the furnace is an important factor to know. As for now this is done by taking samples and measuring the bulk density by hand, however this is very labour intensive and therefore not done too often and thus not giving a very clear picture of the bulk density over time. A new way is in the early stages of development and involves a belt weigher and a profile scanner. The belt weigher measures the mass flow rate of the material and the profile scanner scans the outline of the material on the belt and in that way the volume flow rate can be obtained. Knowing the mass and volume flow rate the bulk density is easily calculated. Here we will look at how to get the volume flow rate from the profile scanner and in the end have a look at what the bulk density looks like over time.

The belt rests on a flat roller and two 45° angled rollers, from now on referred to as the baseline, as shown in Figure 5.1. Due to internal tension the belt lifts itself up when it is empty, shown as the dashed line, and when the belt is loaded it is assumed to be pushed completely in the corners. Also shown are a few of the lines at which the profile scanner measures the shortest distance from point to sensor. There are 640 evenly spread lines with a total angle of 57° and assuming the lines are distributed equally left and right of the vertical axis the angle of each line and thus the x- and y-position of each point are easily calculated. Figure 5.2 shows the outline and baseline for multiple timesteps. It is clear that the profile scanner has a range greater than just the belt and also scans part of the construction. Somehow the way the profile scanner works, there are lines forming from the edge of the belt to the construction. How this happens is not important, but it does show that first, the belt exceeds the angled rollers a bit, second, the belt does indeed lift itself up when it is empty, and third, the assumption that a loaded belt is completely pushed in the corners is already challenged a bit, since a fully filled belt exceeds the angled rollers less than a half filled belt.

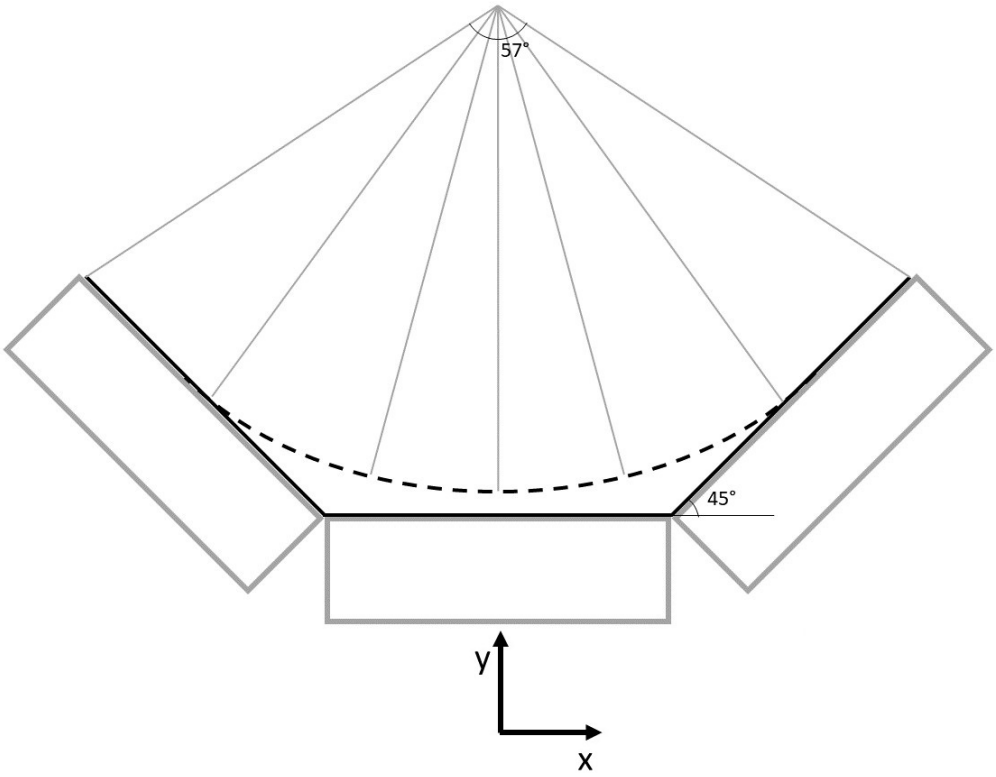


Figure 5.1: Schematic drawing of the rollers of the conveyor belt. The dashed and continuous line indicate an empty and loaded belt, respectively.

Since the outline represent a cross-section of the material at one point in time, multiplying the area of this cross-section with the known velocity of the belt gives the volume flow rate at that time. The baseline is at a known fixed position and the x-values of the outline are used to get the y-values of the corresponding points of the baseline. To approximate the area rectangles are defined between two consecutive points, their average y-value and the x-axis for both the outline and the baseline. From each rectangle of the outline the corresponding rectangle of the baseline is subtracted and if the result is greater than zero it is added to the total area of the cross-section. In this way only the points of the outline that are above the baseline, i.e. on the belt, are taken into account.

Sometimes the profile scanner returns a couple of nan-values, which are assigned a x- and y-position of respectively -600 and -1, so that they are definitely below and beside the belt and their rectangle area will now be negative and thus disregarded. In order to still get a fairly good approximation of the area the points are sorted with regards to their x-value and in this way, whenever a nan-value is present, two rectangles are not ignored, but rather approximated by one bigger rectangle.

The Python code for processing the data of the profile scanner to get the volume flow rate per timestep can be found in Appendix E.1. Now that the volume flow rate is known and saved in a csv file we can further process this in Octave, of which the

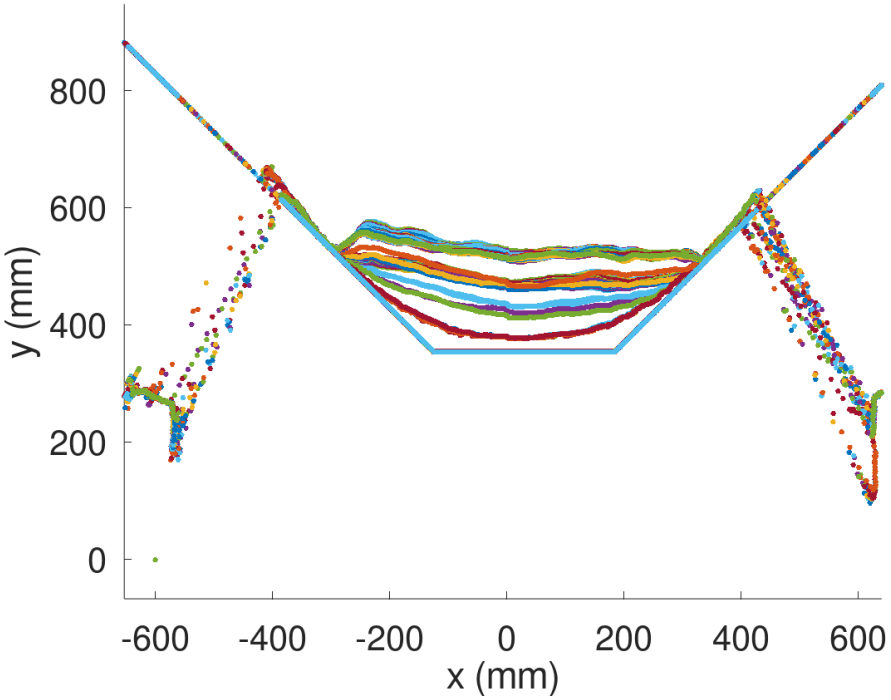


Figure 5.2: The profile of the belt for multiple timesteps. A empty, half filled and fully filled belt are clearly distinguishable.

code can be found in Appendix E.2. After removing a delay between the recorded data of the belt weigher and the profile scanner, the bulk density over time is easily calculated. A plot is shown in Figure 5.3 and it is clear that the calculated bulk density varies quite a bit even over the course of just a few hours. Comparing the plot with measurements done by hand does not give much insight, since these measurements are only done every few days. At best it can be said that the calculated bulk density is about the same. The plot shows clusters of points, from which it is suspected that the variation within is due to uncertainties in measurements and calculations, while the variation between clusters is due to a change in belt velocity or possibly an actual change in bulk density. The latter is not expected, since it is not expected that the bulk density changes that much during a day. A lot of measurements by hand are needed to thoroughly compare the actual and calculated bulk density.

The mass and volume flow rate both have the belt velocity in them and their values should therefore cancel out perfectly when calculating the bulk density. This is, however, not the case since it is highly likely that the belt weigher uses the actual belt velocity, unlike the volume flow rate calculation, which uses a fixed belt velocity. A clear indication for this is the fact that the mass flow rate at times with different amounts of material on the belt is about the same. With more material and a fixed

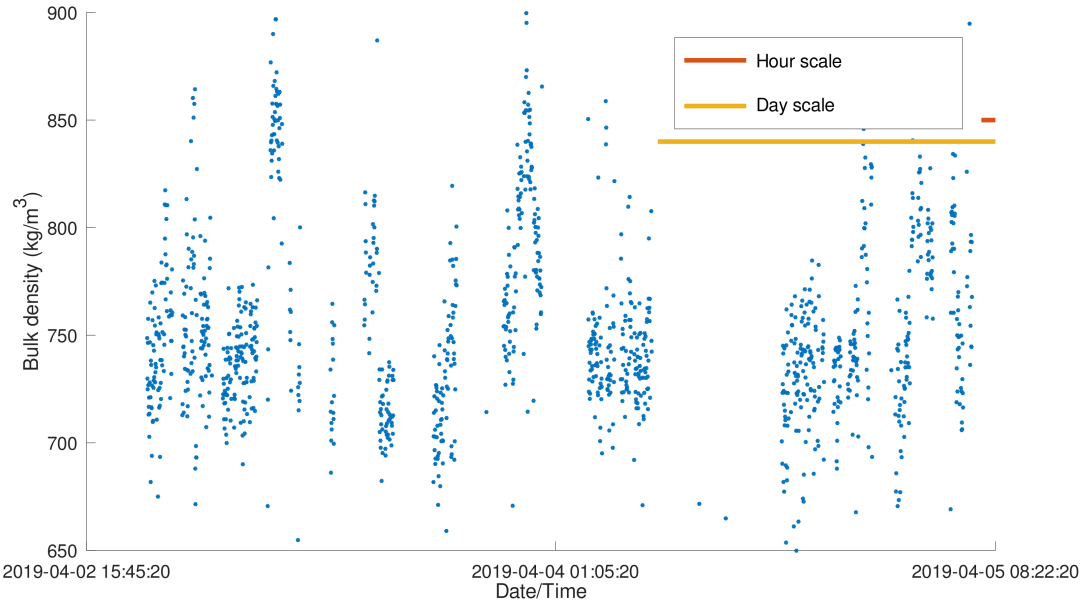


Figure 5.3: The bulk density over time. A day and hour scale are added to give a better idea of the time frames in the plot.

belt velocity one would expect the mass flow rate to increase, however it seems the belt is slowed down by the material on it. Furthermore, a lower actual belt velocity divided by a fixed belt velocity results in a lower calculated bulk density, which is consistent with observations as well. Future research has to show to what extend this explains the difference between the clusters of points of the calculated bulk density by integrating the actual belt velocity into the volume flow rate calculations. A less preferable but possibly quicker and easier way is to change the belt weigher to use a fixed belt velocity as well, however the mass and volume flow rate themselves will not be accurate.

The small variations in the bulk density have multiple causes, small variations in belt velocity being one of them. Another cause is the matching up of the belt weigher and profile scanner data. With the current data sets there is a time difference between the two and the delay was removed by matching up the times at which the belt was loaded for the first time after it had been empty. However, the belt weigher measures every minute and the profile scanner every 10 seconds, so there is a lot of uncertainty in this process. Coupling both systems would most likely solve this problem and can have a big influence. A third cause is the area calculation of the cross-section, since the assumption of the belt being fully pushed in the corners of the rollers likely does not always hold up as well as the sides of the belt curling up a bit instead of going straight like the angled rollers. The magnitude of this error depends on the amount of material on the belt. The last cause is the calibration of

the baseline, because as for now it is taken as a known fixed position and possible vibrations and heat expansion of the construction shifting the position of the profile scanner are not considered. This could give different results for day and night, summer and winter etc. Previously it was thought to use the middle of an empty belt for calibration, however this has a lot of vibration to it especially because the belt lifts itself up. A better way would be to use the surroundings, i.e. a rigid point next to the belt.

Chapter 6

Box Test

The box test mentioned in Section 2.1 is done again multiple times with different particle-particle sliding friction coefficient (SFC) and particle-particle rolling friction coefficient (RFC). They vary respectively from $0.05 - 0.5$, with a 0.05 increment, and $0.01 - 0.28$, with a 0.03 increment, resulting in a total of 100 combinations. The goal is to measure the AoR for each case and present this in a 3D plot with the SFC and RFC. To better represent the real physical world each simulation is run 10 times, with each time a different random seed, and their average AoR is used. Simulations are made for three different materials used a lot in steel making, namely sinter, pellet and coke, their difference being in bulk density, $1660kg/m^3$, $2150kg/m^3$ and $525kg/m^3$ respectively, and porosity, 0.45, 0.41 and 0.51 respectively. General simulation parameters are a particle radius of $2.5mm$, a particle-particle and particle-wall coefficient of restitution of respectively 0.5 and 0.25, a particle-wall sliding and rolling friction coefficient of respectively 0.2 and 0.1, a particle stiffness of $40000N/m$ and a timestep of $1e - 5s$.

Counting all possible combinations of parameters gives a total of 3000 simulations to run and it would, of course, be unrealistic to manually change each parameter and start each simulation by hand. Therefore a shell script is made in which only the name of the simulation has to be set, for example ”Sinter1”, and the bulk density and porosity of the material. The shell script then builds the MercuryDPM drivers code and copies the executable with a different name, so that the drivers code itself can be changed and compiled again without affecting the current simulation. A csv file containing all 100 combinations of SFCs and RFCs is read in and the executable is called with these parameters as well as the bulk density and porosity as additional arguments. Within the drivers code these additional arguments are read and assigned. So in order to run 100 simulations, only the name, bulk density and porosity have to be set within the shell script and the random seed in the drivers code only has to be changed once every 300 simulations. The shell script and the drivers code can be found in respectively Appendix F.1 and F.2. In the shell script it can be seen

that not only the SFC and RFC are read from the csv file, but also for example the coefficient of restitution. There is no real reason for this other than the fact that it was needed for a couple of simulations which were ran before and it might of course be useful in the future.

Getting the AoR was previously done by hand, but since we are now dealing with 3000 simulations this is simply not doable. Therefore a Python script is made, which gets the last vtu file of the simulations and takes a screenshot of it in Paraview. Then an Octave script can be run to take each screenshot and by making a binary image, cutting of the edges and fitting a line the AoR is obtained. By cutting of the edges is meant that 10% of both sides of the outline are ignored when fitting a line in order to prevent influence of possible abnormalities at the left wall and right edge of the box on the value of the AoR. For both scripts only the name has to be changed to get the screenshot and AoR of 100 simulations. It is therefore only a matter of minutes to get all 3000 AoRs. The Python and Octave scripts can be found in respectively Appendix F.3 and F.4.

Once all AoRs are known and saved in a csv file it is very easy to plot the average of the 10 different random seeds. The result is shown in Figure 6.1 and interestingly enough there does not seem to be much difference between the three different materials. The difference might however come down to small detail and the influence of other factors, such as the coefficient of restitution, which has not been investigated further.

The lower the RFC the faster the influence of the SFC flattens and visa versa. The SFC has a bigger influence than the RFC, which becomes especially clear for the lowest value of the SFC plotted here, where the influence of the RFC is basically zero. This makes sense since the RFC in general has a much smaller value than the SFC. It might be interesting to extend the range of both the SFC and RFC to go from 0 to 1 to get a complete picture of the influence of both.

From experiments is known that the AoR of sinter, pellet and coke are respectively $33°$, $26°$ and $35°$ and a plot of where they intersect the 3D plot is also shown in Figure 6.1. Any point on the line would result in the correct AoR, however one should not forget the influence other factors, such as the coefficient of restitution, might have. This has not been investigated further.

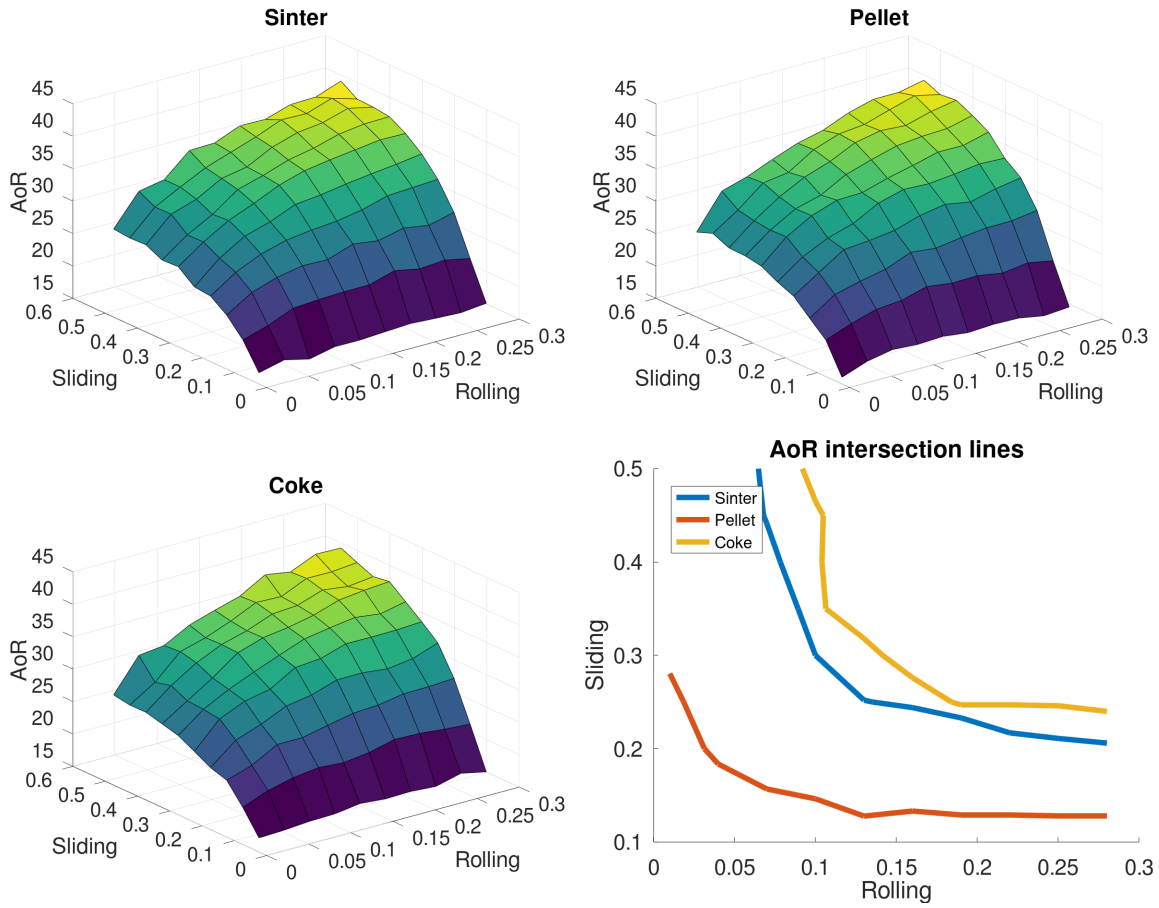


Figure 6.1: 3D plot of the AoR as function of the SFC and RFC for sinter, pellet and coke, as well as a plot of the lines where the 3D plot intersects the AoR found from experiments

Chapter 7

Conclusions and recommendations

7.1 Conclusions

The distribution of particles at the top of the waste gas cleaning plant is not done in an equal manner, mainly because the distributor itself needs the particles to be at the exact center and since the centering rings bounce the particles to the opposite side of which they are dropped this is not the case. Besides that, barely any particles were rolling down the long arms of the distributor.

For a 2D slot model of a blast furnace the solid walls have a huge influence on the formation of the deadman, since using periodic boundaries results in no deadman at all. This, and the fact that an increasing discharge mass flow rate results in a decreasing deadman size, is all in agreement with previous studies [2] [3].

The result of using a profile scanner and a belt weigher to find the bulk density through time is very promising. Small variations are suspected to be from little uncertainties in measurements and a lack of fine tuning and bigger variations, mainly between clusters of data points, are suspected to be from changes in belt velocity or possibly indicate an actual change in bulk density.

Increasing values of the SFC or RFC result in an increase of AoR, although the curve flattens for higher values and having a really small value of one could cancel out the influence of the other. The SFC has a greater influence that the RFC, which is in agreement with reality. When comparing the AoR with experiments a higher value of the SFC has to be compensated with lowering the value of the RFC and visa versa. The different materials considered produce an almost identical AoR plot.

7.2 Recommendations

The simulations of the waste gas cleaning plant took a very long time for what are fairly basic results. Decreasing computation time is an important improvement and

the first step towards that is to have a closer look at the initially filling up of areas with stationary particles and make sure they are rather a bit overflown than not completely filled. Another improvement is using a particle shape and size closer to reality, as well as doing more research at what simulation parameters result in the most realistic particle behaviour. Future research questions might be: How can the centering rings be improved to properly center the particles? Does the way the particles leave the rotary valve influence the distribution of them? Do the long arms of the distributor actually play a roll?

Although the research on the blast furnace was not an in-dept study some improvements are still in place, for example taking a closer look at what good values for the parameters used in coarse graining are instead of using the default ones. Or, like previous studies, use different coloured layers of particles so that coarse graining is unnecessary. Furthermore it might be good to have the actual discharge mass flow rate exactly the same for both solids walls and periodic boundaries by changing the amount of particles that are removed each timestep or by changing the wall placement, so that the effective discharge area is the same. Future research questions might be: How do both models compare to an actual 3D model? How do particle and wall properties effect the formation of a deadman?

Taking the actual belt velocity into account is an important first step to improve the bulk density calculations. Other steps are the matching up of belt weigher and profile scanner data, improving the area calculation of the cross-section and calibration of the baseline. Of course, a thorough comparison between the actual and calculated bulk density is needed and gives a lot of insight. A future research question might be: How can the current research be used to live output an accurate bulk density.

Extending the range of the SFC and RFC gives a bit more certainty about for what values of one the influence of the other is zero as well as at what point taking higher values makes no difference, i.e. the plot flattens. Future research questions might be: How do other factors, such as the coefficient of restitution, influence the angle of repose? When does the particle density play a role on the value of the angle of repose?

Bibliography

[1] T. Weinhart, D. R. Tunuguntla, M. P. v. Schrojenstein-Lantman, A. J. v. d. Horn, I. F. C. Denissen, C. R. Windows-Yule, A. C. d. Jong, and A. R. Thornton, "Mercurydpm: A fast and flexible particle solver part a: Technical advances," *Springer Proceedings in Physics*, vol. 188, pp. 1353–1360, 2016.

[2] S. J. Zhang, A. B. Yu, P. Zulli, B. Wright, and P. Austin, "Numerical simulation of solids flow in a blast furnace," *Applied Mathematical Modelling*, vol. 26, no. 2, pp. 141–154, 2002.

[3] Z. Zhou, H. Zhu, B. Wright, D. Pinson, and P. Zulli, "Discrete particle simulation of solid flow in a model blast furnace," *ISIJ International*, vol. 45, no. 12, pp. 1828–1837, 2005.

[4] D. R. Tunuguntla, A. R. Thornton, and T. Weinhart, "From discrete elements to continuum fields: Extension to bidisperse systems," *Computational Particle Mechanics*, vol. 3, no. 3, pp. 349–365, 2016.

[5] T. Weinhart, A. R. Thornton, S. Luding, and O. Bokhove, "From discrete particles to continuum fields near a boundary," *Granular Matter*, vol. 14, no. 2, pp. 289–294, 2012.

Appendix A

BlueScope Steel

The Port Kembla steelworks have been a staple for over a century and produces flat steel. As of 2002 it is owned by BlueScope Steel and with branches in 24 countries the Port Kembla steelworks still is the largest, with each year 2.6 million tonnes of raw steel being produced. My position was at the Coke and Ironmaking Technology, with me being mostly at the office working on my projects. Of course, I have had a few tours to, among other thing, the waste gas cleaning plant, sinter plant, blast furnace and slab making and furthermore did some quick experiments to get a feel of the scale of the plants and material of what I was dealing with in these projects. I was mostly working on my own, however my supervisor David Pinson was always available for questions at the office or via email or phone. Every week we had a short chat about my progress and David shared some other general steelmaking knowledge. This way of working worked really well for me, since David let me dive into it and figure things out on my own, without constantly breathing in my neck asking for results. He really just guided me, made sure I was on the right track, but let me walk the track myself. In the end we were both very happy with the results obtained that far.

Appendix B

Experiments

B.1 Drivers code of the box test simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 Simple cubic box with periodic boundaries in the y-direction. Once the particles
28 are settled the right side wall is removed and the particles fall away to the side
29 and down. Once they are a bit away from the box they are removed.
30 Goal: measure the angle of repose and compare with experimental data.
31 */
32
33 #include "Species/LinearViscoelasticFrictionSpecies.h"
34 #include "Mercury3D.h"
35 #include "Particles/BaseParticle.h"
36 #include "Walls/InfiniteWall.h"
37 #include "Walls/IntersectionOfWalls.h"
38 #include "Boundaries/PeriodicBoundary.h"
39 #include "Boundaries/DeletionBoundary.h"
40
41 class Box : public Mercury3D
42 {
43 public:
44
45     void setupInitialConditions() override
46     {
47         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
48
49         auto speciesWall =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
50         auto speciesPar =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
51
52         // Wall species
53         speciesWall->setDensity(rho);
54         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mass);
55
56         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
57         speciesWall->setSlidingFrictionCoefficient(particleWallFriction);
58         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
59
60         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
61         speciesWall->setRollingFrictionCoefficient(rollFricCoeff);
62         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
63
64         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
65         speciesWall->setTorsionFrictionCoefficient(0.1);
66         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
67
68         // Particle species
69         speciesPar->setDensity(rho);
70         speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORPar,mass);
71
```

```
72          speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
73          speciesPar->setSlidingFrictionCoefficient(particleParticleFriction);
74          speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
75
76          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
77          speciesPar->setRollingFrictionCoefficient(rollFricCoeff);
78          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
79
80          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
81          speciesPar->setTorsionFrictionCoefficient(0.01);
82          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
83
84          // Wall and Particle species
85          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
86
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mas
            s);
87
88
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
89          speciesWallAndPar->setSlidingFrictionCoefficient(particleWallFriction);
90
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
91
92
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
93          speciesWallAndPar->setRollingFrictionCoefficient(rollFricCoeff);
94
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
95
96
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
97          speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
98
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
99
100
101         // Wall setup
102         InfiniteWall w0;
103         w0.setSpecies(speciesWall);
104         w0.set(Vec3D(-1.0,0.0,0.0),Vec3D(getXMin(),0,0)); // Left wall
105         wallHandler.copyAndAddObject(w0);
106         w0.set(Vec3D(1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Right wall (index 1),
            which will be removed
107         wallHandler.copyAndAddObject(w0);
108         w0.set(Vec3D(0.0,0.0,1.0),Vec3D(0,0,2.0*getZMax())); // Top wall, to prevent
            particles from shooting up too high while settling
109         wallHandler.copyAndAddObject(w0);
110
111         IntersectionOfWalls w1;
112         w1.setSpecies(speciesWall);
113         w1.addObject(Vec3D(0.0,0.0,-1.0),Vec3D(0,0,getZMin())); // Bottom wall
114         w1.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Bottom wall:
            right edge
115         wallHandler.copyAndAddObject(w1);
116
117         // Solid boundaries y-direction
118 //        IntersectionOfWalls w2;
119 //        w2.setSpecies(speciesWall);
120 //        w2.addObject(Vec3D(0.0,1.0,0.0),Vec3D(0,getYMax(),0)); // Back wall
121 //        w2.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Back wall:
    right edge
122 //        wallHandler.copyAndAddObject(w2);
123 //
124 //        IntersectionOfWalls w3;
125 //        w3.setSpecies(speciesWall);
```

```
126   //        w3.addObject(Vec3D(0.0,-1.0,0.0),Vec3D(0,getYMin(),0)); // Front wall
127   //        w3.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Front wall:
      right edge
128   //        wallHandler.copyAndAddObject(w3);
129
130           // Periodic boundaries y-direction
131           PeriodicBoundary b0;
132           b0.set(Vec3D(0.0,1.0,0.0),getYMin(),getYMax());
133           boundaryHandler.copyAndAddObject(b0);
134
135
136           // Deletion boundary at right side wall and bottom to remove particles
              outside the box
137           DeletionBoundary db0;
138           db0.set(Vec3D(1,0,0),getXMax()+5*radius);// Definitely outside the box and
              not interfering with particles partly inside the box
139           boundaryHandler.copyAndAddObject(db0);
140           db0.set(Vec3D(0,0,-1),getZMin()-5*radius); // Definitely below minimum and
              not interfering with particles partly inside the box
141           boundaryHandler.copyAndAddObject(db0);
142
143
144           // Particle setup
145           int numPar =
              fillFraction*(std::abs(getXMax()-getXMin()))*(std::abs(getYMax()-getYMin()))*(
              std::abs(getZMax()-getZMin()))/(4./3*constants::pi*pow(radius,3.0)); //
              Number of particles to be inserted
146           int numParInserted = 0; // Number of particle inserted
147           Vec3D pos; // Position particle
148           while( numParInserted < numPar )
149           {
150               double radiusPar =
                  random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
151               BaseParticle p0;
152               p0.setSpecies(speciesPar);
153               p0.setRadius(radiusPar);
154
155               int failCounter = 0;
156               do
157               {
158                   pos.X =
                      random.getRandomNumber(getXMin()+radiusPar,getXMax()-radiusPar);
159                   pos.Y = random.getRandomNumber(getYMin(),getYMax());
160                   pos.Z = random.getRandomNumber(getZMin()+radiusPar,1.5*getZMax());
161                   p0.setPosition(pos);
162                   p0.setVelocity(Vec3D(0,0,0));
163
164                   failCounter++;
165                   if (failCounter==1000)
166                       break;
167               }
168               while (!checkParticleForInteraction(p0));
169
170               particleHandler.copyAndAddObject(p0);
171
172               numParInserted++;
173           }
174
175           std::cout << "Finished creating particles" << std::endl;
176           std::cout << "Number of particles inserted: " << numParInserted << std::endl;
177           std::cout << "Particles settling down" << std::endl;
178           step = 2; // Allow next step to be executed
179           checkTime = getTime() + .1; // 0.1 Time to check the total kinetic energy
180       }
181
182       void actionsBeforeTimeStep() override
183       {
184           // Open the gate once the kinetic energy is low enough to consider the
              particles settled (value found by trial and error)
185           if (step==2) // To stop checking the kinetic energy once the particles are
              settled
186           {
187               if (getTime() > checkTime) // Only check at certain times
```

```
188                 {
189                     std::cout << "Current KE: " << getKineticEnergy() << std::endl;
190                     if (getKineticEnergy() < 0.0001) // For 0.0001 the particles are
                        settled quite well
191                     {
192                         step = 3; // Allow next step to be executed
193                         std::cout << "Particles settled" << std::endl;
194                         wallHandler.removeObject(1); // Remove right wall
195                         std::cout << "Gate open" << std::endl;
196                     }
197                     else
198                     {
199                         checkTime = getTime() + .1;
200                     }
201                 }
202             }
203         }
204 
205 
206         void setFillFraction (double ff)
207         {
208             fillFraction = ff;
209         }
210 
211         void setRadiusVariation (double rv)
212         {
213             radiusVariation = rv;
214         }
215 
216         void setDensity (double r)
217         {
218             rho = r;
219         }
220 
221         void setRadius (double r)
222         {
223             radius = r;
224         }
225 
226         void setCOR (double wallCOR, double parCOR)
227         {
228             CORWall = wallCOR;
229             CORPar = parCOR;
230         }
231 
232         void setCollisionTime (double coltime)
233         {
234             tc = coltime;
235         }
236 
237         void setStiffness (double k)
238         {
239             stiffness = k;
240         }
241 
242         void setFrictionCoeff (double pwf, double ppf, double rfc)
243         {
244             particleWallFriction = pwf;
245             particleParticleFriction = ppf;
246             rollFricCoeff = rfc;
247         }
248 
249         double getLargestParticleDiameter ()
250         {
251             double lpd = 2*(radius+radiusVariation);
252             return lpd;
253         }
254 
255     private:
256 
257         double fillFraction;
258         double rho, radius, radiusVariation, mass;
259 
```

```
260         double CORWall, CORPar, tc, stiffness;
261
262         double particleWallFriction, particleParticleFriction, rollFricCoeff;
263
264         double checkTime;
265         int step;
266     };
267
268     int main(int argc, char *argv[])
269     {
270         // Problem setup
271         Box problem;
272
273         problem.setFillFraction(0.55);
274         problem.setRadiusVariation(0.0005); // 0.5mm added to or subtracted from radius
275         problem.setDensity(900);
276         problem.setRadius(0.003);
277         problem.setCOR(0.25,0.25); // Wall, particle
278         problem.setFrictionCoeff(0.25,0.25,0.40); // SET sliding particle-wall, sliding
            particle-particle, rolling friction coefficient
279         problem.setStiffness(40000); // SET 40000
280
281         problem.setName("Box"); // SET: Box_pwf025_ppf025_rfc040
282         problem.setSystemDimensions(3);
283         problem.setGravity(Vec3D(0.0,0.0,-9.81));
284         problem.setXMin(0.0);
285         problem.setYMin(0.0);
286         problem.setZMin(0.0);
287         problem.setXMax(0.15);
288         problem.setYMax(5.*problem.getLargestParticleDiameter());
289         problem.setZMax(0.15);
290         problem.setTimeMax(5.0); // SET
291         problem.setTimeStep(1.0e-5); // SET 1.0e-5
292
293         problem.setSaveCount(1000); // SET
294         problem.dataFile.setFileType(FileType::ONE_FILE);
295         problem.restartFile.setFileType(FileType::ONE_FILE);
296         problem.fStatFile.setFileType(FileType::NO_FILE);
297         problem.eneFile.setFileType(FileType::NO_FILE);
298
299         problem.setXBallsAdditionalArguments(" -solidf -v0 -s 8");
300
301         problem.solve(argc, argv);
302
303         return 0;
304     }
305
```

B.2 Drivers code of the tube lift simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 A tube standing on the ground is filled with particles and once settled the tube
28 is slowly lifted upwards so that the particles spill out from the bottom.
29 Goal: measure the angle of repose and compare with experimental data.
30 */
31
32 #include "Species/LinearViscoelasticFrictionSpecies.h"
33 #include "Mercury3D.h"
34 #include "Particles/BaseParticle.h"
35 #include "Walls/InfiniteWall.h"
36 #include "Walls/AxisymmetricIntersectionOfWalls.h"
37
38 class Cylinder : public Mercury3D
39 {
40 public:
41
42     void setupInitialConditions() override
43     {
44         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
45
46         auto speciesWall =
            speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
47         auto speciesPar =
            speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
48
49         // Wall species
50         speciesWall->setDensity(rho);
51         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mass);
52
53         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
54         speciesWall->setSlidingFrictionCoefficient(particleWallFriction);
55         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
56
57         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
58         speciesWall->setRollingFrictionCoefficient(rollFricCoeff);
59         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
60
61         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
62         speciesWall->setTorsionFrictionCoefficient(0.1);
63         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
64
65         // Particle species
66         speciesPar->setDensity(rho);
67         speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORPar,mass);
68
69         speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
70         speciesPar->setSlidingFrictionCoefficient(particleParticleFriction);
71         speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
```

```
72
73          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
74          speciesPar->setRollingFrictionCoefficient(rollFricCoeff);
75          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
76
77          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
78          speciesPar->setTorsionFrictionCoefficient(0.01);
79          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
80
81          // Wall and Particle species
82          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
83
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mas
            s);
84
85
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
86          speciesWallAndPar->setSlidingFrictionCoefficient(particleWallFriction);
87
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
88
89
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
90          speciesWallAndPar->setRollingFrictionCoefficient(rollFricCoeff);
91
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
92
93
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
94          speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
95
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
96
97
98          // Wall setup
99          InfiniteWall w0;
100         w0.setSpecies(speciesWall);
101         w0.set(Vec3D(0.0,0.0,-1.0),Vec3D(0.0,0.0,getZMin())); // Bottom wall
102         wallHandler.copyAndAddObject(w0);
103         w0.set(Vec3D(0.0,0.0,1.0),Vec3D(0,0,2.0*getZMax())); // Top wall, to prevent
            particles from shooting up too high while settling
104         wallHandler.copyAndAddObject(w0);
105
106         Vec3D cylinderCenter = {0.5*(getXMin()+getXMax()),
107                                 0.5*(getYMin()+getYMax()),
108                                 getZMin()};
109
110         AxisymmetricIntersectionOfWalls w1;
111         w1.setSpecies(speciesWall);
112         w1.setPosition(cylinderCenter);
113         w1.setOrientation(Vec3D(0.0,0.0,1.0));
114         w1.addObject(Vec3D(1,0,0),Vec3D(cylinderRadius,0.0,0.0)); // Cylinder wall
115         w1.addObject(Vec3D(0,0,1),Vec3D(2.0*cylinderRadius,0,0)); // Cylinder wall:
            bottom edge
116         wallHandler.copyAndAddObject(w1);
117
118
119         // Particle setup
120         int numPar =
            fillFraction*constants::pi*pow(cylinderRadius,2.0)*(std::abs(getZMax()-getZMin
            ()))/(4./3*constants::pi*pow(radius,3.0)); // Number of particles to be
            inserted
121         int numParInserted = 0; // Number of particle inserted
122         Vec3D pos; // Position particle
123         double r, theta, z; // For particle position
124         while( numParInserted < numPar )
```

```
125         {
126             double radiusPar =
                random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
127             BaseParticle p0;
128             p0.setSpecies(speciesPar);
129             p0.setRadius(radiusPar);
130 
131             int failCounter = 0;
132             do
133             {
134                 r = random.getRandomNumber(0,cylinderRadius-radiusPar);
135                 theta = random.getRandomNumber(0,2.0*constants::pi);
136                 z = random.getRandomNumber(getZMin()+radiusPar,1.5*getZMax());
137 
138                 pos.X = cylinderCenter.X+r*cos(theta);
139                 pos.Y = cylinderCenter.Y+r*sin(theta);
140                 pos.Z = z;
141                 p0.setPosition(pos);
142                 p0.setVelocity(Vec3D(0,0,0));
143 
144                 failCounter++;
145                 if (failCounter==1000)
146                     break;
147             }
148             while (!checkParticleForInteraction(p0));
149 
150             particleHandler.copyAndAddObject(p0);
151 
152             numParInserted++;
153         }
154 
155         std::cout << "Finished creating particles" << std::endl;
156         std::cout << "Number of particles inserted: " << numParInserted << std::endl;
157         std::cout << "Particles settling down" << std::endl;
158         step = 2; // Allow next step to be executed
159         checkTime = getTime() + 0.1; // 0.1 Time to check the total kinetic energy
160     }
161 
162     void actionsBeforeTimeStep() override
163     {
164         if (step==2) // To stop checking the kinetic energy once the particles are
                settled
165         {
166             if (getTime() > checkTime)
167             {
168                 std::cout << "Current KE: " << getKineticEnergy() << std::endl;
169                 if (getKineticEnergy() < 0.0001)
170                 {
171                     step = 3;
172                     std::cout << "Particles settled" << std::endl;
173 
                        wallHandler.getObject(2)->setVelocity(Vec3D(0.0,0.0,cylinderVeloci
                        ty));
174                     std::cout << "Cylinder moving up" << std::endl;
175                 }
176                 else
177                 {
178                     checkTime = getTime() + .1;
179                 }
180             }
181         }
182     }
183 
184     void setFillFraction (double ff)
185     {
186         fillFraction = ff;
187     }
188 
189     void setRadiusVariation (double rv)
190     {
191         radiusVariation = rv;
192     }
193 
```

```
194         void setDensity (double r)
195         {
196             rho = r;
197         }
198 
199         void setRadius (double r)
200         {
201             radius = r;
202         }
203 
204         void setCOR (double wallCOR, double parCOR)
205         {
206             CORWall = wallCOR;
207             CORPar = parCOR;
208         }
209 
210         void setCollisionTime (double coltime)
211         {
212             tc = coltime;
213         }
214 
215         void setStiffness (double k)
216         {
217             stiffness = k;
218         }
219 
220         void setFrictionCoeff (double pwf, double ppf, double rfc)
221         {
222             particleWallFriction = pwf;
223             particleParticleFriction = ppf;
224             rollFricCoeff = rfc;
225         }
226 
227         void setCylinderRadius (double cr)
228         {
229             cylinderRadius = cr;
230         }
231 
232         double getCylinderRadius()
233         {
234             return cylinderRadius;
235         }
236 
237         void setCylinderVelocity (double cv)
238         {
239             cylinderVelocity = cv;
240         }
241 
242     private:
243 
244         double fillFraction;
245         double rho, radius, radiusVariation, mass;
246 
247         double CORWall, CORPar, tc, stiffness;
248 
249         double particleWallFriction, particleParticleFriction, rollFricCoeff;
250 
251         double checkTime;
252         int step;
253 
254         double cylinderRadius, cylinderVelocity;
255     };
256 
257     int main(int argc, char *argv[])
258     {
259 
260         // Problem setup
261         Cylinder problem;
262 
263         problem.setFillFraction(0.50);
264         problem.setRadiusVariation(0.0005); // 0.5mm added to or subtracted from radius
265         problem.setDensity(900);
266         problem.setRadius(0.003);
```

```
267       problem.setCOR(0.9,0.831); // Wall, particle
268       problem.setCylinderRadius(0.05);
269       problem.setCylinderVelocity(0.001);
270       problem.setFrictionCoeff(0.25,0.25,0.50); // SET sliding particle-wall, sliding
          particle-particle, rolling friction coefficient
271       problem.setStiffness(40000); // SET 40000
272
273       problem.setName("Cylinder2"); // SET: Cylinder_pwf025_ppf025_rfc040
274       problem.setSystemDimensions(3);
275       problem.setGravity(Vec3D(0.0,0.0,-9.81));
276       problem.setXMin(0.0);
277       problem.setYMin(0.0);
278       problem.setZMin(0.0);
279       problem.setXMax(2.*problem.getCylinderRadius());
280       problem.setYMax(2.*problem.getCylinderRadius());
281       problem.setZMax(0.1);
282       problem.setTimeMax(55.0); // SET
283       problem.setTimeStep(1e-5); // SET 1.0e-5
284
285       problem.setSaveCount(1000); // SET
286       problem.dataFile.setFileType(FileType::ONE_FILE);
287       problem.restartFile.setFileType(FileType::ONE_FILE);
288       problem.fStatFile.setFileType(FileType::NO_FILE);
289       problem.eneFile.setFileType(FileType::NO_FILE);
290
291       problem.setXBallsAdditionalArguments(" -solidf -v0 -s 7 -moh 115"); // -moh
          horizontal offset of negative .. pixels
292
293       problem.solve(argc, argv);
294
295       return 0;
296   }
297
```

B.3 Drivers code of the bridging test simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 Hopper filled with particles, which (once settled) drop away from the bottom
28 through different diameter orifices and are then removed.
29 Goal: find out for which diameter orifice bridging starts happening and compare
30 with experimental data.
31 */
32
33 #include "Species/LinearViscoelasticFrictionSpecies.h"
34 #include "Mercury3D.h"
35 #include "Particles/BaseParticle.h"
36 #include "Walls/InfiniteWall.h"
37 #include "Walls/AxisymmetricIntersectionOfWalls.h"
38 #include "Boundaries/DeletionBoundary.h"
39
40 class Bridging : public Mercury3D
41 {
42 public:
43
44     void setupInitialConditions() override
45     {
46         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
47
48         auto speciesWall =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
49         auto speciesPar =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
50
51         // Wall species
52         speciesWall->setDensity(rho);
53         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mass);
54
55         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
56         speciesWall->setSlidingFrictionCoefficient(particleWallFriction);
57         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
58
59         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
60         speciesWall->setRollingFrictionCoefficient(rollFricCoeff);
61         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
62
63         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
64         speciesWall->setTorsionFrictionCoefficient(0.1);
65         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
66
67         // Particle species
68         speciesPar->setDensity(rho);
69         speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORPar,mass);
70
71         speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
```

```
72          speciesPar->setSlidingFrictionCoefficient(particleParticleFriction);
73          speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
74
75          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
76          speciesPar->setRollingFrictionCoefficient(rollFricCoeff);
77          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
78
79          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
80          speciesPar->setTorsionFrictionCoefficient(0.01);
81          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
82
83          // Wall and Particle species
84          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
85
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mas
            s);
86
87
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
88          speciesWallAndPar->setSlidingFrictionCoefficient(particleWallFriction);
89
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
90
91
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
92          speciesWallAndPar->setRollingFrictionCoefficient(rollFricCoeff);
93
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
94
95
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
96          speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
97
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
98
99
100         // Wall setup
101         Vec3D center = {0.5*(getXMin()+getXMax()),
102                         0.5*(getYMin()+getYMax()),
103                         getZMin()};
104
105         InfiniteWall w0;
106         w0.setSpecies(speciesWall);
107         w0.set(Vec3D(0,0,-1),Vec3D(center.X,center.Y,-orificeHeight)); // Bottom
            wall, which will be removed
108         wallHandler.copyAndAddObject(w0);
109
110         AxisymmetricIntersectionOfWalls w1;
111         w1.setSpecies(speciesWall);
112         w1.setPosition(center);
113         w1.setOrientation(Vec3D(0,0,1));
114         w1.addObject(Vec3D(1,0,0),Vec3D(outerCylinderRadius,0,0)); // Outer cylinder
115         wallHandler.copyAndAddObject(w1);
116
117         AxisymmetricIntersectionOfWalls w2;
118         w2.setSpecies(speciesWall);
119         w2.setPosition(center);
120         w2.setOrientation(Vec3D(0,0,1));
121         std::vector<Vec3D> w2Points(3);
122         w2Points[0] = Vec3D(outerCylinderRadius,0,topConeHeight);
123         w2Points[1] = Vec3D(innerCylinderRadius,0,bottomConeHeight);
124         w2Points[2] = Vec3D(innerCylinderRadius,0,0);
125         w2.createOpenPrism(w2Points); // Cone and inner cylinder
126         wallHandler.copyAndAddObject(w2);
127
128         AxisymmetricIntersectionOfWalls w3;
```

```
129         w3.setSpecies(speciesWall);
130         w3.setPosition(center);
131         w3.setOrientation(Vec3D(0,0,1));
132         std::vector<Vec3D> w3Points(4);
133         w3Points[0] = Vec3D(outerCylinderRadius,0,0); // outerCylinderRadius is used
            just a random point greater than orificeRadius
134         w3Points[1] = Vec3D(orificeRadius,0,0);
135         w3Points[2] = Vec3D(orificeRadius,0,-orificeHeight);
136         w3Points[3] = Vec3D(outerCylinderRadius,0,-orificeHeight);
137         w3.createOpenPrism(w3Points); // Orifice
138         wallHandler.copyAndAddObject(w3);
139
140
141         // Deletion boundary at bottom to remove particles that left the hopper
142         DeletionBoundary db0;
143         db0.set(Vec3D(0,0,-1),getZMin()-orificeHeight-5*radius);// Definitely below
            minimum and not interfering with particles partly inside the hopper
144         boundaryHandler.copyAndAddObject(db0);
145
146
147         // Particle setup
148         double fillHeight = 8.0*2.0*orificeRadius;
149         double volume;
150         if (fillHeight<=topConeHeight)
151         {
152             volume =
                1.0/3.0*constants::pi*pow(fillHeight/topConeHeight*outerCylinderRadius,2.0
                )*fillHeight; // Volume cone
153         }
154         else
155         {
156             volume =
                1.0/3.0*constants::pi*pow(outerCylinderRadius,2.0)*topConeHeight +
                constants::pi*pow(outerCylinderRadius,2.0)*(fillHeight-topConeHeight);
                // Volume cone + outer (top) cylinder
157         }
158
159         int numPar = fillFraction*volume/(4./3*constants::pi*pow(radius,3.0)); //
            Number of particles to be inserted
160         int numParInserted = 0; // Number of particle inserted
161         Vec3D pos; // Position particle
162         double r, theta, z; // For particle position
163         while( numParInserted < numPar )
164         {
165             double radiusPar =
                random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
166             BaseParticle p0;
167             p0.setSpecies(speciesPar);
168             p0.setRadius(radiusPar);
169
170             int failCounter = 0;
171             do
172             {
173                 r = random.getRandomNumber(0,outerCylinderRadius-radiusPar);
174                 theta = random.getRandomNumber(0,2.0*constants::pi);
175                 z = random.getRandomNumber(getZMin()+topConeHeight,1.5*getZMax());
176
177                 pos.X = center.X+r*cos(theta);
178                 pos.Y = center.Y+r*sin(theta);
179                 pos.Z = z;
180                 p0.setPosition(pos);
181                 p0.setVelocity(Vec3D(0,0,0));
182
183                 failCounter++;
184                 if (failCounter==1000)
185                     break;
186             }
187             while (!checkParticleForInteraction(p0));
188
189             particleHandler.copyAndAddObject(p0);
190
191             numParInserted++;
192         }
```

```
193
194         std::cout << "Finished creating particles" << std::endl;
195         std::cout << "Number of particles inserted: " << numParInserted << std::endl;
196         std::cout << "Particles settling down" << std::endl;
197         step = 2; // Allow next step to be executed
198         checkTime = getTime() + .1; // 0.1 Time to check the total kinetic energy
199     }
200
201     void actionsBeforeTimeStep() override
202     {
203         // Open the gate once the kinetic energy is low enough to consider the
            particles settled (value found by trial and error)
204         if (step==2) // To stop checking the kinetic energy once the particles are
            settled
205         {
206             if (getTime() > checkTime) // Only check at certain times
207             {
208                 std::cout << "Current KE: " << getKineticEnergy() << std::endl;
209                 if (getKineticEnergy() < 0.0001) // For 0.0001 the particles are
                    settled quite well
210                 {
211                     step = 3; // Allow next step to be executed
212                     std::cout << "Particles settled" << std::endl;
213                     wallHandler.removeObject(0); // Remove bottom wall
214                     std::cout << "Gate open" << std::endl;
215                 }
216                 else
217                 {
218                     checkTime = getTime() + .1;
219                 }
220             }
221         }
222     }
223
224
225     bool continueSolve() const override
226     {
227         // CheckTime+1.0 just to be sure not to stop solving while the particles are
            settling
228         // Comparing energies is a useful criteria to determine arresting flow
            (according to website mercuryDPM)
229         if (getTime() > checkTime+1.0 && getKineticEnergy() < 1e-5*getElasticEnergy())
230         {
231             std::cout << "No more flow" << std::endl;
232             return false;
233         }
234         else
235         {
236             return true;
237         }
238     }
239
240     void setFillFraction (double ff)
241     {
242         fillFraction = ff;
243     }
244
245     void setRadiusVariation (double rv)
246     {
247         radiusVariation = rv;
248     }
249
250     void setDensity (double r)
251     {
252         rho = r;
253     }
254
255     void setRadius (double r)
256     {
257         radius = r;
258     }
259
260     void setCOR (double wallCOR, double parCOR)
```

```
261         {
262             CORWall = wallCOR;
263             CORPar = parCOR;
264         }
265 
266         void setCollisionTime (double coltime)
267         {
268             tc = coltime;
269         }
270 
271         void setStiffness (double k)
272         {
273             stiffness = k;
274         }
275 
276         void setFrictionCoeff (double pwf, double ppf, double rfc)
277         {
278             particleWallFriction = pwf;
279             particleParticleFriction = ppf;
280             rollFricCoeff = rfc;
281         }
282 
283         void setCylinderRadius (double ocr, double icr)
284         {
285             outerCylinderRadius = ocr;
286             innerCylinderRadius = icr;
287         }
288 
289         double getOuterCylinderRadius()
290         {
291             return outerCylinderRadius;
292         }
293 
294         void setConeHeight (double bch, double tch)
295         {
296             bottomConeHeight = bch;
297             topConeHeight = tch;
298         }
299 
300         void setOrificeDimensions (double odr, double odh)
301         {
302             orificeRadius = odr;
303             orificeHeight = odh;
304         }
305 
306     private:
307 
308         double fillFraction;
309         double rho, radius, radiusVariation, mass;
310 
311         double CORWall, CORPar, tc, stiffness;
312 
313         double particleWallFriction, particleParticleFriction, rollFricCoeff;
314 
315         double checkTime;
316         int step;
317 
318         double outerCylinderRadius, innerCylinderRadius;
319         double bottomConeHeight, topConeHeight;
320         double orificeRadius, orificeHeight;
321     };
322 
323     int main(int argc, char** argv)
324     {
325         // Problem setup
326         Bridging problem;
327 
328         problem.setFillFraction(0.5);
329         problem.setRadiusVariation(0.0005); // 0.5mm added to or subtracted from radius
330         problem.setDensity(900);
331         problem.setRadius(0.003);
332         problem.setCOR(0.25,0.25); // Wall, particle
333         problem.setCylinderRadius(0.375,0.0525); // Outer, inner
```

```
334         problem.setConeHeight(0.08,0.58); // Bottom, top
335         problem.setOrificeDimensions(0.055/2,0.006); // Radius hole, height
336         problem.setFrictionCoeff(0.60,0.25,0.30); // SET sliding particle-wall, sliding
            particle-particle, rolling friction coefficient
337         problem.setStiffness(40000); // SET 40000
338
339         problem.setName("Bridging"); // SET: Bridging_pwf060_ppf020_rfc040_odia0030
340         problem.setSystemDimensions(3);
341         problem.setGravity(Vec3D(0.0,0.0,-9.81));
342         problem.setXMin(0.0);
343         problem.setYMin(0.0);
344         problem.setZMin(0.0);
345         problem.setXMax(2.0*problem.getOuterCylinderRadius());
346         problem.setYMax(2.0*problem.getOuterCylinderRadius());
347         problem.setZMax(1.16);
348         problem.setTimeMax(500.0); // SET
349         problem.setTimeStep(1.0e-5); // SET 1.0e-5
350
351         problem.setSaveCount(1000); // SET
352         problem.dataFile.setFileType(FileType::ONE_FILE);
353         problem.restartFile.setFileType(FileType::ONE_FILE);
354         problem.fStatFile.setFileType(FileType::NO_FILE);
355         problem.eneFile.setFileType(FileType::NO_FILE);
356
357         problem.setXBallsAdditionalArguments(" -solidf -v0 -s 3");
358
359         problem.solve(argc, argv);
360
361         return 0;
362     }
363
```

Appendix C

Waste Gas Cleaning Plant

C.1 Drivers code of the simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 In progress..
28 Divider at top of waste gas cleaning plant, which spreads the particles out.
29 First fill as must non-moving areas as possible with particles (rings, boxsection,
   I-types).
30 Then insert bunch of particles at certain time intervals at different places above
   rings.
31 Goal: find out what influence the position of the particle drop has on there final
32 position after going through the divider.
33 */
34
35 #include "Mercury3D.h"
36 #include "Species/LinearViscoelasticFrictionSpecies.h"
37 #include "Walls/TriangleWall.h"
38 #include "Particles/BaseParticle.h"
39 #include "Boundaries/CubeInsertionBoundary.h"
40
41 class WGCP : public Mercury3D
42 {
43 public:
44
45     void setupInitialConditions() override
46     {
47         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
48
49         speciesWall =
           speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
50         speciesPar =
           speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
51
52         // Wall species
53         speciesWall->setDensity(rho);
54         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mass);
55
56         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
57         speciesWall->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
58         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
59
60         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
61         speciesWall->setRollingFrictionCoefficient(rollingFrictionCoefficient);
62         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
63
64         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
65         speciesWall->setTorsionFrictionCoefficient(0.1);
66         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
67
68         // Particle species
69         speciesPar->setDensity(rho);
```

```
70          speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORPar,mass);
71
72          speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
73          speciesPar->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
74          speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
75
76          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
77          speciesPar->setRollingFrictionCoefficient(rollingFrictionCoefficient);
78          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
79
80          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
81          speciesPar->setTorsionFrictionCoefficient(0.01);
82          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
83
84          // Wall and Particle species
85          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
86
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mas
            s);
87
88
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
89          speciesWallAndPar->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
90
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
91
92
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
93          speciesWallAndPar->setRollingFrictionCoefficient(rollingFrictionCoefficient);
94
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
95
96
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
97          speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
98
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
99
100
101         // Wall setup
102         // wallHandler.readTriangleWall(filename, species, scaleFactor,
            centerOfRotation, velocity, angularVelocity)
103
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_boxsection.stl.stl",speciesW
            all,1,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
104
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_I-type-1.stl.stl",speciesWal
            l,1,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
105
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_I-type-2.stl.stl",speciesWal
            l,1,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
106
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_rings.stl.stl",speciesWall,1
            ,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
107
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_adsorber.stl.stl",speciesWal
            l,1,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
108
            wallHandler.readTriangleWall("wgcp_ac_inlet_3.dem_floor.stl.stl",speciesWall,1
            ,Vec3D(0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
109
110
111         // Particle setup general
112         double volumePar = (4./3.*constants::pi*pow(radius,3.0)); // Volume of one
            particle
113         Vec3D pos; // Position particle
```

```
114         double r, theta, y; // For position particle in cylindrical coordinates
115         double radiusPar; // For temporary random particle radius
116         BaseParticle p0;
117         p0.setSpecies(speciesPar);
118         p0.setVelocity(Vec3D(0,0,0));
119
120
121         // Particle setup for I-type
122         /*
123         Both I-types are identical and have a symmetry line at z=0.
124         One section at the top on the small leg at in positive x- and z-coordinates
            is "filled" with particles.
125         While this is done, each particle position is mirrored to fill the I-type
            halves and translated to fill
126         the consecutive sections.
127         This is first done for all the rectangled sections, after which the angled
            sections are filled in a similar way.
128         */
129         if (createParItype)
130         {
131             double angle = -40./180.*constants::pi; // Angle of I-types from degree
                to radians
132             Vec3D pos2; // To temporarely save position
133
134             // Particle setup one small section
135             double zsideItypeTopLeg = zsideBoxsection-widthItype; // Top leg
                z-position from zsideItypeTopLeg to zsideBoxsection
136             int numParItype = widthItype*spaceItype*heightItype/volumePar*0.5; //
                0.5 fill fraction
137             int numParItypeInserted = 0, numParItypeInteraction = 0;
138             while (numParItypeInserted < numParItype)
139             {
140                 int failCounter = 0;
141                 do
142                 {
143                     // Position withing one section volume at origin
144                     pos.X = random.getRandomNumber(radiusPar,spaceItype-radiusPar);
145                     pos.Y = random.getRandomNumber(radiusPar,2*heightItype); //
                        factor 2 to prevent interaction
146                     pos.Z = random.getRandomNumber(radiusPar,widthItype-radiusPar);
147                     // Rotate positions in xy-plane
148                     pos2.X = cos(angle)*pos.X-sin(angle)*pos.Y;
149                     pos2.Y = sin(angle)*pos.X+cos(angle)*pos.Y;
150                     // Translate positions to top of small leg
151                     pos.X = pos2.X + xsideBoxsection;
152                     pos.Y = pos2.Y + bottomBoxsection;
153                     pos.Z += zsideItypeTopLeg;
154                     // Set particle position
155                     p0.setPosition(pos);
156
157                     failCounter++;
158                     if (failCounter==1000)
159                     {
160                         std::cout << "Added particle with interaction" << std::endl;
161                         numParItypeInteraction++;
162                         break;
163                     }
164                 }
165                 while (!checkParticleForInteraction(p0));
166
167                 // Increase counter for number of particles in one section
168                 numParItypeInserted++;
169
170                 // Save original position
171                 pos2 = pos;
172                 // For each half I-type fill consecutive sections
173                 for (int j=0; j<4; j++)
174                 {
175                     // 4 combinations of [x,z]: [1,1], [-1,1], [1,-1], [-1,-1]
176                     // x-position alternating positive and negative
177                     pos.X = pow(-1,j)*pos2.X;
178                     // y-position doesn't change
179                     pos.Y = pos2.Y;
```

```
180                     // z-position positive for first two iterations, then negative
181                     if (j<2)
182                     {
183                         pos.Z = pos2.Z;
184                     }
185                     else
186                     {
187                         pos.Z = -pos2.Z;
188                     }
189
190                     // Set positions of first section
191                     p0.setPosition(pos);
192                     // Change radius to random value, so only positions are similar
193                     radiusPar =
                        random.getRandomNumber(radius-radiusVariation,radius+radiusVariati
                        on);
194                     p0.setRadius(radiusPar);
195                     // Place particle in first section
196                     particleHandler.copyAndAddObject(p0);
197
198
199                     // 5 more sections to fill for small leg, in total 6 are filled
200                     for (int i=0; i<5; i++)
201                     {
202                         // Go down following small leg, z-position doesn't change
203                         pos.X += pow(-1,j)*spaceItype*cos(angle);
204                         pos.Y += spaceItype*sin(angle);
205                         p0.setPosition(pos);
206
207                         // Change radius to random value, so only positions are
                            similar
208                         radiusPar =
                            random.getRandomNumber(radius-radiusVariation,radius+radiusVar
                            iation);
209                         p0.setRadius(radiusPar);
210
211                         particleHandler.copyAndAddObject(p0);
212                     }
213
214                     // Next start at top of main leg
215                     pos.X += pow(-1,j)*spaceItype*cos(angle)*3; // Skip 3 sections
216                     pos.Y += spaceItype*sin(angle)*3; // Skip 3 sections
217                     if (j<2)
218                     {
219                         pos.Z -= zsideItypeTopLeg;
220                     }
221                     else
222                     {
223                         pos.Z += zsideItypeTopLeg;
224                     }
225
226                     // 9 sections to fill in total for main leg
227                     for (int i=0; i<9; i++)
228                     {
229                         // Go down following main leg, z-position doesn't change
230                         pos.X += pow(-1,j)*spaceItype*cos(angle);
231                         pos.Y += spaceItype*sin(angle);
232                         p0.setPosition(pos);
233
234                         // Change radius to random value, so only positions are
                            similar
235                         radiusPar =
                            random.getRandomNumber(radius-radiusVariation,radius+radiusVar
                            iation);
236                         p0.setRadius(radiusPar);
237
238                         particleHandler.copyAndAddObject(p0);
239                     }
240                 }
241             }
242
243             /*
244             Now the angled sections connecting the small and main leg are filled.
```

```
245             Since the 3 sections are all a bit different we take a parrallelogram
                like area, which fits all of them.
246             */
247             double ratioItypeParal = 2.5*spaceItype/zsideItypeTopLeg; // Ratio
                angled line: x/z
248             int numParItypeParal =
                (widthItype+0.015)*spaceItype*heightItype/volumePar*0.5; // 0.015
                because more parrallelogram shaped, 0.5 fill fraction
249             int numParItypeParalInserted = 0, numParItypeParalInteraction = 0;
250             while (numParItypeParalInserted < numParItypeParal)
251             {
252                 int failCounter = 0;
253                 do
254                 {
255                     // See notebook for simple calculations
256                     pos.X = random.getRandomNumber(radiusPar,spaceItype-radiusPar);
257                     pos.Y = random.getRandomNumber(radiusPar,2*heightItype); //
                    factor 2 to prevent interaction
258                     if (pos.X <= 0.5*spaceItype)
259                     {
260                         pos.Z =
                        random.getRandomNumber(0.03-3./5.*pos.X+radiusPar,widthItype+0
                        .03-radiusPar);
261                     }
262                     else if (pos.X > 0.5*spaceItype)
263                     {
264                         pos.Z =
                        random.getRandomNumber(radiusPar,widthItype+3./5.*(spaceItype-
                        pos.X)-radiusPar);
265                     }
266                     // Rotate positions in xy-plane
267                     pos2.X = cos(angle)*pos.X-sin(angle)*pos.Y;
268                     pos2.Y = sin(angle)*pos.X+cos(angle)*pos.Y;
269                     // Translate positions to lowest section of angled leg
270                     pos.X = pos2.X + xsideBoxsection + spaceItype*cos(angle)*8;
271                     pos.Y = pos2.Y + bottomBoxsection + spaceItype*sin(angle)*8;
272                     // Set particle position
273                     p0.setPosition(pos);
274
275                     failCounter++;
276                     if (failCounter==1000)
277                     {
278                         std::cout << "Added particle with interaction" << std::endl;
279                         numParItypeParalInteraction++;
280                         break;
281                     }
282                 }
283                 while (!checkParticleForInteraction(p0));
284
285                 // Increase counter for number of particles in one section
286                 numParItypeParalInserted++;
287
288                 // Save original position
289                 pos2 = pos;
290                 // For each half I-type fill consecutive sections
291                 for (int j=0; j<4; j++)
292                 {
293                     // 4 combinations of [x,z]: [1,1], [-1,1], [1,-1], [-1,-1]
294                     // x-position alternating positive and negative
295                     pos.X = pow(-1,j)*pos2.X;
296                     // y-position doesn't change
297                     pos.Y = pos2.Y;
298                     // z-position positive for first two iterations, then negative
299                     if (j<2)
300                     {
301                         pos.Z = pos2.Z;
302                     }
303                     else
304                     {
305                         pos.Z = -pos2.Z;
306                     }
307
308                     // Set positions of first section
```

```
309                     p0.setPosition(pos);
310                     // Change radius to random value, so only positions are similar
311                     radiusPar =
                        random.getRandomNumber(radius-radiusVariation,radius+radiusVariati
                        on);
312                     p0.setRadius(radiusPar);
313                     // Place particle in first section
314                     particleHandler.copyAndAddObject(p0);
315
316                     // 2 more sections to fill, 3 in total
317                     for (int i=0; i<2; i++)
318                     {
319                         // Go up following angled leg
320                         pos.X -= pow(-1,j)*spaceItype*cos(angle);
321                         pos.Y -= spaceItype*sin(angle);
322                         if (j<2)
323                         {
324                             pos.Z += 0.06;
325                         }
326                         else
327                         {
328                             pos.Z -= 0.06;
329                         }
330                         p0.setPosition(pos);
331
332                         // Change radius to random value, so only positions are
                            similar
333                         radiusPar =
                            random.getRandomNumber(radius-radiusVariation,radius+radiusVar
                            iation);
334                         p0.setRadius(radiusPar);
335
336                         particleHandler.copyAndAddObject(p0);
337                     }
338                 }
339             }
340             int numParItypeInsertedTotal =
                4*15*numParItypeInserted+4*3*numParItypeParalInserted;
341             int numParItypeInteractionTotal =
                4*15*numParItypeInteraction+4*3*numParItypeParalInteraction;
342             std::cout << "Finished creating particles I-types. \nNumber inserted: "
                << numParItypeInsertedTotal << "\nNumber with interaction: " <<
                numParItypeInteractionTotal << std::endl;
343         }
344
345
346         // Particle setup rings
347         if (createParRings)
348         {
349             // Particle setup outer-middle ring section
350             int numParOutMidRing =
                constants::pi*(pow(radiusRings,2)-pow(radiusMiddleRing,2))*heightMiddleRin
                g/volumePar*0.5; // 0.5 fill fraction
351             int numParOutMidRingInserted = 0, numParOutMidRingInteraction = 0;
352             double ratioOutMidRing =
                (std::abs(topMiddleRing-topRings)+heightMiddleRing)/(radiusRings-radiusMid
                dleRing); // Ratio right angle from middle ring to top
353             while (numParOutMidRingInserted < numParOutMidRing)
354             {
355                 radiusPar =
                    random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
356                 p0.setRadius(radiusPar);
357
358                 int failCounter = 0;
359                 do
360                 {
361                     // See notebook for simple calculations
362                     r =
                        random.getRandomNumber(radiusMiddleRing+radiusPar,radiusRings);
363                     theta = random.getRandomNumber(0,2*constants::pi);
364                     y =
                        random.getRandomNumber(topRings-(radiusRings-r)*ratioOutMidRing+ra
                        diusPar,topRings+3*radiusPar); // 3 times radius to prevent
```

```
                    interaction
365
366                 pos.X = r*cos(theta);
367                 pos.Y = y;
368                 pos.Z = r*sin(theta);
369                 p0.setPosition(pos);
370
371                 failCounter++;
372                 if (failCounter==1000)
373                 {
374                     std::cout << "Added particle with interaction" << std::endl;
375                     numParOutMidRingInteraction++;
376                     break;
377                 }
378             }
379             while (!checkParticleForInteraction(p0));
380
381             particleHandler.copyAndAddObject(p0);
382             numParOutMidRingInserted++;
383         }
384         std::cout << "Finished creating particles outer-middle ring section.
            \nNumber inserted: " << numParOutMidRingInserted << "\nNumber with
            interaction: " << numParOutMidRingInteraction << std::endl;
385
386
387         // Particle setup middle-inner ring section
388         int numParMidInRing =
            constants::pi*(pow(radiusMiddleRing,2)-pow(radiusInnerRing,2))*heightInner
            Ring/volumePar*0.5; // 0.5 fill fraction
389         int numParMidInRingInserted = 0, numParMidInRingInteraction = 0;
390         double ratioMidInRing =
            (std::abs(topInnerRing-topMiddleRing)-heightMiddleRing+heightInnerRing)/(r
            adiusMiddleRing-radiusInnerRing); // Ratio right angle from inner to
            middle ring
391         while (numParMidInRingInserted < numParMidInRing)
392         {
393             radiusPar =
            random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
394             p0.setRadius(radiusPar);
395
396             int failCounter = 0;
397             do
398             {
399                 // See notebook for simple calculations
400                 r =
                    random.getRandomNumber(radiusInnerRing+radiusPar,radiusMiddleRing-
                    radiusPar-0.006); // 6mm width ring
401                 theta = random.getRandomNumber(0,2*constants::pi);
402                 y =
                    random.getRandomNumber(topMiddleRing-heightMiddleRing-(radiusMiddl
                    eRing-r)*ratioMidInRing+radiusPar,topMiddleRing+5*radiusPar); //
                    5 times radius to prevent interaction
403
404                 pos.X = r*cos(theta);
405                 pos.Y = y;
406                 pos.Z = r*sin(theta);
407                 p0.setPosition(pos);
408
409                 failCounter++;
410                 if (failCounter==1000)
411                 {
412                     std::cout << "Added particle with interaction" << std::endl;
413                     numParMidInRingInteraction++;
414                     break;
415                 }
416             }
417             while (!checkParticleForInteraction(p0));
418
419             particleHandler.copyAndAddObject(p0);
420             numParMidInRingInserted++;
421         }
422         std::cout << "Finished creating particles middle-inner ring section.
            \nNumber inserted: " << numParMidInRingInserted << "\nNumber with
```

```
            interaction: " << numParMidInRingInteraction << std::endl;
423         }
424
425
426         // Particle setup for boxsection
427         if (createParBoxsection)
428         {
429             // Particle setup for boxsection bed
430             double heightBoxsectionBed = 0.03;
431             int numParBoxsectionBed =
                2*xsideBoxsection*2*zsideBoxsection*heightBoxsectionBed/volumePar*0.5;
                // 0.5 fill fraction
432             int numParBoxsectionBedInserted = 0, numParBoxsectionBedInteraction = 0;
433             while (numParBoxsectionBedInserted < numParBoxsectionBed)
434             {
435                 radiusPar =
                    random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
436                 p0.setRadius(radiusPar);
437
438                 int failCounter = 0;
439                 do
440                 {
441                     pos.X =
                        random.getRandomNumber(-xsideBoxsection+radiusPar,xsideBoxsection-
                        radiusPar);
442                     pos.Y =
                        random.getRandomNumber(bottomBoxsection+radiusPar,bottomBoxsection
                        +heightBoxsectionBed+7*radiusPar);
443                     pos.Z =
                        random.getRandomNumber(-zsideBoxsection+radiusPar,zsideBoxsection-
                        radiusPar);
444                     p0.setPosition(pos);
445
446                     failCounter++;
447                     if (failCounter==1000)
448                     {
449                         std::cout << "Added particle with interaction" << std::endl;
450                         numParBoxsectionBedInteraction++;
451                         break;
452                     }
453                 }
454                 while (!checkParticleForInteraction(p0));
455
456                 particleHandler.copyAndAddObject(p0);
457                 numParBoxsectionBedInserted++;
458             }
459             std::cout << "Finished creating particles boxsection bed. \nNumber
                inserted: " << numParBoxsectionBedInserted << "\nNumber with
                interaction: " << numParBoxsectionBedInteraction << std::endl;
460
461
462             // Particle setup for boxsection cone
463             double radiusBoxsectionCone = zsideBoxsection;
464             int numParBoxsectionCone =
                1./3.*constants::pi*pow(radiusBoxsectionCone,2)*std::abs(topBoxsection-bot
                tomBoxsection)/volumePar*0.5; // 0.5 fill fraction
465             int numParBoxsectionConeInserted = 0, numParBoxsectionConeInteraction = 0;
466             while (numParBoxsectionConeInserted < numParBoxsectionCone)
467             {
468                 radiusPar =
                    random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
469                 p0.setRadius(radiusPar);
470
471                 int failCounter = 0;
472                 do
473                 {
474                     // See notebook for simple calculations
475                     r = random.getRandomNumber(0,radiusBoxsectionCone-radiusPar);
476                     theta = random.getRandomNumber(0,2*constants::pi);
477                     y =
                        random.getRandomNumber(bottomBoxsection+heightBoxsectionBed+7*radi
                        usPar,bottomBoxsection+heightBoxsectionBed+7*radiusPar+(radiusBoxs
                        ectionCone-r)*tan(40./180.*constants::pi)+10*radiusPar);
```

```
478
479                     pos.X = r*cos(theta);
480                     pos.Y = y;
481                     pos.Z = r*sin(theta);
482                     p0.setPosition(pos);
483
484                     failCounter++;
485                     if (failCounter==1000)
486                     {
487                         std::cout << "Added particle with interaction" << std::endl;
488                         numParBoxsectionConeInteraction++;
489                         break;
490                     }
491                 }
492                 while (!checkParticleForInteraction(p0));
493
494                 particleHandler.copyAndAddObject(p0);
495                 numParBoxsectionConeInserted++;
496             }
497             std::cout << "Finished creating particles boxsection cone. \nNumber
                inserted: " << numParBoxsectionConeInserted << "\nNumber with
                interaction: " << numParBoxsectionConeInteraction << std::endl;
498
499
500             // Particle setup for boxsection strips
501             double widthBoxsectionStrip = 0.1, heightBoxsectionStrip = 0.05;
502             int numParBoxsectionStrip =
                2*zsideBoxsection*widthBoxsectionStrip*heightBoxsectionStrip/volumePar*0.5
                ; // 0.5 fill fraction
503             int numParBoxsectionStripInserted = 0, numParBoxsectionStripInteraction
                = 0;
504             while (numParBoxsectionStripInserted < numParBoxsectionStrip)
505             {
506                 radiusPar =
                    random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
507                 p0.setRadius(radiusPar);
508
509                 int failCounter = 0;
510                 do
511                 {
512                     pos.X =
                        random.getRandomNumber(xsideBoxsection-widthBoxsectionStrip,xsideB
                        oxsection-radiusPar);
513                     pos.Y =
                        random.getRandomNumber(bottomBoxsection+heightBoxsectionBed+7*radi
                        usPar,topBoxsection);
514                     pos.Z =
                        random.getRandomNumber(-zsideBoxsection+radiusPar,zsideBoxsection-
                        radiusPar);
515                     p0.setPosition(pos);
516
517                     failCounter++;
518                     if (failCounter==1000)
519                     {
520                         std::cout << "Added particle with interaction" << std::endl;
521                         numParBoxsectionStripInteraction++;
522                         break;
523                     }
524                 }
525                 while (!checkParticleForInteraction(p0));
526
527                 particleHandler.copyAndAddObject(p0);
528                 numParBoxsectionStripInserted++;
529
530                 // Copy to other side box
531                 pos.X = -pos.X;
532                 p0.setPosition(pos);
533                 particleHandler.copyAndAddObject(p0);
534             }
535             numParBoxsectionStripInserted *= 2;
536             numParBoxsectionStripInteraction *= 2;
537             std::cout << "Finished creating particles boxsection strip. \nNumber
                inserted: " << numParBoxsectionStripInserted << "\nNumber with
```

```
            interaction: " << numParBoxsectionStripInteraction << std::endl;
538         }
539
540
541         // Particle setup flow
542         if (createParFlow)
543         {
544             int numParFlow =
                constants::pi*pow(radiusFlow,2)*heightFlow/volumePar*0.25; // 0.25 fill
                fraction, also to prevent interaction
545             int numParFlowInserted = 0, numParFlowInteraction = 0;
546             while( numParFlowInserted < numParFlow )
547             {
548                 radiusPar =
                    random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
549                 p0.setRadius(radiusPar);
550
551                 int failCounter = 0;
552                 do
553                 {
554                     r = random.getRandomNumber(0,radiusFlow-radiusPar);
555                     theta = random.getRandomNumber(0,2*constants::pi);
556                     y = random.getRandomNumber(0,heightFlow);
557
558                     pos.X =
                        r*cos(theta)+cos(anglePosFlow/180*constants::pi)*radiusPosFlow;
559                     pos.Y = y-0.15; // LITTLE LOWER
                        !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
560                     pos.Z =
                        r*sin(theta)+sin(anglePosFlow/180*constants::pi)*radiusPosFlow;
561                     p0.setPosition(pos);
562
563                     failCounter++;
564                     if (failCounter==1000)
565                     {
566                         std::cout << "Added particle with interaction" << std::endl;
567                         numParFlowInteraction++;
568                         break;
569                     }
570                 }
571                 while (!checkParticleForInteraction(p0));
572
573                 particleHandler.copyAndAddObject(p0);
574                 numParFlowInserted++;
575             }
576             std::cout << "Finished creating particles for Flow. \nNumber inserted: "
                << numParFlowInserted << "\nNumber with interaction: " <<
                numParFlowInteraction << std::endl;
577         }
578
579
580         std::cout << "Finished creating particles\n";
581         std::cout << "Total number of particles inserted: " <<
            particleHandler.getNumberOfRealObjects() << std::endl;
582
583     }
584
585     void actionsAfterTimeStep() override
586     {
587         // If particles for flow have to be created
588         if (createParFlow)
589         {
590             // If number of intervals less than total number of intervals
591             if (numIntFlowCount<numIntFlow)
592             {
593                 // Calculate check for new interval
594                 int checkIntFlow = getTime()/timeIntFlow/numIntFlowCount;
595                 // If check is true insert particles
596                 if (checkIntFlow==1)
597                 {
598                     // Particle set up general
599                     double volumePar = (4./3.*constants::pi*pow(radius,3.0)); //
                        Volume of one particle
```

```
600                     Vec3D pos; // Position particle
601                     double r, theta, y; // For position particle in cylindrical
                        coordinates
602                     double radiusPar; // For temporarely random particle radius
603                     BaseParticle p0;
604                     p0.setSpecies(speciesPar);
605                     p0.setVelocity(Vec3D(0,0,0));
606
607                     // Particle setup flow
608                     int numParFlow =
                        constants::pi*pow(radiusFlow,2)*heightFlow/volumePar*0.25; //
                        0.25 fill fraction, also to prevent interaction
609                     int numParFlowInserted = 0, numParFlowInteraction = 0;
610                     while( numParFlowInserted < numParFlow )
611                     {
612                         radiusPar =
                            random.getRandomNumber(radius-radiusVariation,radius+radiusVar
                            iation);
613                         p0.setRadius(radiusPar);
614
615                         int failCounter = 0;
616                         do
617                         {
618                             r = random.getRandomNumber(0,radiusFlow-radiusPar);
619                             theta = random.getRandomNumber(0,2*constants::pi);
620                             y = random.getRandomNumber(0,heightFlow);
621
622                             pos.X =
                                r*cos(theta)+cos(anglePosFlow/180*constants::pi)*radiusPos
                                Flow;
623                             pos.Y = y-0.15; // LITTLE LOWER
                                !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
                                !!!!!
624                             pos.Z =
                                r*sin(theta)+sin(anglePosFlow/180*constants::pi)*radiusPos
                                Flow;
625                             p0.setPosition(pos);
626
627                             failCounter++;
628                             if (failCounter==1000)
629                             {
630                                 std::cout << "Added particle with interaction" <<
                                    std::endl;
631                                 numParFlowInteraction++;
632                                 break;
633                             }
634                         }
635                         while (!checkParticleForInteraction(p0));
636
637                         particleHandler.copyAndAddObject(p0);
638                         numParFlowInserted++;
639                     }
640                     std::cout << "Inserted new set of particles. Total of " <<
                        numParFlowInserted << " particles with " <<
                        numParFlowInteraction << " interactions." << std::endl;
641
642                     // Increase counter number of intervals
643                     numIntFlowCount++;
644                 }
645             }
646         }
647     }
648
649     bool continueSolve() const override
650     {
651         // After last interval is started and add 1.0 second to be sure flow started
652         // AND kinetic energy is smaller than 1e-5 times the kinetic energy
653         if ( (getTime() > (((numIntFlow-1)*timeIntFlow)+1.0)) && (getKineticEnergy()
            < 1e-5*getElasticEnergy()) )
654         {
655             std::cout << "No more flow" << std::endl;
656             return false;
657         }
```

```
658         else
659         {
660             return true;
661         }
662     }
663
664     void setDensity (double d)
665     {
666         rho = d;
667     }
668
669     void setRadius (double r, double rv)
670     {
671         radius = r;
672         radiusVariation = rv;
673     }
674
675     void setCOR (double wallCOR, double parCOR)
676     {
677         CORWall = wallCOR;
678         CORPar = parCOR;
679     }
680
681     void setStiffness (double k)
682     {
683         stiffness = k;
684     }
685
686     void setFrictionCoefficient (double sfc, double rfc)
687     {
688         slidingFrictionCoefficient = sfc;
689         rollingFrictionCoefficient = rfc;
690     }
691
692     void setBoxsection (double bottom, double top, double xside, double zside)
693     {
694         bottomBoxsection = bottom;
695         topBoxsection = top;
696         xsideBoxsection = xside;
697         zsideBoxsection = zside;
698     }
699
700     void setRings (double top, double rad, double topMiddle, double radMiddle,
    double topInner, double radInner)
701     {
702         topRings = top;
703         radiusRings = rad;
704         topMiddleRing = topMiddle;
705         radiusMiddleRing = radMiddle;
706         topInnerRing = topInner;
707         radiusInnerRing = radInner;
708     }
709
710     void setHeightRings (double hmr, double hir)
711     {
712         heightMiddleRing = hmr;
713         heightInnerRing = hir;
714     }
715
716     void setItype (double space, double height, double width)
717     {
718         spaceItype = space;
719         heightItype = height;
720         widthItype = width;
721     }
722
723     void setAdsorber (double rad)
724     {
725         radiusAdsorber = rad;
726     }
727
728     void setCreateParticles (bool Itype, bool rings, bool boxsection, bool flow)
729     {
```

```
730             createParItype = Itype;
731             createParRings = rings;
732             createParBoxsection = boxsection;
733             createParFlow = flow;
734         }
735
736         void setFlow (double rad, double height, double anglePos, double radPos)
737         {
738             radiusFlow = rad;
739             heightFlow = height;
740             anglePosFlow = anglePos;
741             radiusPosFlow = radPos;
742         }
743
744         void setFlowInterval (int numInt, double timeInt)
745         {
746             numIntFlow = numInt;
747             timeIntFlow = timeInt;
748         }
749
750     private:
751         double rho, radius, radiusVariation, mass;
752         double CORWall, CORPar, stiffness;
753         double slidingFrictionCoefficient, rollingFrictionCoefficient;
754
755         double bottomBoxsection, topBoxsection, xsideBoxsection, zsideBoxsection;
756         double topRings, radiusRings, topMiddleRing, radiusMiddleRing, topInnerRing,
            radiusInnerRing;
757         double heightMiddleRing, heightInnerRing;
758         double spaceItype, heightItype, widthItype;
759         double radiusAdsorber;
760
761         bool createParItype, createParRings, createParBoxsection, createParFlow;
762
763         double radiusFlow, heightFlow, anglePosFlow, radiusPosFlow, timeIntFlow;
764         int numIntFlow, numIntFlowCount=1;
765
766         LinearViscoelasticFrictionSpecies* speciesWall;
767         LinearViscoelasticFrictionSpecies* speciesPar;
768         LinearViscoelasticFrictionSpecies* speciesWallAndPar;
769     };
770
771     int main(int argc, char** argv)
772     {
773         WGCP problem;
774
775         problem.setSystemDimensions(3);
776         problem.setGravity(Vec3D(0,-9.81,0));
777         // Domain boundaries tightly around distributor etc. Comments is around whole
            system
778         problem.setXMin(-2); // -4.037
779         problem.setYMin(-2.4630); // -5
780         problem.setZMin(-0.325); // -2.4
781         problem.setXMax(2); // 4.037
782         problem.setYMax(0); // 0
783         problem.setZMax(0.325); // 2.4
784         problem.dataFile.setFileType(FileType::ONE_FILE);
785         problem.restartFile.setFileType(FileType::ONE_FILE);
786         problem.fStatFile.setFileType(FileType::ONE_FILE);
787         problem.eneFile.setFileType(FileType::ONE_FILE);
788         problem.setWallsWriteVTK(FileType::ONE_FILE);
789         problem.setParticlesWriteVTK(1);
790
791         problem.setCOR(0.25,0.25); // Particle-wall, particle-particle
792         problem.setStiffness(40000);
793         problem.setFrictionCoefficient(0.25,0.40); // Sliding, rolling
794         problem.setBoxsection(-1.306,-1.046,0.331,0.325); // Bottom, top, xside, zside
795         problem.setRings(-0.5,0.275,-0.538,0.2,-0.6,0.10955); // Top outer, radius
            outer, top middle, radius middle, top inner, radius inner
796         problem.setHeightRings(0.03,0.05); // Middle, inner
797         problem.setItype(0.1,0.05,0.175); // Space, height, width of one section
798         problem.setAdsorber(0.2); // Radius
799
```

```
800
801         // Add any configuration which is modified often below...
802         problem.setName("WGCP3"); // Make sure to add an (unique) name here
803         problem.setTimeMax(100);
804         problem.setTimeStep(1e-5);
805         problem.setSaveCount(2500); // 40 fps with timestep 1e-5
806
807         problem.setDensity(900);
808         problem.setRadius(0.007,0.0005); // Radius, radius variation
809         problem.setCreateParticles(1,1,1,1); // Itype, rings, boxsection, flow
810         problem.setFlow(0.1,0.15,90,0.1); // Radius, height, angle position (0:x, 45:xz,
            90:z), radius position (0:center)
811         // Flow bottom is placed at top adsorber minus 0.15 in the code.
812         problem.setFlowInterval(100,0.75); // Number of flows (default: 1), time between
            flows
813
814         // Set the number of domains for parallel decomposition
815         problem.setNumberOfDomains({3,1,1});
816         // For whatever reason a species have to be set in main() for parallel
            decomposition
817         LinearViscoelasticFrictionSpecies* species =
            problem.speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
818
819         // Now, start the simulation
820         problem.solve(argc, argv);
821         return 0;
822     }
823
```

C.2 Python code for quick visualisation of cluster data in Paraview

```
#script to visualise the output of data2pvd of MercuryDPM in paraview.
#usage: change the path below to your own path, open paraview
#Tools->Python Shell->Run Script->VisualisationScript.py
#or run paraview --script=VisualisationScript.py

from paraview.simple import *
import os
import glob

# Set path and name of simulation
os.chdir('c:/Apps/MercuryDPM/MercuryBuild/Drivers/USER/BlueScope/WasteGasCleaningPlant
/cluster')
name = 'WGCPc3_XZside'

#Load data in any order
Data = glob.glob('./' + name + 'Processor_*_Particle_*.vtu')

# Find the maximum time step and maximum number of processors
maxTime = 0
maxProc = 0
for fileName in Data:
    tokens1 = fileName.split('.')
    tokens2 = tokens1[1].split('_')
    if int(tokens2[-1]) > maxTime:
        maxTime = int(tokens2[-1])
    if int(tokens2[-3]) > maxProc:
        maxProc = int(tokens2[-3])
print 'maxTimeStep =',str(maxTime)

# For each processor
for i in range(maxProc+1):
    # Create correct order of time steps
    DataSorted = []
    for x in range(0,maxTime+1):
        DataSorted.append('./' + name + 'Processor_' + str(i) + '_Particle_' +
        str(x) + '.vtu')

    # Load the data and visualise it in Paraview
    particles = XMLUnstructuredGridReader(FileName=DataSorted)
    glyphP = Glyph(particles)
    glyphP.GlyphType = 'Sphere'
    glyphP.Scalars = 'Radius'
    glyphP.Vectors = 'None'
    glyphP.ScaleMode = 'scalar'
    glyphP.ScaleFactor = 2
    glyphP.GlyphMode = 'All Points'
    glyphP.Orient = 0
    Show(glyphP)
    #glyphP.Coloring = 'Velocity'

# Load wall and visualise it in Paraview
walls = XMLUnstructuredGridReader(FileName=glob.glob('./' + name + 'Wall_0.vtu'))
Show(walls)
```

Appendix D

Blast Furnace

D.1 Drivers code of the simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 In progress..
28 Particles are added to a blast furnace, which is kept completely filled by continously
29 adding particles to the top. At the bottom at both sides the particles are discharged.
30 Goal: compare difference between using solid walls and periodic boundaries in
   z-direction.
31 */
32
33 #include "Mercury3D.h"
34 #include "Species/LinearViscoelasticFrictionSpecies.h"
35 #include "Walls/TriangleWall.h"
36 #include "Walls/InfiniteWall.h"
37 #include "Boundaries/PeriodicBoundary.h"
38 #include "Particles/BaseParticle.h"
39 #include "Boundaries/CubeInsertionBoundary.h"
40 #include "Boundaries/DeletionBoundary.h"
41
42 class BF : public Mercury3D
43 {
44 public:
45
46     void setupInitialConditions() override
47     {
48         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
49
50         auto speciesWall =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
51         auto speciesPar =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
52
53         // Wall species
54         speciesWall->setDensity(rho);
55         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mass);
56
57         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
58         speciesWall->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
59         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
60
61         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
62         speciesWall->setRollingFrictionCoefficient(rollingFrictionCoefficient);
63         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
64
65         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
66         speciesWall->setTorsionFrictionCoefficient(0.1);
67         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
68
69         // Particle species
70         speciesPar->setDensity(rho);
```

```
71          speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORPar,mass);
72
73          speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
74          speciesPar->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
75          speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
76
77          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
78          speciesPar->setRollingFrictionCoefficient(rollingFrictionCoefficient);
79          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
80
81          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
82          speciesPar->setTorsionFrictionCoefficient(0.01);
83          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
84
85          // Wall and Particle species
86          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
87
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORWall,mas
            s);
88
89
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
90          speciesWallAndPar->setSlidingFrictionCoefficient(slidingFrictionCoefficient);
91
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
92
93
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
94          speciesWallAndPar->setRollingFrictionCoefficient(rollingFrictionCoefficient);
95
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
96
97
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
98          speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
99
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
100
101
102         // Wall setup
103         // wallHandler.readTriangleWall(filename, species, scaleFactor,
            centerOfRotation, velocity, angularVelocity)
104
            wallHandler.readTriangleWall("furnace.dem_furnace.stl.stl",speciesWall,1,Vec3D
            (0,0,0),Vec3D(0,0,0),Vec3D(0,0,0));
105
106         if (useInfWalls)
107         {
108             // Infinite walls in z-direction
109             InfiniteWall w0;
110             w0.setSpecies(speciesWall);
111             w0.set(Vec3D(0,0,-1),Vec3D(0,0,getZMin()));
112             wallHandler.copyAndAddObject(w0);
113             w0.set(Vec3D(0,0,1),Vec3D(0,0,getZMax()));
114             wallHandler.copyAndAddObject(w0);
115         }
116         else
117         {
118             // Periodic boundaries in z-direction
119             PeriodicBoundary b0;
120             b0.set(Vec3D(0,0,1),getZMin(),getZMax());
121             boundaryHandler.copyAndAddObject(b0);
122         }
123
124
125         // Particle setup
```

```
126             BaseParticle p0;
127             p0.setSpecies(speciesPar);
128             p0.setVelocity(Vec3D(0,0,0));
129             p0.setRadius(radius);
130             double volumePar = 4./3.*constants::pi*pow(radius,3);
131
132             // Bottom half
133             int numParBottom =
                2*radiusBottomBF*0.5*(getYMax()-getYMin())*(getZMax()-getZMin())/volumePar*0.2
                5; // 0.25 fill fraction
134             int numParBottomInteraction =
                particleInsertion(numParBottom,p0,-radiusBottomBF+radius,radiusBottomBF-radius
                ,getYMin()+radius,0.5*getYMax(),getZMin()+radius,getZMax()-radius);
135             std::cout << "Bottom half: inserted " << numParBottom << " particles, " <<
                numParBottomInteraction << " with interaction" << std::endl;
136
137             // Top half
138             int numParTop =
                2*radiusTopBF*0.5*(getYMax()-getYMin())*(getZMax()-getZMin())/volumePar*0.25;
                // 0.25 fill fraction
139             int numParTopInteraction =
                particleInsertion(numParTop,p0,-radiusTopBF+radius,radiusTopBF-radius,0.5*getY
                Min()+radius,getYMax(),getZMin()+radius,getZMax()-radius);
140             std::cout << "Top half: inserted " << numParTop << " particles, " <<
                numParTopInteraction << " with interaction" << std::endl;
141
142             std::cout << "Total number of particles inserted: " << numParBottom +
                numParTop << std::endl;
143
144             // Insertion boundary at top blast furnace
145             CubeInsertionBoundary insertionBoundary;
146
                insertionBoundary.set(p0,1,Vec3D(-radiusTopBF,getYMax()-0.066,getZMin()),Vec3D
                (radiusTopBF,getYMax(),getZMax()),Vec3D(0,0,0),Vec3D(0,0,0),radius-radiusVaria
                tion,radius+radiusVariation);
147             insertionBoundary.setVolumeFlowRate(1);
148             boundaryHandler.copyAndAddObject(insertionBoundary);
149
150             // Deletion boundary at top BF, to prevent spillage
151             DeletionBoundary db;
152             db.set(Vec3D(0,1,0),getYMax());
153             boundaryHandler.copyAndAddObject(db);
154
155         }
156
157
158         void actionsAfterTimeStep() override
159         {
160             particleDischarge();
161         }
162
163
164         double particleInsertion (int numPar, BaseParticle p0, double xmin, double xmax,
                double ymin, double ymax, double zmin, double zmax)
165         {
166             int numParInserted=0, numParInteraction=0;
167             Vec3D pos;
168             while (numParInserted < numPar)
169             {
170                 int failCounter = 0;
171                 do
172                 {
173                     pos.X = random.getRandomNumber(xmin,xmax);
174                     pos.Y = random.getRandomNumber(ymin,ymax);
175                     pos.Z = random.getRandomNumber(zmin,zmax);
176                     p0.setPosition(pos);
177
178                     failCounter++;
179                     if (failCounter==1000)
180                     {
181                         std::cout << "Added particle with interaction" << std::endl;
182                         numParInteraction++;
183                         break;
```

```
184                 }
185             }
186             while (!checkParticleForInteraction(p0));
187 
188             particleHandler.copyAndAddObject(p0);
189             numParInserted++;
190         }
191         return numParInteraction;
192     }
193 
194 
195     void particleDischarge ()
196     {
197         /*
198         The particles are checked if they touch the bottom at the discharge ports.
            This is done
199         starting from the lowest particle id, since these are likely to be at the
            bottom.
200         When the particles are inserted completely random the order of their ids
            will be arbitrary
201         and a particle touching the bottom can be removed immediately. When the
            order of ids is not
202         arbitrary, for example when inserting sets with different kind of particles
            after each other,
203         there would be a bias towards the kind of particles inserted the latest. In
            both cases the
204         particle ids are first saved, but in case of random insertion the loop is
            stopped once there
205         are enough particles touching the bottom, after which they are all removed.
            In case of non-
206         random insertion, however, the particles to remove are picked at random first.
207         A simple if-statement switches between the two cases.
208         Keep in mind that removing particles needs to be done from the highest
            particle id downwards,
209         to not affect the remaining particles id which still have to be checked or
            removed.
210         */
211         // Mass to be removed per timestep
212         massToBeRemoved += flowRate*getTimeStep();
213         // Number of particles in the system at current time
214         int numPar = particleHandler.getNumberOfObjects();
215         // Average mass
216         double massParAverage = particleHandler.getMass()/numPar;
217         // Number of particles to be removed
218         int numParToBeRemoved = massToBeRemoved/massParAverage;
219         // Declare Vec3D for position particles
220         Vec3D pos;
221         // Declare radius and mass particle
222         double radiusPar, massPar;
223         // Declare vector to save particle id's
224         std::vector<int> idParInRegion;
225         // Declare counter for number of particles in region touching the bottom
226         int numParInRegion = 0;
227 
228         // If particles have to be removed
229         if (numParToBeRemoved > 0)
230         {
231             // For every particle, starting from lowest id since these are likely
                closest to the bottom
232             for (int i=0; i<numPar; i++)
233             {
234                 // Get position and radius
235                 pos = particleHandler.getObject(i)->getPosition();
236                 radiusPar = particleHandler.getObject(i)->getRadius();
237 
238                 // If bottom is touched
239                 if (pos.Y-radiusPar <= getYMin())
240                 {
241                     // If within discharge port region
242                     if (std::abs(pos.X) > radiusBottomBF-widthPort &&
                        std::abs(pos.X) < radiusBottomBF)
243                     {
244                         // Save particle id
```

```
245                             idParInRegion.push_back(i);
246                             // Increase counter number of particles in region
247                             numParInRegion++;
248                             // If random insertion and enough particles to remove
249                             if (randomInsertion && numParInRegion==numParToBeRemoved)
250                             {
251                                 // Break from loop
252                                 break;
253                             }
254                         }
255                     }
256                 }
257 
258                 // If less particles in region than number to be removed
259                 if (numParInRegion < numParToBeRemoved)
260                 {
261                     numParToBeRemoved = numParInRegion;
262                 }
263                 // Else if non random insertion
264                 else if (!randomInsertion)
265                 {
266                     // Pick random particles to remove, not needed when
                        numParInRegion<numParToBeRemoved since they're all removed anyway
267                     // Arrange particle ids in random order
268                     std::random_shuffle(idParInRegion.begin(),idParInRegion.end());
269                     // Sort particle ids of first number of particles to be removed in
                        ascending order
270 
                        std::sort(idParInRegion.begin(),idParInRegion.begin()+numParToBeRemove
                        d);
271                 }
272 
273                 // For number of particles to be removed, highest id first
274                 for (int i=numParToBeRemoved-1; i>=0; i--)
275                 {
276                     // Get particle mass
277                     massPar = particleHandler.getObject(idParInRegion[i])->getMass();
278                     // Remove particle
279                     particleHandler.removeObject(idParInRegion[i]);
280                     // Update mass to be removed
281                     massToBeRemoved -= massPar;
282                 }
283             }
284         }
285 
286 
287         void setDensity (double d)
288         {
289             rho = d;
290         }
291 
292         void setRadius (double r, double rv)
293         {
294             radius = r;
295             radiusVariation = rv;
296         }
297 
298         void setCOR (double wallCOR, double parCOR)
299         {
300             CORWall = wallCOR;
301             CORPar = parCOR;
302         }
303 
304         void setStiffness (double k)
305         {
306             stiffness = k;
307         }
308 
309         void setFrictionCoefficient (double sfc, double rfc)
310         {
311             slidingFrictionCoefficient = sfc;
312             rollingFrictionCoefficient = rfc;
313         }
```

```
314
315         void useInfiniteWalls (bool uif)
316         {
317             useInfWalls = uif;
318         }
319
320         void setDischargePort (double w)
321         {
322             widthPort = w;
323         }
324
325         void setDischarge (double fr, bool ri)
326         {
327             flowRate = fr;
328             randomInsertion = ri;
329         }
330
331         void setFurnace (double rb, double rt)
332         {
333             radiusBottomBF = rb;
334             radiusTopBF = rt;
335         }
336
337     private:
338         double rho, radius, radiusVariation, mass;
339         double CORWall, CORPar, stiffness;
340         double slidingFrictionCoefficient, rollingFrictionCoefficient;
341
342         bool useInfWalls, randomInsertion;
343
344         double widthPort;
345         double flowRate;
346         double massToBeRemoved=0;
347
348         double radiusBottomBF, radiusTopBF;
349     };
350
351     int main(int argc, char** argv)
352     {
353         BF problem;
354
355         problem.setSystemDimensions(3);
356         problem.setGravity(Vec3D(0,-9.81,0));
357         problem.setXMin(-0.215);
358         problem.setYMin(0);
359         problem.setZMin(0);
360         problem.setXMax(0.215); // Half the total width of bottom
361         problem.setYMax(0.8);
362         problem.setZMax(0.03); // Depth of furnace
363         problem.dataFile.setFileType(FileType::ONE_FILE);
364         problem.restartFile.setFileType(FileType::ONE_FILE);
365         problem.fStatFile.setFileType(FileType::ONE_FILE);
366         problem.eneFile.setFileType(FileType::ONE_FILE);
367         problem.setWallsWriteVTK(FileType::ONE_FILE);
368         problem.setParticlesWriteVTK(1);
369
370         problem.setCOR(0.25,0.25); // Particle-wall, particle-particle
371         problem.setStiffness(40000);
372         problem.setFrictionCoefficient(0.25,0.40); // Sliding, rolling
373         problem.setFurnace(0.18,0.137); // Radius bottom, radius top
374
375
376         //Add any configuration which is modified often below...
377         problem.setName("BF"); //Make sure to add an (unique) name here
378         problem.setTimeMax(100);
379         problem.setTimeStep(1e-5);
380         problem.setSaveCount(2500); // 40 fps with timestep 1e-5
381
382         problem.setDensity(2500);
383         problem.setRadius(0.003,0.000); // Radius, radius variation
384         problem.useInfiniteWalls(1); // 1: Infinite Wall; 0: Periodic Boundary
385         problem.setDischarge(1.64,1); // Mass flow rate (200 g/min), random insertion
            (1: yes, 0: no)
```

```
386         problem.setDischargePort(0.035); // Width 0.035 OR 0.03 DIFFERS PER PAPER!!
387
388
389         //Now, start the simulation
390         problem.solve(argc, argv);
391         return 0;
392     }
393
```

Appendix E

Conveyor Belt

E.1 Python code for getting the volume flow rate

```
import math
import sys

# The offset moves the baseline from the origin to the flat roller
offset = 360.0

# Corners left and right where the flat and angled rollers meet
cornerLeft = -130+2.5
cornerRight = 180+2.5
angleRoller = 45

# Open the xbox data, which is given as an argument in the command prompt
with open(sys.argv[1]) as f:
    # Every line in f corresponds to one timestep
    for line in f:
        # Separate line by comma's to get array (0: date; 1-640: data points)
        t = line.split(',')

        # Initialize empty x, y and baseline array
        x = []
        y = []
        baseline = []
        beltempty = False
        # For every datapoint, so ignoring the first (index 0) element (date)
        for i in range(1,len(t)):
            # If the value is a number
            if t[i] != '-nan':
                # Convert datapoint (from string) to float, which is the distance to the center (xbox)
                dist = (float(t[i]))
                # The angle of each point is calculated with the ratio of half the total angle (28.5) and
                # half the total number of points (320). This is assuming the points are evenly distributed,
                # i.e. the xbox sensor is perfectly horizontal.
                thetax = (28.5*(i-320.0)/320.0 * 3.14/180.0)

                # Calculate x and y position and add to array
                x.append(dist * math.sin(thetax))
                y.append(1500.0 - dist * math.cos(thetax))

            # If the value is not a number: add to array, but give values outside system and below baseline
            else:
                x.append(-600)
                y.append(-1)

            # Previously the baseline offset was estimated by taking it as the middle of a empty belt.
            # This of course varies a bit over time and is thus less predictable. Besides that it still needed
            # some correction (e.g. -20), since a running empty belt lifts itself a bit from the flat roller.
#            # If belt empty, reestimate offset
#            if i == 340 and t[i] != '-nan':
#                if y[i-1] < 400:
#                    offset = y[i-1] # -20
#                    beltempty = True
#                    #print("beltempty",offset)
#                else:
#                    beltempty = False

        # Sort x (and corresponding y) for more accurate area calculations
        y = [i for _,i in sorted(zip(x,y))]
        x = sorted(x)
        area = 0.0
        # For every x- and y-coordinate
        for i in range(len(x)):
            # Recompute the baseline based on the x-coordinates
            # Left angled roller
            if x[i] < cornerLeft:
                baseline.append(-math.tan(angleRoller*math.pi/180)*(x[i] -
```

```
                cornerLeft) + offset )
66              # Flat roller
67              elif x[i] < cornerRight:
68                  baseline.append(offset)
69              # Right angled roller
70              else:
71                  baseline.append(+math.tan(angleRoller*math.pi/180)*(x[i] -
                    cornerRight) + offset )
72
73              # Calculate area baseline and measurement between two consecutive points
                and y=0.
74              # Distance between x-coordinates times average of y-coordinates.
75              if i > 1:
76                  areabaseline=(abs(x[i-1]-x[i]) * (baseline[i-1] + baseline[i])*0.5)
77                  areameasurement=(abs(x[i-1]-x[i]) * (y[i-1] + y[i])*0.5)
78                  # Partial area is the difference between both areas
79                  A = areameasurement - areabaseline
80                  # Only add partial area when it's positive, so only measurements
                    above baseline add to total area
81                  if A > 0.0:
82                      area = area + A
83
84          # Calculate volume flow rate (and approximate mass flow rate)
85          beltdisplacement = 2.15 * 10.0 # m/10s
86          volrate = (area * 0.000001) * beltdisplacement # 0.000001 m2/mm2 -> m3/s
87          bulkdensity = 800 # kg/m3
88          massrate = volrate * bulkdensity * 360.0 * 0.001 # ton/h 360 samples/h 0.001
            ton/kg
89
90          # Print date and volume flow rate (and approximate mass flow rate)
91          #print t[0].strip(),",",volrate,",",massrate
92          print t[0].strip(),",",volrate
93
94          # If the profile outline is needed, uncomment the two lines below (pos, print)
95          # and make sure to comment the line that prints the date and volume flow
            rate above
96          # Position array_x,array_y,array_baseline
97          #pos = [str(a) for a in x]+[str(a) for a in y]+[str(a) for a in baseline]
98          #print(', '.join(pos))
99
```

E.2 Octave script for calculating the bulk density

```
clear, clc

# First run profile.py for the raw xbox data. The output csv files containing
# the date/time and volume flow rate are used in this script.

## MANUAL INPUT ######################################################################
## Flags
# Removes delay when xbox flow rate is behind on BW
removeDelay = true;
# Plots day and hour scale. If data is less than one day this will give an error so turn off.
plotDayHourScale = true;
# Plots a "trendline": horizontal lines at average of each cluster of points
plotTrendline = true;
# Saves dates and bulk density as two columns in temporary csv file. It will be
# overwritten every run and is just there for easy copy paste to another csv/excel file.
saveBulkDensity = false;
# Saves figure to fullscreen PDF, with name: BulkDensity_(name)
saveFigPDF = false;


# Single filename of xbox file from output profile.py
name = 'test3';

# OR store multiple filenames in cell array and specify index of filename of interest
# (Typing three dots ... allows to continue on the next line for clarity)
names = {'centerline_2019-03-15_110250_out','centerline_2019-03-18_112120_out',...
         'centerline_2019-04-02_154520_out','centerline_2019-04-05_082540_out',...
         'centerline_2019-04-15_083540_out','centerline_2019-04-16_101330_out',...
         'centerline_2019-05-08_104920_out'};
# Specify index of filename in names
numFile = 5;
# Name is now assigned the numFile^th filename from array with filenames
#name = names{numFile};

# BW filename
BWname = 'BWdata';
# To know which column from BW to read, specify the column number (as given in
# excel) of column containing dates of day of interest. We take the number of
# dates (not mass flow rate), because for reading the column and row numbering
# starts from 0, while in excel is starts from 1. So now the next column after
# the dates will be read, which contains the mass flow rate.
column = 26;
# Or, in case of multiple filenames stored, store all column numbers in array
# corresponding to array with filenames (same order)
columns = [14,17,20,23,26,29,32];
# Column in now assigned the numFile^th column number from array with column numbers
#column = columns(numFile);

# To know until which row in BW to read, specify the row number (as given in
# excel) of last row with data. This can be higher without problem (empty cells
# ar not read, so just take the last row number of the longest column in BW),
# but when it's lower not all data will be read.
lastrow = 10000;


## READ DATA #########################################################################
# Read data BW
BWmassrate = csvread([BWname,'.csv'],[1,column,lastrow-1,column]);
# Read data xbox. Depending on profile.py output containing the mass flow rate
# or not, comment/uncomment these two lines.
#[date,volrate,massrate] = textread([name,'.csv'],'%s %f %f','delimiter',',');
[date,volrate] = textread([name,'.csv'],'%s %f','delimiter',',');


## REMOVE DELAY ######################################################################
## It is possible that there is a delay in data between BW and xbox. This was
## definitely the case for a few datasets starting from 2019-03-15. The delay is
## supposedly solved, so new datasets should not have it. If this is true, this
## section might be completely removed.

if removeDelay
## Check for both xbox and BW the first time the belt is loaded after it has been
## empty. Compare the corresponding values to get rid of delay.
# Initialize
```

```
75  BWbegin = 1;
76  xboxBegin = 1;
77
78  # Index first time empty
79  while BWmassrate(BWbegin) >= 25
80    BWbegin = BWbegin+1;
81  endwhile
82  # Since xbox is the one behind, index should be higher or equal to corresponding index BW
83  # Volume flow rate is exactly zero only when raw xbox data (ran through profile.py)
84  # has a line with only NaN values, which does not mean the belt is empty
85  while volrate(xboxBegin) >= 0.7 || xboxBegin < 6*(BWbegin-1) || volrate(xboxBegin) == 0
86    xboxBegin = xboxBegin+1;
87  endwhile
88
89  # Index first time loaded after empty
90  while BWmassrate(BWbegin) < 25
91    BWbegin = BWbegin+1;
92  endwhile
93  while volrate(xboxBegin) < 0.7
94    xboxBegin = xboxBegin+1;
95  endwhile
96
97  # Delay is difference between corresponding indices
98  delay = xboxBegin-1-6*(BWbegin-1);
99  # Get rid of delay: remove first few elements from array
100 # Date array stays intact, since it's not delayed (only the flow rate is)
101 if delay > 0
102   volrate(1:delay) = [];
103 endif
104 endif
105
106
107 ## CALCULATE DENSITY ###########################################################
108 # BW and Xbox measure respectively every 1 minute and 10 seconds, so number of
109 # full minutes for xbox is number of elements in volrate divided by 6 and rounded down
110 numMinutes = floor(length(volrate)/6);
111 # Average volume rate per minute. First reshape the array to a matrix with 6 rows
112 # (every 10 seconds) and the number of columns equal to the number of full minutes.
113 # Then take the mean of this matrix (which does this column wise). The result is
114 # a row array, which is transposed (') to give a column array.
115 volrateMean = mean(reshape(volrate(1:numMinutes*6),6,numMinutes))';
116
117 # BW and xbox can only be compared for as long as both have data, so the maximum
118 # index for which comparing can be done is equal to the length of the shortest array.
119 idxMax = min(length(BWmassrate),length(volrateMean));
120 # Calculate density (BW from ton/h to kg/s)
121 density = BWmassrate(1:idxMax)*1000/360./volrateMean(1:idxMax);
122
123 # Time array. Take every 6th date, i.e. every minute. Uncommenting the last
124 # brackets will give only the time (without date).
125 time = char(date(1:6:idxMax*6));#(:,end-7:end);
126 # Temporary "time" array used for plotting x positions
127 timex = [1:length(time)]';
128
129 # Save bulk density to temporary file
130 if saveBulkDensity
131   # Open file
132   fidTemp = fopen('BulkDensityTemp.csv','w+');
133   for i=1:length(density)
134     # Print to file
135     fprintf(fidTemp,[time(i,:),',',num2str(density(i)),'\n']);
136   endfor
137   # Close file
138   fclose(fidTemp);
139 endif
140
141
142 ## CLUSTERS ####################################################################
143 ## First find the start and end indices of all clusters. Then calculate the a
144 ## average density of each cluster. A 1st order polynomial fit has also been tried
145 ## (and can still quickly be uncommented to try it out), but resulted in a mess.
146
147 # Get indices of density in appropriate region (ignore empty belt and other irregularities)
148 idxCluster = density>650 & density<900;
149 # Get difference of consecutive terms, which gives 1 and -1 at respectively the
```

```
150  # start and end indices of the clusters and zeros everywhere else.
151  idxClusterDiff = diff([0;idxCluster;0]);
152  # Find indices with difference of 1, indicating start clusters
153  startCluster = find(idxClusterDiff>0);
154  # Find indices with difference -1 and subtract 1, to get indices end clusters
155  endCluster = find(idxClusterDiff<0)-1;
156  # For each cluster
157  for i=1:length(startCluster)
158    # Get corresponding part of the temporary "time" array
159    timexFit{i} = timex(startCluster(i):endCluster(i));
160    # Get a 1st order polynomial fit
161    #coeffs = polyfit(timexFit{i},density(startCluster(i):endCluster(i)),1);
162    #densityFit{i} = polyval(coeffs,timexFit{i});
163    # Get array with same length as timexFit and with mean density of cluster as constant
       value
164    densityFit{i} = ones(size(timexFit{i}))*mean(density(startCluster(i):endCluster(i)));
165  endfor
166
167
168  ## PLOT FIGURE ########################################################################
169  hf=figure(1); clf(1), hold on
170  # Plot density as function of (temporary) time
171  plot(timex,density,'.')
172  # Plot lines for scale
173  if plotDayHourScale
174    hourplot=plot([timex(end-60) timex(end)],[860 860],'LineWidth',5); # Hour
175    dayplot=plot([timex(end-60*24) timex(end)],[850 850],'LineWidth',5); # Day
176    legend([hourplot dayplot],'Hour scale','Day scale')
177  endif
178  # Plot "trendline"
179  if plotTrendline
180    for i=1:length(startCluster)
181      plot(cell2mat(timexFit(1,i)),cell2mat(densityFit(1,i)),'k','LineWidth',5)
182    endfor
183  endif
184  # Set axis to fit around area of interest ([xmin xmax ymin ymax])
185  axis([timex(1) timex(end) 650 900])
186  # Change axis labels from temporary time values to actual time
187  # First set number of ticks
188  set(gca,'XTick',[timex(1):2000:timex(end-1000),timex(end)])
189  # Set label of each tick to actual time
190  set(gca,'XTickLabel',{char([time(1:2000:end-1000,:);time(end,:)])})
191  hold off
192
193  # Print to save figure to fullscreen pdf
194  if saveFigPDF
195    print(hf,['BulkDensity_',name,'.pdf'],'-dpdf','-S1280,720')
196  endif
```

Appendix F

Box Test

F.1 Shell script for running multiple simulations

```
#!/bin/sh

# Run this script with AoR.csv as input: ./RunMBox.sh AoR.csv

# Set (part of) name of simulation
simName="NoNameNew8"
# Set bulk density and porosity
bulkdensity=525
porosity=0.51

# Compile simulation code
make MBox
# Copy executable so that the simulation code itself can continued to be worked on and compiled
cp MBox MBox_$simName
# Initialize line number, only used to skip first line
lineNum=0
# For every line in file
while IFS= read -r line
do
    # Increase line number
    lineNum=$((lineNum+=1))
    # Only do stuff once line number > 1, i.e. skip first line
    if [ $lineNum -gt 1 ]
    then
    # Assign values
    set -- $line
    ID=$1
    CORpp=$3
    CORpw=$4
    SFCpp=$5
    SFCpw=$6
    RFCpp=$7
    RFCpw=$8
    # Run simulation with extra command plus arguments (always has to be command + 10 arguments)
    ./MBox_$simName -cor_sfc_rfc $ID $CORpp $CORpw $SFCpp $SFCpw $RFCpp $RFCpw $simName $bulkdensity $porosity
    fi
done < "$1"
```

F.2 Drivers code of the simulation

```
1  //Copyright (c) 2013-2018, The MercuryDPM Developers Team. All rights reserved.
2  //For the list of developers, see <http://www.MercuryDPM.org/Team>.
3  //
4  //Redistribution and use in source and binary forms, with or without
5  //modification, are permitted provided that the following conditions are met:
6  //  * Redistributions of source code must retain the above copyright
7  //    notice, this list of conditions and the following disclaimer.
8  //  * Redistributions in binary form must reproduce the above copyright
9  //    notice, this list of conditions and the following disclaimer in the
10 //    documentation and/or other materials provided with the distribution.
11 //  * Neither the name MercuryDPM nor the
12 //    names of its contributors may be used to endorse or promote products
13 //    derived from this software without specific prior written permission.
14 //
15 //THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
16 //ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
17 //WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
18 //DISCLAIMED. IN NO EVENT SHALL THE MERCURYDPM DEVELOPERS TEAM BE LIABLE FOR ANY
19 //DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
20 //(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
21 //LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
22 //ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 //(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
24 //SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25
26 /*
27 Pretty much the same as box test (Box.cpp) made before, but some improvement and
28 can handle inputs for COR and sliding and rolling friction. Also a small lip is
29 added at the right wall, so particle-wall friction plays less of a roll for the AoR.
30
31 Simple cubic box with periodic boundaries in the y-direction. Once the particles
32 are settled the right side wall is removed and the particles fall away to the side
33 and down. Once they are a bit away from the box they are removed.
34 Goal: measure the angle of repose and compare with experimental data.
35 */
36
37 #include "Species/LinearViscoelasticFrictionSpecies.h"
38 #include "Mercury3D.h"
39 #include "Particles/BaseParticle.h"
40 #include "Walls/InfiniteWall.h"
41 #include "Walls/IntersectionOfWalls.h"
42 #include "Boundaries/PeriodicBoundary.h"
43 #include "Boundaries/DeletionBoundary.h"
44 #include <cstring> // strcmp
45
46 class Box : public Mercury3D
47 {
48 public:
49
50     void setupInitialConditions() override
51     {
52         mass = 4.0/3.0*constants::pi*pow(radius,3)*rho;
53
54         auto speciesWall =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
55         auto speciesPar =
          speciesHandler.copyAndAddObject(LinearViscoelasticFrictionSpecies());
56
57         // Wall species
58         speciesWall->setDensity(rho);
59         speciesWall->setStiffnessAndRestitutionCoefficient(stiffness,CORpw,mass);
60
61         speciesWall->setSlidingStiffness(speciesWall->getStiffness()*2./7.);
62         speciesWall->setSlidingFrictionCoefficient(SFCpw);
63         speciesWall->setSlidingDissipation(speciesWall->getDissipation()*2./7.);
64
65         speciesWall->setRollingStiffness(speciesWall->getStiffness()*2./7.);
66         speciesWall->setRollingFrictionCoefficient(RFCpw);
67         speciesWall->setRollingDissipation(speciesWall->getDissipation()/2./7.);
68
69         speciesWall->setTorsionStiffness(speciesWall->getStiffness()*2./7.);
70         speciesWall->setTorsionFrictionCoefficient(0.1);
71         speciesWall->setTorsionDissipation(speciesWall->getDissipation()*2./7.);
```

```
72
73          // Particle species
74          speciesPar->setDensity(rho);
75          speciesPar->setStiffnessAndRestitutionCoefficient(stiffness,CORpp,mass);
76
77          speciesPar->setSlidingStiffness(speciesPar->getStiffness()*2./7.);
78          speciesPar->setSlidingFrictionCoefficient(SFCpp);
79          speciesPar->setSlidingDissipation(speciesPar->getDissipation()*2./7.);
80
81          speciesPar->setRollingStiffness(speciesPar->getStiffness()*2./7.);
82          speciesPar->setRollingFrictionCoefficient(RFCpp);
83          speciesPar->setRollingDissipation(speciesPar->getDissipation()*2./7.);
84
85          speciesPar->setTorsionStiffness(speciesPar->getStiffness()*2./7.);
86          speciesPar->setTorsionFrictionCoefficient(0.01);
87          speciesPar->setTorsionDissipation(speciesPar->getDissipation()*2./7.);
88
89          // Wall and Particle species
90          auto speciesWallAndPar =
            speciesHandler.getMixedObject(speciesWall,speciesPar);
91
            speciesWallAndPar->setStiffnessAndRestitutionCoefficient(stiffness,CORpw,mass)
            ;
92
93
            speciesWallAndPar->setSlidingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
94          speciesWallAndPar->setSlidingFrictionCoefficient(SFCpw);
95
            speciesWallAndPar->setSlidingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
96
97
            speciesWallAndPar->setRollingStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
98          speciesWallAndPar->setRollingFrictionCoefficient(RFCpw);
99
            speciesWallAndPar->setRollingDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
100
101
            speciesWallAndPar->setTorsionStiffness(speciesWallAndPar->getStiffness()*2./7.
            );
102         speciesWallAndPar->setTorsionFrictionCoefficient(0.1);
103
            speciesWallAndPar->setTorsionDissipation(speciesWallAndPar->getDissipation()*2
            ./7.);
104
105
106         // Wall setup
107         InfiniteWall w0;
108         w0.setSpecies(speciesWall);
109         // Left wall
110         w0.set(Vec3D(-1.0,0.0,0.0),Vec3D(getXMin(),0,0));
111         wallHandler.copyAndAddObject(w0);
112         // Right wall (index 1), which will be removed
113         w0.set(Vec3D(1.0,0.0,0.0),Vec3D(getXMax(),0,0));
114         wallHandler.copyAndAddObject(w0);
115         // Top wall, to prevent particles from shooting up too high while settling
116         w0.set(Vec3D(0.0,0.0,1.0),Vec3D(0,0,2.0*getZMax()));
117         wallHandler.copyAndAddObject(w0);
118
119         IntersectionOfWalls w1;
120         w1.setSpecies(speciesWall);
121         // Bottom wall
122         w1.addObject(Vec3D(0.0,0.0,-1.0),Vec3D(0,0,getZMin()));
123          // Bottom wall: right edge
124         w1.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0));
125         wallHandler.copyAndAddObject(w1);
126
127         // Small lip at right wall
128         IntersectionOfWalls w2;
129         w2.setSpecies(speciesWall);
```

```
130         std::vector<Vec3D> w2Points(4);
131         w2Points[0] = Vec3D(getXMax(),0,getZMin());
132         w2Points[1] = Vec3D(getXMax(),0,getZMin()+0.01);
133         w2Points[2] = Vec3D(getXMax()+0.000001,0,getZMin()+0.01);
134         w2Points[3] = Vec3D(getXMax()+0.000001,0,getZMin());
135         w2.createOpenPrism(w2Points);
136         wallHandler.copyAndAddObject(w2);
137
138
139         // Solid boundaries y-direction
140 //        IntersectionOfWalls w2;
141 //        w2.setSpecies(speciesWall);
142 //        w2.addObject(Vec3D(0.0,1.0,0.0),Vec3D(0,getYMax(),0)); // Back wall
143 //        w2.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Back wall:
    right edge
144 //        wallHandler.copyAndAddObject(w2);
145 //
146 //        IntersectionOfWalls w3;
147 //        w3.setSpecies(speciesWall);
148 //        w3.addObject(Vec3D(0.0,-1.0,0.0),Vec3D(0,getYMin(),0)); // Front wall
149 //        w3.addObject(Vec3D(-1.0,0.0,0.0),Vec3D(getXMax(),0,0)); // Front wall:
    right edge
150 //        wallHandler.copyAndAddObject(w3);
151
152         // Periodic boundaries y-direction
153         PeriodicBoundary b0;
154         b0.set(Vec3D(0.0,1.0,0.0),getYMin(),getYMax());
155         boundaryHandler.copyAndAddObject(b0);
156
157
158         // Deletion boundary angled 45 degrees at right side box
159         DeletionBoundary db0;
160         db0.set(Vec3D(1,0,-1),getXMax());
161         boundaryHandler.copyAndAddObject(db0);
162
163
164         // Particle setup
165         double volumePar = (4./3.*constants::pi*pow(radius,3.0)); // Volume of one
            particle
166         Vec3D pos; // Position particle
167         double radiusPar; // For temporary random particle radius
168         BaseParticle p0;
169         p0.setSpecies(speciesPar);
170         p0.setVelocity(Vec3D(0,0,0));
171
172         int numPar =
            fillFraction*(std::abs(getXMax()-getXMin()))*(std::abs(getYMax()-getYMin()))*(
            std::abs(getZMax()-getZMin()))/volumePar; // Number of particles to be
            inserted
173         int numParInserted = 0, numParInteraction = 0; // Number of particle inserted
174         while( numParInserted < numPar )
175         {
176             radiusPar =
                random.getRandomNumber(radius-radiusVariation,radius+radiusVariation);
177             p0.setRadius(radiusPar);
178
179             int failCounter = 0;
180             do
181             {
182                 pos.X =
                    random.getRandomNumber(getXMin()+radiusPar,getXMax()-radiusPar);
183                 pos.Y = random.getRandomNumber(getYMin(),getYMax());
184                 pos.Z =
                    random.getRandomNumber(getZMin()+radiusPar,2*getZMax()-radiusPar);
185                 p0.setPosition(pos);
186
187                 failCounter++;
188                 if (failCounter==1000)
189                 {
190                     std::cout << "Added particle with interaction" << std::endl;
191                     numParInteraction++;
192                     break;
193                 }
```

```
194             }
195             while (!checkParticleForInteraction(p0));
196
197             particleHandler.copyAndAddObject(p0);
198
199             numParInserted++;
200         }
201
202         std::cout << "Finished creating particles. \nNumber inserted: " <<
            numParInserted << "\nNumber with interaction: " << numParInteraction <<
            std::endl;
203         std::cout << "Particles settling down" << std::endl;
204         parSettled = false; // Particles are not settled
205         checkTime = getTime() + .1; // Time to check if particles are settled
206     }
207
208
209     void actionsBeforeTimeStep() override
210     {
211         // Open the gate once the kinetic energy is low enough to consider the
            particles settled (value found by trial and error)
212         if (!parSettled) // To stop checking the kinetic energy once the particles
            are settled
213         {
214             if (getTime() > checkTime) // Only check at certain times
215             {
216                 std::cout << "Current KE: " << getKineticEnergy() << std::endl;
217                 if (getKineticEnergy() < 0.0001) // For 0.0001 the particles are
                    settled quite well
218                 {
219                     parSettled = true; // Particles are settled, no longer checking
220                     std::cout << "Particles settled" << std::endl;
221                     wallHandler.removeObject(1); // Remove right wall
222                     std::cout << "Gate open" << std::endl;
223                 }
224                 else
225                 {
226                     checkTime = getTime() + .1;
227                 }
228             }
229         }
230     }
231
232
233     bool continueSolve() const override
234     {
235         // CheckTime+1.0 just to be sure not to stop solving while the particles are
            settling
236         // Comparing energies is a useful criteria to determine arresting flow
            (according to website mercuryDPM)
237         if (getTime() > checkTime+1.0 && getKineticEnergy() < 1e-5*getElasticEnergy())
238         {
239             std::cout << "No more flow" << std::endl;
240             return false;
241         }
242         else
243         {
244             return true;
245         }
246     }
247
248     void setDensity (double r)
249     {
250         rho = r;
251     }
252
253     void setRadius (double r, double rv)
254     {
255         radius = r;
256         radiusVariation = rv;
257     }
258
259     void setFillFraction (double ff)
```

```
260         {
261             fillFraction = ff;
262         }
263 
264         void setStiffness (double k)
265         {
266             stiffness = k;
267         }
268 
269         void setCOR_SFC_RFC (char* COR1, char* COR2, char* SFC1, char* SFC2, char* RFC1,
            char* RFC2)
270         {
271             // From char* to double
272             CORpp = strtod(COR1,NULL);
273             CORpw = strtod(COR2,NULL);
274             SFCpp = strtod(SFC1,NULL);
275             SFCpw = strtod(SFC2,NULL);
276             RFCpp = strtod(RFC1,NULL);
277             RFCpw = strtod(RFC2,NULL);
278         }
279 
280         double getLargestParticleDiameter ()
281         {
282             double lpd = 2*(radius+radiusVariation);
283             return lpd;
284         }
285 
286     private:
287         double rho, radius, radiusVariation, mass, fillFraction;
288         double CORpp, CORpw, SFCpp, SFCpw, RFCpp, RFCpw, stiffness;
289 
290         double checkTime;
291         bool parSettled;
292     };
293 
294     int main(int argc, char *argv[])
295     {
296         // Problem setup
297         Box problem;
298 
299         // Initialize simulation ID and (part of) name and bulkdensity and porosity
300         int simID = 0;
301         std::string simName;
302         double bulkdensity = 900, porosity = 0;
303 
304         // Check for additional arguments.
305         // IMPORTANT: When other arguments are passed (e.g. -tmin -tmax) they should be
            put in front of these.
306         // The argument counter (argc) is simply made smaller so the last elements of
            argv are not read in problem.solve().
307         // Not doing this gives errors, because these commands are not known by
            MercuryDPM.
308         // There might be an more elegant way, but this works.
309         for (int i=1; i<argc; i++)
310         {
311             if (!strcmp(argv[i], "-cor_sfc_rfc"))
312             {
313                 if (i+10!=argc-1)
314                 {
315                     std::cout << "WARNING: Make sure -cor_sfc_rfc is the last argument
                        called and has exactly 6 inputs! " << std::endl;
316                 }
317 
318                 // Assign values to variables for the Coefficient Of Restitution,
                    Sliding Friction Coefficient and Rolling Friction Coefficent
319                 // Order: COR particle-particle, particle-wall; SFC particle-particle,
                    particle-wall; RFC particle-particle, particle-wall
320 
                    problem.setCOR_SFC_RFC(argv[i+2],argv[i+3],argv[i+4],argv[i+5],argv[i+6],a
                    rgv[i+7]);
321                 // Set simulation ID (char to int)
322                 simID = strtol(argv[i+1],NULL,10);
323                 // Set simulation name
```

```
324                 simName = argv[i+8];
325                 // Set bulk density and porosity
326                 bulkdensity = strtod(argv[i+9],NULL);
327                 porosity = strtod(argv[i+10],NULL);
328 
329                 std::cout << "Started simulation with name " << simName << " and ID " <<
                    simID << std::endl;
330 
331                 // Decrease argument counter by number of odd arguments
332                 argc -= 11;
333                 // Increase i by number of inputs to skip these and end loop
334                 i+=10;
335             }
336         }
337 
338         problem.setSystemDimensions(3);
339         problem.setGravity(Vec3D(0.0,0.0,-9.81));
340         problem.setXMin(0.0);
341         problem.setYMin(0.0);
342         problem.setZMin(0.0);
343         problem.setXMax(0.1);
344         // YMax is dependent on radius, so set after radius is set
345         problem.setZMax(0.1);
346         problem.dataFile.setFileType(FileType::ONE_FILE);
347         problem.restartFile.setFileType(FileType::ONE_FILE);
348         problem.fStatFile.setFileType(FileType::ONE_FILE);
349         problem.eneFile.setFileType(FileType::ONE_FILE);
350         problem.setXBallsAdditionalArguments(" -solidf -v0 -s 8");
351         problem.setWallsWriteVTK(1);//FileType::ONE_FILE);
352         problem.setParticlesWriteVTK(1);
353 
354         problem.setStiffness(40000);
355         problem.setFillFraction(0.55);
356 
357 
358         // Add any configuration which is modified often below...
359         problem.setName("MBox_"+simName+"_SimID"+std::to_string(simID)); // Make sure to
            add an (unique) name here
360         problem.setTimeMax(10);
361         problem.setTimeStep(1e-5);
362         problem.setSaveCount(10000); // 40 fpx with timestep 1e-5
363 
364         problem.setDensity(bulkdensity/(1-porosity));
365         problem.setRadius(0.0025,0); // Radius, radius variation
366         problem.setYMax(5.*problem.getLargestParticleDiameter()); // Set after radius is
            set
367         // For every new random run, change these parameters to whatever (mercury
            default: 607,273)
368         problem.random.setLaggedFibonacciGeneratorParameters(540,341);
369 
370 
371         // Now, start the simulation
372         problem.solve(argc, argv);
373         return 0;
374     }
375 
```

F.3 Python script for making screenshots

```
# Makes screenshots in Paraview of the last vtu files

from paraview.simple import *
import os
import glob

# Set path to vtu files, (part of) name of simulation and path to save screenshots
os.chdir('c:/Apps/MercuryDPM/MercuryBuild/Drivers/USER/BlueScope/MultiBoxTest')
name = 'CokeNew3'
path = 'c:/Apps/ProjectMercuryDPM/Multi Box Test/ImagesNew'

# Load simulations in any order
Data = glob.glob('./MBox_' + name + '_SimID*Particle_0.vtu')

# Find the maximum number of simulation IDs
maxID = 0
for fileName in Data:
    tokens1 = fileName.split('.')
    tokens2 = tokens1[1].split('_')
    tokens3 = tokens2[-2].split('P') # P of Particle
    tokens4 = tokens3[-2].split('D') # D of SimID
    if int(tokens4[-1]) > maxID:
        maxID = int(tokens4[-1])

# For each simulation
for i in range(1,maxID+1):
    # Load all vtu files in any order
    Data2 = glob.glob('./Mbox_' + name + '_SimID' + str(i) + 'Particle_*.vtu')
    # Find the maximum time step
    maxTime = 0
    for fileName in Data2:
        tokens1 = fileName.split('.')
        tokens2 = tokens1[1].split('_')
        if int(tokens2[-1]) > maxTime:
            maxTime = int(tokens2[-1])
    print 'ID and maxTimeStep =', i, maxTime

    # Load last vtu file and visualise it in Paraview
    particles = XMLUnstructuredGridReader(FileName='./MBox_' + name + '_SimID' +
    str(i) + 'Particle_' + str(maxTime) + '.vtu')
    glyphP = Glyph(particles)
    glyphP.GlyphType = 'Sphere'
    glyphP.Scalars = 'Radius'
    glyphP.Vectors = 'None'
    glyphP.ScaleMode = 'scalar'
    glyphP.ScaleFactor = 2
    glyphP.GlyphMode = 'All Points'
    glyphP.Orient = 0
    Show(glyphP)

    # Set camera to horizontal front view and zoom to fit
    view = GetActiveView()
    view.CameraPosition = [0,0,0]
    view.CameraFocalPoint = [0,1,0]
    view.CameraViewUp = [0,0,1]
    view.Background = [1,1,1] # White
    view.ViewSize = [1920,1080] # Fullscreen
    view.InteractionMode = '2D'
    view.OrientationAxesVisibility = 0
    ResetCamera()

    # Save screenshot
    WriteImage(path + '/MBox_' + name + '_SimID' + str(i) + '.png',view)

    # Delete generated object for next iteration
    Delete(glyphP)
```

F.4 Octave script for getting AoR

```
clear, clc

# Set (part of) name and path to screenshots
name = 'CokeNew3';
path = 'ImagesNew';
# Initialize simulation id
id = 1;
# Plot figures true/false
doPlot = false;

# Open temporary file (This is a temporary file, which will be overwritten every
# run and is just there for easy copy paste.
fidTemp = fopen('AoRTemp.csv','w+');
fprintf(fidTemp,[name,'\n']);

# For all simulations
while exist([path,'/MBox_',name,'_SimID',num2str(id),'.png'])
# Load image
rgbImage = imread([path,'/MBox_',name,'_SimID',num2str(id),'.png']);
# Get number of rows and columns
[rows,columns,numberOfColorBands] = size(rgbImage);

# Extract the individual red, green and blue color channels
#redChannel = rgbImage(:,:,1);
greenChannel = rgbImage(:,:,2);
#blueChannel = rgbImage(:,:,3);

# Get the binary image
binaryImage = greenChannel < 200;

# Plot originial and binary image
if doPlot
  figure(1), clf(1)
  subplot(2,1,1)
  imshow(rgbImage)
  title('Original')
  subplot(2,1,2)
  imshow(binaryImage)
  title('Binary')
endif

# Find the baseline
verticalProfile = sum(binaryImage,2);
topBed = find(verticalProfile,1,'last');
horizontalProfile = sum(binaryImage);
sidesBed = [find(horizontalProfile,1), find(horizontalProfile,1,'last')];

# Arrays for x,y position bed
x = sidesBed(1):sidesBed(2);
y = zeros(size(x));

# Scan across columns finding where the top of the hump is
for i=1:length(x)
  col = x(i);
  yy = topBed - find(binaryImage(:,col),1,'first');
  if isempty(yy)
    y(i)=0;
  else
    y(i)=yy;
  endif
endfor

% Cut off 10% at both sides for more accurate representation of AoR
cutoff = round(diff(sidesBed)*0.1);
xAoR = x(cutoff:end-cutoff);
yAoR = y(cutoff:end-cutoff);

# Fit straight line through points
p = polyfit(xAoR,yAoR,1);
y1AoR = polyval(p,xAoR);

# Calculate AoR
AoR(id) = atand((y1AoR(1)-y1AoR(end))/(xAoR(end)-xAoR(1)));

# Write to temporary file
```

```
76  fprintf(fidTemp,[num2str(AoR(id)),'\n']);
77
78  # Plot outline and AoR line
79  if doPlot
80    figure(2), clf(2), hold on
81    axis equal
82    plot(x,y)
83    plot(xAoR,y1AoR)
84    hold off
85  endif
86
87  # Increase simulation id
88  id = id+1;
89  endwhile
90  # Close temporary file
91  fclose(fidTemp);
```