Modelling the Properties of Packed Bed Structures Formed During Filtration

William Eales

Department of Chemical and Process Engineering
University of Strathclyde

This dissertation is submitted for the degree of *Doctor of Philosophy*

2023

Signed: William Eales

Date: 27/09/2023

Acknowledgements

I would like to thank my supervisors Prof. Chris Price and Dr. Paul Mulheran, as well as William Hicks and Colm Cotter from Astra Zeneca, for their guidance and direction during my PhD.

I would also like to thank Lewis Cartwright and Luke Convery for their contributions.

I was supported through an Engineering and Physical Sciences Research Council iCASE Award co-funded by AstraZeneca EP/T517665/1.

COVID-19 Statement

This project was affected by the COVID-19 pandemic, however due to its nature as a computational project, it was still able to progress during lockdown, even if at a slightly slower pace. The main effect the pandemic had on this project was the decreased accessibility to my supervisors and others in my department, as whilst email contact was maintained, there was a natural delay compared to being able to discuss issues or ideas in person.

Abstract

Agglomeration is an issue that causes many problems during secondary processing for pharmaceutical companies, causing material to need further processing, and costing additional time and resources to ensure a satisfactory outcome. A potential source of agglomeration arises from the particle contacts established during filtration that lead to robust agglomerates forming during drying, so that a necessary first step towards understanding agglomeration is to study the packing properties of filtration beds. Here I present two and three-dimensional models simulating the formation of packed bed structures during filtration.

These models were coded from the ground up using the ForTran programming language, starting with the 2D algorithm as it was a simpler algorithm compared to jumping straight to 3D systems. Once an algorithm was formed that could create realistic 2D systems of packed circular particles, it was extended so that it could also create systems of spherical particles in 3D. A variety of improvements and modifications were made to the algorithm as part of this change, including adding a stochastic optimisation function for determining particle locations, which was found to be a much more efficient method than the equations used in the 2D algorithm, so the stochastic optimisation method was used in all algorithms going forwards. The final modification made to the algorithm was the option to create systems formed of chain structures, made up of circular particles attached together; this enabled the investigation of more realistic systems.

These models use circular and spherical particles of different sizes, mimicking the bimodal particle size distributions sometimes encountered in industrial practice. The systems containing chain particles made up of these circular particles were varied by both particle size and chain length to observe the effect of these parameters on systems with more realistic particles. The statistics of packing and void formation, the distribution of inter-particle

contacts and percolation structures, and the breakage of these systems under shear forces, are presented and discussed in the context of filtration, drying and agglomeration. The 3D model was also compared to current industry standard software, Ansys Rocky DEM, as part of a placement with AstraZeneca, where it was found my model produces very similar packing fraction outputs to those produced by Rocky DEM.

The model paves the way for predictive capabilities that can lead to the rational design of processes to minimise the impact of agglomeration.

Contents

1 Introduction

1.1 Aims of Project

The aim of this project is to better understand how agglomeration occurs during particle drying, and how it can be mitigated. To do this I have designed a computer model that can emulate the packing of particles into filter beds. This model will enable us to gain greater understanding of the properties of these packed beds, and how those properties, such as particle size distribution and the presence of percolation structures potentially affect agglomeration and fragmentation. At this time, the model does not contain enough features to describe agglomeration but is in a position to be used in future projects, therefore this thesis is mainly investigating the packing of shapes in 2D and 3D.

The model will start with circular and spherical particles due to the ease of creation as well as aligning with laboratory work undertaken at the University by PhD student Mariam Siddique investigating the agglomeration of glass beads as an insoluble substitute for crystalline particles. This will allow future comparisons of the systems created by the model with physical examples.

I have also upgraded the model to be able to produce systems involving non-spherical particles, potentially allowing us to investigate specific crystalline Active Pharmaceutical Ingredients (APIs).

The distribution of forces between these particles when a shear stress is applied has also been investigated in collaboration with MEng project students, so that weak points of the packed bed structure could be identified, to better understand how these systems would break apart under stress. The impact of structural properties, such as how the particles in the bed are arranged, can be explored in future work.

This project whilst producing this model has also given me a lot of experience into both regular programming workflow, as well as the work that goes into producing a computational model, even one as simple as mine. As part of using this model, I have shown that it produces realistic systems consistent with those formed under gravity, with parameters, such as packing fractions and number of contacts per particle, that fall within the calculated minimum and maximum ranges. As stated above, the model did not reach the stage during this project where it could be applied further beyond testing initial systems, so we were not able to investigate larger, more realistic systems.

1.2 Significance of Project

Agglomeration during pharmaceutical processing, particularly drying, can cause many problems further down the line such as ensuring the content uniformity of tablets. Whilst the mechanics of how agglomerates are formed are known, little is known about the best practices to avoid it. The aim is that the information gained from analysing the beds created with our model will give us greater insight into how agglomeration can be lessened and its negative repercussions prevented.

1.3 Limitations of Project

Due to gravity being the only force accounted for (in an approximate way) during the bed formation, our model produces a more simplistic representation of a packed bed system, than if all the forces between the particles, such as friction in a dry system or hydrodynamics in a wet system, were accounted for. This route was chosen to allow us to produce a model that enables us to have close control over its direction to keep it in line with the scope of the project, as well as significantly decreased computational times compared to more complex models.

1.4 Structure of Thesis

In the pages following this introduction, I will review the literature surrounding agglomeration, how agglomerates form and their impact on the pharmaceutical production process, as well as modelling as a scientific tool.

The next chapter will focus on the stages of development the 2D model underwent along the course of my project, as well as a step by step look at how the model runs to create a bed system. Following that will be a similar walkthrough and description of the 3D algorithm.

Then there will be a discussion of the results I obtained from investigating the properties of the systems created by the 2D model, and then a discussion of the systems created by the 3D model.

Next will follow a description of the algorithm used for modelling non-spherical particles, as well as the results from the systems it produced.

As part of my project, I undertook a three-month placement at AstraZeneca, Chapter 7 discusses the work performed as part of that placement.

Finally, the conclusions gained from these investigations will be discussed.

2 Literature Review

The global pharmaceutical industry provides medicines for the world population, now approaching eight billion people[1], and has an annual turnover of around one trillion pounds[2]. The vast majority of these medicines are supplied as tablets and capsules[3] in which the API exists as a crystalline solid formulated with multiple excipients to aid both the formulation process and as vehicles for carrying the APIs. As a consequence of this the physical properties of the APIs, including particle size distribution, crystal shape and the extent of agglomeration, are often critical quality attributes of the API because they play an important part in powder flow and hence formulation performance.

There are many problems that can occur during secondary processing of APIs. The aim of this project is to investigate agglomeration, as its effect on the filtration process is of great importance to AstraZeneca, who are funding this work, both how it occurs and what can be done to prevent it. This is in the hope of creating a model to better understand its phenomenology and then to obtain new insight into how best to negate its effects. As previously stated, the model described within is not yet capable of completing this task.

2.1 Agglomeration

2.1.1 What it is and its mechanisms

Agglomeration can be defined as the process of particulate solids gathering into an agglomerate, which is a robust cluster of these particulate solids[4]. This can sometimes be preferable as the larger agglomerates have better flowability compared to groups of smaller particles. However, during drug processing agglomeration can result in various inconveniences, such as affecting content uniformity, through increasing variation in the quantity of API contained within individual tablets, and damaging processing machinery[5].

The binding mechanisms present during agglomeration, shown in Figure 2.1, were defined and ordered by H. Rumph et al[4] and will each be briefly explained below. My model assumes solid bridges are the mechanism by which particles are joined together within the systems, shown in Figure 2.1a.

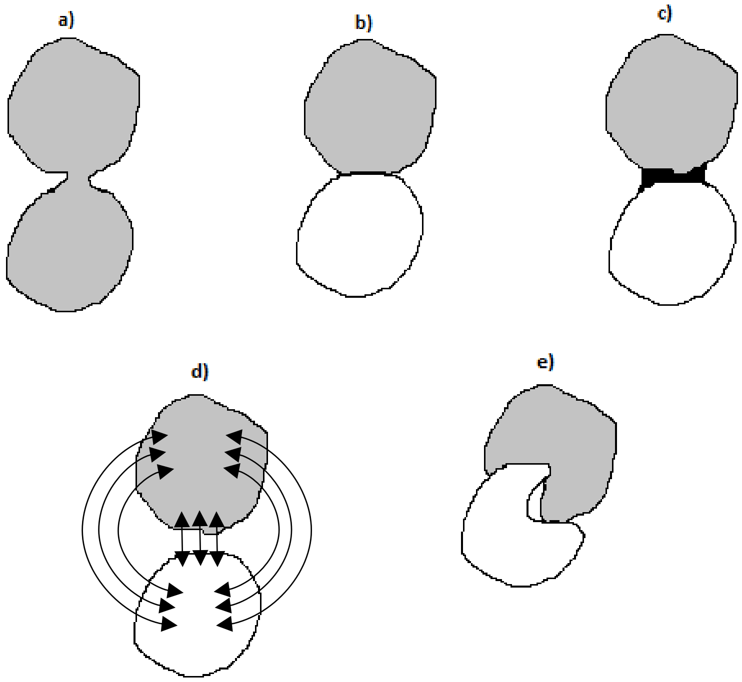


Figure 2.1: Representation of the different mechanisms for Agglomerate formation (redrawn from [6]) a) Solid Bridges b) Adhesion and Cohesion Forces c) Surface Tension and Capillary Forces d) Attraction Between Solid Particles e) Interlocking Bonds

2.1.1.1 Solid Bridges (Figure 2.1a)

In systems where the temperature rises sufficiently, the particles start to melt. This can result in the particles merging at contact points with other particles, when the temperature cools the melted material solidifies causing the particles to fuse together, forming an agglomerate. Solid bridges can also occur even below the melting temperature of the solids present, as diffusion of atoms or molecules can occur across the contact points, over time forming bridges. This process is known as sintering and the bridges called sinter bridges. This heat can be from a deliberate heat source due to the processing requirements of the reaction or to enable agglomeration in reactions where it is favourable. This is mostly applied in industries that process minerals and ores, to combine fines into agglomerates for easier handling[7], and not in pharmaceutical processes as high temperatures can lead to chemical degradation[4].

2.1.1.2 Adhesion and Cohesion Forces (Figure 2.1b)

In the instances when agglomeration is preferred, a binding agent can be added to the system to aid in the formation of agglomerates and to increase their strength. These are generally viscous substances that cause particles to stick together by filling in the gaps between them. Resin and tar are used in non-medical applications, however there are also binders that are suitable for pharmaceutical processing[4] such as sugars like sucrose and liquid glucose and binders such as microcrystalline cellulose[8].

2.1.1.3 Surface Tension and Capillary Forces (Figure 2.1c)

Liquid bridges can also form between particles which have strong forces that maintain the bond between the particles. These forces are created by a negative capillary pressure that occurs when a liquid is filling the whole pore volume between two particles, causing the particles to be pulled together[4]. If the liquid is a solvent used in a previous purification process, when it evaporates it will leave behind the dissolved impurities and API, which can then form solid bridges between the particles. These bridges then act as bonding agents in the solvent's place, holding the particles together[9]. This is one of the most common mechanisms of agglomeration in pharmaceutical manufacturing.

One example of this, known as Snowballing or balling up, occurs during agitated drying with too much solvent present. The agitation causes the clusters to move throughout the system, allowing them to come into contact with more solvent-wet particles, which then join the cluster. This process then repeats with very large agglomerates forming due to the increased opportunities to bind to particles.[9]

2.1.1.4 Attraction Between Solid Particles (Figure 2.1d)

It is possible that there are interactions occurring between the particles that cause them to attract each other, such as hydrogen bonding, if there is a suitably electronegative atom present, as well as the van der Waals forces that occur at all solid surfaces.[4]

2.1.1.5 Interlocking Bonds (Figure 2.1e)

Interlocking Bonds occur when the particles have irregular shapes so that they can intertwine and become entangled with each other, so forming the agglomerates.[4] This occurs more readily in systems of needle like particles, where groups of particles snag on each other as they pass. Such agglomerates often look like sea urchins.

2.1.2 Problems caused by Agglomeration

Agglomeration frequently occurs during secondary processing, particularly during washing and drying. Due to their ability to differ greatly in size and strength, agglomerates cause various problems during the processing of drug products, some of which are discussed below.

2.1.2.1 Content Uniformity

Content uniformity is a property that needs to be maintained to ensure quality control of capsules and tablets and is assessed as follows. Randomly selected capsules or tablets are taken from a batch of product and then tested to determine if they each contain an amount of active ingredient that falls within the acceptable range[10]. Multiple studies to determine the effectiveness of methods of ensuring content uniformity have been carried out.[11, 12, 13, 14]

There are guidelines which set out the acceptable ranges for how much drug substance should be present in each type of tablet, capsule and other dosage types. Maintaining content uniformity across the tablets is essential[6] as if tablets or capsules are produced that contain too little active ingredient, a patient will not be getting the treatment they need, and if a tablet or capsule contains too much active ingredient, it increases the possibility of an overdose[15].

The variation in size between agglomerates results in a broad particle size distribution within the system. This in turn increases the difficulty of maintaining content uniformity between individual tablets or capsules. Therefore, the presence of agglomerates within a processing system will result in extra care having to be taken to ensure the consistency of the tablets, through specific use of solvents to try and prevent agglomeration or by further processing, typically milling, after agglomeration has occurred.[15]

2.1.2.2 Processing Issues

The formation of agglomerates also greatly decreases the efficiency of drug processing, as any product that is part of a large agglomerate needs to go through further processing. Some agglomerates can be too difficult to break open due to the strength of the contacts. Some agglomerates may have impurities trapped inside, meaning it is not always cost effective to send the material through processing a second time[16]. This may occur when mother liquor is trapped inside agglomerates during ineffective washing, thus the agglomerates can take longer to fully dry, and the purity of the product is impacted[17].

Particularly large and strong agglomerates can also cause damage to the machinery itself, due to their size and strength blocking powder flow within the machines and grinding against parts, as well as causing difficulties in removing the batch so it can be processed further[9].

2.1.3 Continuous Systems

The importance of being able to efficiently process larger quantities of product at the same time is relatively obvious, as it would potentially reduce the costs and time that would be required to process the same amount in multiple smaller batches. A good deal of the work investigating the drying process has been done into how to scale it up to be able to process larger quantities or to create a continuous system.[18]

A continuous system would speed up the process as the system would be capable of removing its own waste and transferring the products onto the next step of processing without someone having to be present to do it. This means, in principle, that the machines would require less supervision and also increase the speed of the process as any transfer time between steps of the process would be cut out, as well as allowing for full end to end processing.[19]

One of the issues with creating a continuous system is ensuring that the machinery is able to deal with any unwanted circumstances that occur within the system during each of the processing stages; for example, if agglomerates form during drying, they should not be ignored and passed along onto the next processing stage. Instead, they would need to be separated out, broken down and potentially rewashed or dried before being able to be added back into the system. Additionally, if an error occurs within the system, it is easier to identify in a batch system where the issue originated so a fix can be sorted. Whereas in a continuous system, it can be hard to identify at what point the issue arose.[19]

2.2 Modelling

Computational modelling of complex systems has become more widely available due to the increase in computational power over time. This has allowed researchers to analyse systems that were previously difficult to investigate experimentally, for example due to stability issues or lack of availability of reactants.[20]

One of the ways in which computational modelling can be utilised is as a predictive tool prior to undertaking experimental work. This allows us to investigate the proposed experiment ahead of time, before any reactants are potentially wasted, and to ensure that the experiment would act as planned and produce a useful result. The model can also help show the preferred conditions for the experiment to run under; this should reduce negative effects and prevent accidents.[21]

Another way in which modelling can be applied is alongside experimental work, so that the data produced from the model can be compared with data collected from physical experiments. This is especially useful whilst the model is relatively new to ensure that the data it is producing is similar to the results from a physical experiment to check whether there may be a calculation error within the model. But once the model has been validated, it can also be used to produce data that would otherwise be difficult to collect with laboratory experiments, for example due to lack of access to reactants, danger to researchers or unrealistic time lengths.[22]

Some examples of types of models and their capabilities are discussed below.

2.2.1 Molecular Dynamics

Molecular dynamics (MD) is a type of simulation that analyses the physical movement of molecules. The most common versions of MD simulations use Newton's equations of motion to calculate the trajectories of the particles; the forces between them are calculated using interatomic potentials or molecular mechanics force fields[23]. As molecular systems generally consist of large quantities of particles, it is often impossible to determine properties of very complex systems analytically, therefore MD simulations use numerical methods. As a result, these systems are an approximation to reality, often not covering all the complexities a real system would have to compromise for the computational power that would be required to run a truly realistic simulation. This makes longer MD simulations less viable, as a single error early in the simulation would propagate throughout the simulation potentially causing the later stages to be far less accurate. Algorithmic developments mean that such an error is more likely to be from the user, in the way that they set up the simulation, rather than a failure of the numerical procedures.

One advantage of MD simulations is that they work on an atomistic level therefore they can give information about the molecular detail of a system. However, as with most computer simulations, MD is computationally intensive and depending on the specific system being investigated could require a dedicated computer setup to run. Due to the timeframes of MD simulations being extremely short (typically on the 100 ns timescale), it can mean that vast quantities of simulations need to be run to gather enough data on a system[24]. Additionally, MD simulations are not useful when investigating large scale systems due to their small-scale nature (typically on the 10 nm scale), so a different modelling method would be needed, otherwise the MD simulation runtime would be unacceptably long.

2.2.2 Discrete Element Method

Discrete Element Modelling (DEM) is closely related to molecular dynamics, where it differs is that it includes more complicated geometries as well as rotational degrees of freedom. An important distinction is that the elements of the model are the granular particles rather than atoms, so it works on much longer length and time scales. The force fields and equations of motion used must therefore embody all the relevant physics.

A DEM simulation works by setting up a model with all of the particles placed within it and given an initial velocity. The forces acting on each particle are then calculated based on factors such as friction, gravity, or attractive and repulsive forces between particles. These forces and the initial velocities can then be used to compute an updated location of each particle following a short time step. These updated positions are then used to calculate the next round of forces, and the process then loops until the simulation ends. More detail is giving on the workings of a DEM system in section 7.2.

DEM has many advantages, including its ability to simulate a variety of particle flow mechanics, as well as being able to be implemented into other engineering applications. DEM also allows for more detailed analysis of powder systems than would normally be achievable using physical experiments, allowing for a greater range of data to be collected.

As with other computational modelling, the extent of the system being investigated is limited by the computational power available. Due to DEM being relatively computationally intensive compared to other model types, its capabilities are limited in relation to the size of the system being analysed and the duration of the simulations being run[25, 26]. In a simulation of a fluidized bed, a time step of 10^{-3}s has been used[27], with other experiments using smaller timesteps of down to 10^{-5}s[28].

During my project I used Rocky DEM modelling software (part of ANSYS), however there are many more packages available. Another source of DEM modelling is EDEM[29], simulation software that uses DEM simulations for "bulk and granular material simulation". EDEM has been used across multiple industries in different applications, such as investigating the strength of potato starch agglomerates for the food industry[30], simulations of fluidized beds[27], and modelling granular flow of systems to analyse the effect of different blade shapes[31].

Within the filtration space, DEM has been used in multiple instances to simulate filtration processes, for example to determine the porosity of systems[32] or to compare wet and dry filtration, where either hydrodynamic or gravitational forces are used to filter the small particles[33]. Simulations have also been done with a variety of particle shapes and sizes, ranging from more spherical particles[34] to fibrous particles[35, 36].

Whilst the above methods could have been used for this project, the approach of creating a new model was decided upon as it ensures that we had direct control over the direction and application of the model, keeping it simple compared to other models to aid in the speed of bed creation, as well as providing a unique learning opportunity for me as part of the project.

2.3 Areas Investigated

There are many different parameters across processing that affect agglomeration, as detailed in Figure 2.2.

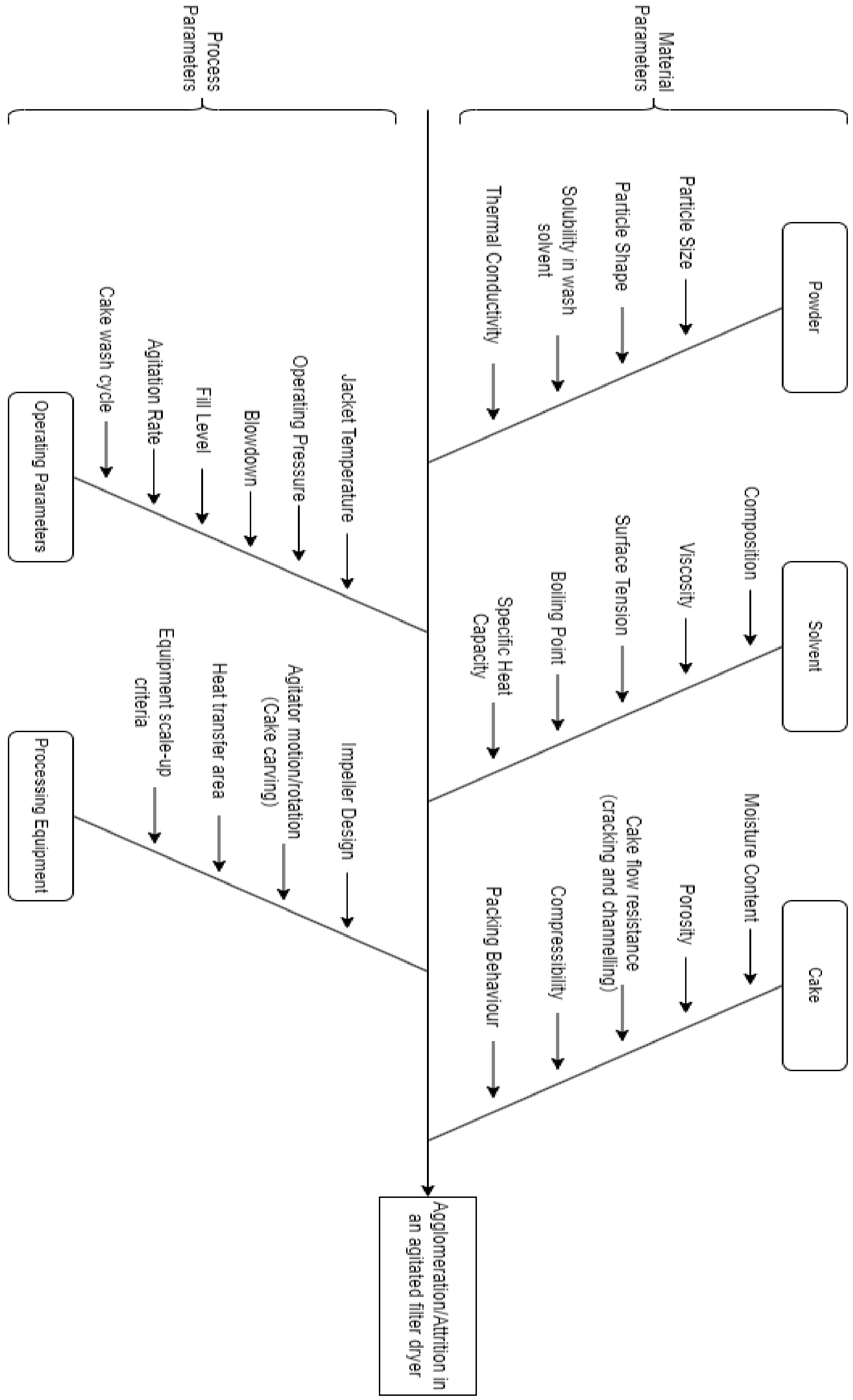


Figure 2.2: Parameters that influence agglomeration in agitated filter dryers (redrawn from [37])

I aim to investigate some of these parameters, specifically the particle size and shape, particle size distribution, and the packing behaviour, through investigating simulated packed bed structures. Discussed below is some of the previous research into each of these properties and how they are relevant to agglomeration.

2.3.1 Packing

2.3.1.1 Packing of objects in two dimensions (2D)

The packing of shapes in 2D has been extensively researched and many models have been created to determine the possible packing fractions under different circumstances. The most random of these types of packing is Random Sequential Adsorption (RSA), where “particles” are added to a system entirely at random, with the only restriction being that they cannot overlap. This results in low packing fractions, with the maximum packing fraction when using RSA in a system with single sized circular particles being roughly 0.547[38], due to the lack of order.

Previous investigations have also looked into the maximum possible packing of different systems. The highest packing fraction possible in a system of identical circles is $\frac{\pi}{\sqrt{12}} \approx 0.9069$[39], when the circles form a triangular lattice.

2.3.1.2 Packing of objects in three dimensions (3D)

Packing of 3D shapes has also been previously investigated. When using identical spherical shapes, there are two lattices that can occur to achieve the highest packing fraction[40], which is $\frac{\pi}{3\sqrt{2}} \approx 0.74048$[41]. These two lattices, as seen in Figure 2.3, are face-centred cubic (FCC) and hexagonal close-packed (HCP) and are formed dependant on the symmetry of the system.

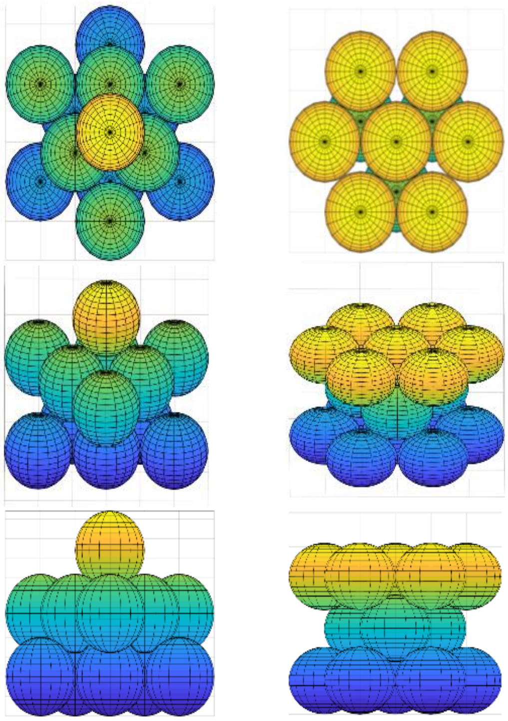


Figure 2.3: An FCC lattice (left) and HCP lattice (right) (redrawn from [42])

Other examples of packing types and their maximum densities are: random close packing, 0.6400[43]; the tetrahedral lattice, $\frac{\pi\sqrt{3}}{16} \approx 0.3041$[44]; and the loosest possible density that has been found is 0.0555 in the Heesch and Laves loose-packing structure[45].

Spherical packing was first analysed around 1587, when the question was posed about whether or not there was a method to quickly determine the number of cannonballs in a square pyramidal stack, which is known as the cannonball problem.[46]

2.3.1.3 Packing of Multiple Sized Particle Systems

Most of the research into systems where there are multiple sizes of particle present investigate binary systems, i.e. those with two distinct particle sizes present. In a 2D square packing system, it has been found that up until a radius ratio of 0.41:1[47], the system packs densely by filling in the voids created by the larger particles with the smaller particles. However, after this point, due to the sizes of the particles being more similar, the system rearranges into a different structure in order to maintain their density[47]. Additionally, if the radius ratio is above 0.742:1, the binary system is no longer able to pack better than a system with same sized particles.[48] The binary system with the highest possible packing fraction is with a particle ratio of 0.1:1, having a packing fraction of 0.9624.[47]

Descartes circle theorem, shown in Equation 2.1, can be used to determine the radius of the particle that would fit perfectly between three particles, so that all four of them would share an edge with all of the others, as shown in Figure 2.4.

$$k_4 = k_1 + k_2 + k_3 \pm 2\sqrt{k_1k_2 + k_2k_3 + k_3k_1} \tag{2.1}$$

where k is the curvature, 1/radius, of each of the circles 1 to 4. The two solutions to this theorem, through the ±, are due to the possibility of a large circle encompassing the three present circles, as well as a smaller one present between them. When the radii of the three present circles are the same, the ratio of their radii to the radius of the circle in between them is 0.1547.[49]

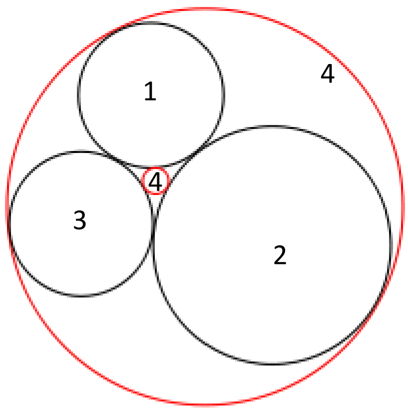


Figure 2.4: Example of tangent circles with the black circles being present particles, and red circles being possible solutions through Descartes theorem

Binary systems have also been investigated in 3D, where it has been found that if the radius ratio is 0.299099:1 or lower, then it is always possible for smaller spheres to pack inside the interstices between the larger spheres[50]. When the radius ratio exceeds 0.4142:1[51], the smaller spheres are no longer able to pack inside even the octahedral voids within the structure, meaning that above this ratio, the structure either needs to expand to allow the more similarly sized smaller particles to fill the voids inside, which decreases the overall density, or it would rearrange into a more complex structure[51, 52].

The packing fractions of binary systems of a radius ratio either side of the perfect 0.4142 value have been investigated. The packing fractions of the systems below this value were generally greater than the systems with a ratio above that value, except when comparing the minimum and maximum of some of the systems ranges. The systems with a radius ratio lower than the perfect value having packing fractions of around 0.8 and the systems with a radius ratio higher than the perfect value having packing fractions around 0.75. All of the systems packing fractions with a ratio below the value, except one, decrease when the ratio increases. The only system that increases alongside an increase in radius ratio is an orthorhombic lattice system with six small particles for each large particle. When the radius ratio is above the perfect value, of the five systems investigated, two showed a decrease in packing fraction, two showed an increase in packing fraction, and one showed no change in packing fraction.[53]

2.3.2 Percolation

Percolation theory is the study of percolation, which is generally used to investigate how fluids flow through porous materials through the connectivity of the pores. The theory describes how a network is affected when nodes or links between them are added or taken away. It was first elucidated in the Flory-Stockmayer Theory[54], which governs the point at which a gel forms from a system of polymers[55]. This point is generally known as the percolation threshold, which is the critical value for a system where below it a giant connected component does not exist, and above it one does exist. However, in this work we are instead looking at the connectivity of particles within the system, describing a percolation chain as a chain of particles connecting edges of the box, instead of a group of pores for fluid to pass through.

There are two different “models” for investigating percolation: bond percolation and site percolation. Bond percolation uses the frequency of the presence of bonds between nodes to determine if a percolated structure is present. Site percolation determines whether or not a site is open and connected open sites form percolated structures. To visualise this using a grid of squares, bond percolation works using the connections between the vertices, whereas site percolation connects whole squares of the grid that are “open”, as can be seen in Figure 2.5.[56]

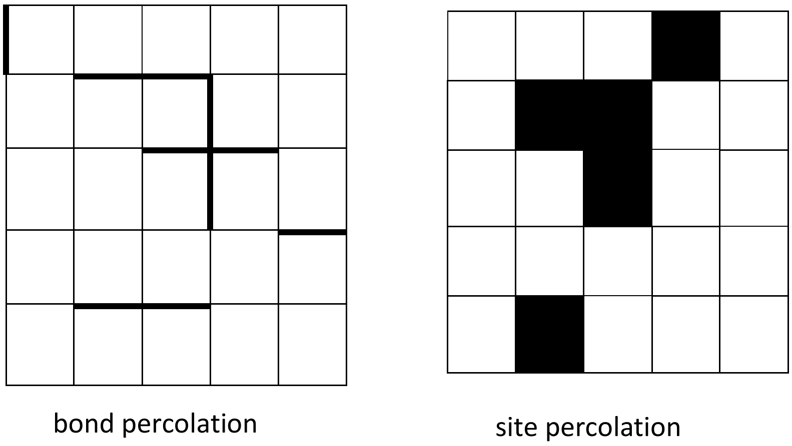


Figure 2.5: Representation of Bond and Site Percolation (redrawn from [57])

Generally, systems of infinite size are examined, so a percolated structure would be a connected cluster of infinite length. Kolmogorov’s zero-one law states that, ‘the probability of an infinite cluster existing is either zero or one, for any given probability of a site being open or closed’[58]. In our systems, the chance of a percolation structure occurring is based on the probability of large particle to large particle contacts forming. This allows us to investigate the critical value of the site probability to determine the critical probability, known as the percolation threshold, at which the cluster forms.

Previously exact and approximate values of percolation thresholds for different lattices have been successfully calculated. It has been found that regular triangular lattices, shown in Figure 2.6, have a site percolation threshold of 0.5[59], which is the type of lattice that a fully regular system of circle packing could be likened to, with the centre of each circle a point on the lattice. In 3D, the site percolation threshold for FCC lattices has been calculated to be 0.1998 ± 0.0006 and the bond percolation threshold is 0.1198 ± 0.0003[60]. This shows that a percolated structure is much more likely to occur in 3D, as the percolation threshold in 3D systems is much lower than in 2D systems.

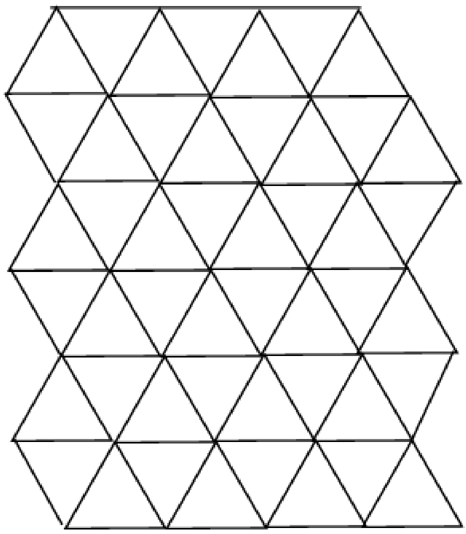


Figure 2.6: Triangular Lattice

The investigation of percolation theory can give insights into multiple different disciplines, including studying the flow of traffic through cities when certain roads are available, or not, to determine bottlenecks[61], as well as in ecology to study environmental fragmentation effects on habitats[62]. Our interest lies in its uses, investigating how contacts within beds of particles might break when the bed system is altered, through the presence of differing particle sizes and other factors.

2.3.3 Finite Size Effect

Models also need to deal with the finite size effect[63], which is where, potentially due to the small size of a system, the edges of the system can skew the data. Using one of our produced systems as an example, the packing of the particles at the edges of the box is distinctly different to the packing in the centre, with denser packing in the centre of the bed. In a small enough system, this difference would affect the average data values and therefore give an incorrect outcome, compared to a larger system where the edge values are not as big a part of the system as a whole.

Models can either be run with increasingly large systems to determine at which size there is no longer a skewing effect from edge cases, and the data used to extrapolate to an infinite system, or smaller systems run and the data from the edge cases discarded. Both strategies have their pros and cons, with larger systems taking longer and more computational power to complete however being able to lessen or remove the finite size effect, and smaller systems being easier to produce however still retaining the finite size effect as well as the potential of producing skewed results when looking at averages of a system as a large fraction of it has been removed.

2.3.4 Angle of Repose

The angle of repose is defined as the steepest angle of descent, relative to a horizontal surface, that a material can be piled without slumping.[64]

There are various methods for determining the angle of repose for a material. The simplest is using the following equation if the coefficient of static friction is known for the material,

$$\tan(\theta) \approx \mu_s \tag{2.2}$$

where μ_s is the coefficient of static friction and θ is the angle of repose.[65]

The other methods are experimentally based and are each suitable for a different type of material. The tilting box method is suitable for fine-grained materials with a grain size of 10mm or less and allows the coefficient of static friction to be determined for a material, from which the previous equation can be used to determine the angle of repose. This method works by filling a box with the granular material, and then tilting it gradually until the material begins to slide, as depicted in Figure 2.7.[66]

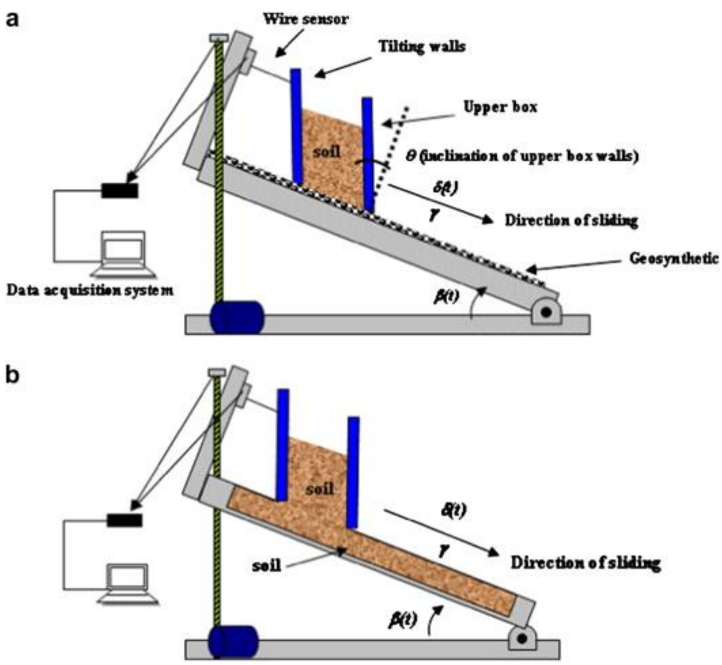


Figure 2.7: Depicting the Tilting Box Method
(with permission from [66])

Another method is the fixed funnel method, where the material is poured through a funnel to allow it to form a cone shape on a base. Once the cone has reached either a set height or a set width, the angle of repose can then be calculated using the following equation,

$$\theta = \tan^{-1}(\frac{2h}{b}) \quad (2.3)$$

where θ is the angle of repose, h is the height of the cone, and b is the width of the base of the cone.[66]

The angle of repose is useful to investigate as it has links to the flowability of granular materials, which helps when designing processing equipment and storage for particulate solids as it ensures the stability of the material and reliable flow from hoppers.[67]

Another use of the angle of repose is to allow calibration of models for when a specific material is being simulated, the angle of repose of the system created in the model can be compared to the expected value to determine if the model is running accurately.[68] However, as the model designed for this thesis does not consider frictional forces, only gravity, it is likely that this method will not be applicable to my model.

2.3.5 Bed Fragmentation

With the help of two final year MEng project students, we were also looking to investigate the effect of a shear force on the contacts between the particles within the modelled systems, discussed further in section 3.1.1.12. This allows us to see how they break apart, and so we can potentially compare different system parameters, such as particle size distribution, with how clusters can form within the systems.

Agglomerates are generally defined by their size and strength, with the strength being defined by how difficult they are to break apart. Different factors have been investigated for their effects on agglomerate properties, including particle size and shape distribution, the solvents used during processing, and agitation during filtration.[69, 70] Higher agitation does result in smaller agglomerates, however also runs the risk of particle breakage, so often it is a balancing act between reducing particle size to an acceptable level, without breaking them down too far.[71]

Investigation has also gone into the factors that affect the breakage of agglomerated systems in industries other than pharmaceutical, such as within the food industry, where DEM has been used to investigate how cereal grains break apart under impact within an agitated system.[72, 73]

There is no unified way of reporting agglomerate properties, although some methods are becoming more consistent, such as the agglomerate brittleness index which describes the strength of an agglomerate.[74] We anticipate that with the basis of the model complete, it can be used to add some more knowledge to how agglomerates act under various stresses and circumstances to greater aid the industries that need to find a solution to this issue.

2.3.6 Non-Circular / Non-Spherical Particles

Whilst spherical particles are a good starting point for simulating packed beds, being able to simulate specific particle shapes and sizes is incredibly useful. Whilst some modelling algorithms are capable of simulating non-spherical shapes and sizes, to accurately simulate a specific substance, many more parameters describing its particles are required beyond its shape and size, all of which massively increases the computational time and power required. This is briefly discussed further in Chapter 7.

Therefore, when simulating non-spherical particles, it is often done using spherical particles as a starting point[75], forming them into chains as illustrated in Figure 2.8. This allows simulation of more sophisticated systems without driving up the computational power required as far. The use of the circular particles as building blocks instead of swapping to polygons was to allow the new algorithm to build off the old one, thus decreasing the workload.

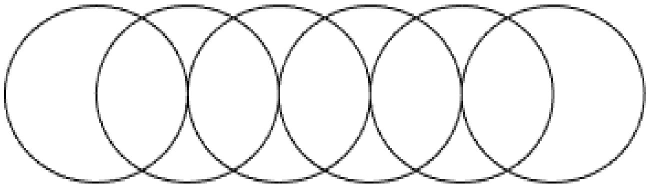


Figure 2.8: Example of a chain made up of circular particles.

Generalisations about system properties, like packing fractions, cannot be made across all systems of non-circular particles due to the wide range of particle shapes and sizes possible, only being somewhat possible when examining systems made up of a specific set of particles, due to the large variance that can now occur, even when only accounting for shape and size. Therefore, I will specifically be investigating chains made up of spherical particles.

Research has gone into the packing on non-spherical 3D shapes, with the two shapes closest to my research being cylinders and spherocylinders: spherocylinders being cylinders with rounded ends. The maximum packing fraction found for both cylinders and spherocylinders is ~0.9069[76], dependant on the ratio between the diameter and height of the shape. Figure 2.9 shows the effect on changing the ratio of the diameter and height of cylinders and spherocylinders was found to have on the packing fraction of their systems using a relaxation algorithm. This algorithm works by filling the system with randomly placed shapes, i.e. cylinders, with large overlaps. When the system then iterates, the particles move away from each other, lessening the overlaps, and the system size increases. The algorithm ends once the total overlap has become lower than a predefined value.[77]

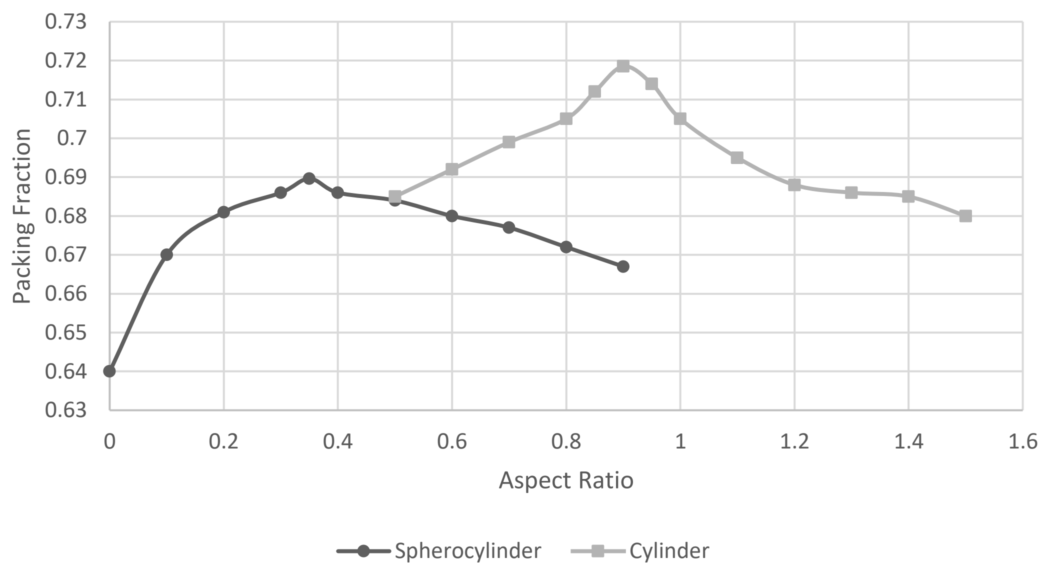


Figure 2.9: Packing fractions of systems of cylinders or spherocylinders at different aspect ratios of height/diameter (redrawn from [76])

Both the cylinder and spherocylinders have a peak packing fraction, with the cylinder graph having a more defined peak. The cylinder systems are also shown to have higher packing fractions across the range of aspect ratios simulated.

The packing of these shapes under gravity has also been investigated using DEM[78], where the effect of different parameters within the DEM software on the packing fraction of a system were analysed and compared to experimental data. In contrast to Figure 2.9, this research found that when increasing the aspect ratio, it often resulted in the packing fraction plateauing instead of showing a consistent decrease, which is likely due to the different addition methods and the complexity of the simulation methods used.

2.4 References

(1) *United Nations Population Division*. https://www.un.org/development/desa/pd/ (accessed 2020).

(2) *Global Pharmaceutical Industry - Statistics & Facts*, https://www.statista.com/topics/1764/global-pharmaceutical-industry/ (accessed 2020).

(3) Shepherd, M., and Shepherd, E., Medicines administration 2: procedure for administration of oral medicines. Nursing Times [online], 2020, 116(7), 42-44.

(4) Pietsch, W. (2008) *Agglomeration in industry: Occurrence and applications*, *Agglomeration in Industry: Occurrence and Applications*; Wiley, **2008**.

(5) Mehos, G. and C. Kozicki, "Consider Wet Agglomeration to Improve Powder Flow", *Chem. Eng.*, **2011**, *121*, 46-49.

(6) Pietsch, W., *Agglomeration Processes: Phenomena, Technologies, Equipment;* Wiley, **2002**.

(7) Fernández-González, D., Ruiz-Bustinza, I., Mochón, J., González-Gasca, C. and Verdeja, L. F., Iron Ore Sintering: Process, *Mineral Processing and Extractive Metallurgy Review*, 2017, 38(4), 215–227.

(8) *Tablet Presses: Tablet Binders*. https://www.lfatabletpresses.com/articles/tablet-binders (accessed 2020).

(9) am Ende, D., Birch, M., Brenek, S. J., and Maloney, M. T., Development and Application of Laboratory Tools To Predict Particle Properties upon Scale-Up in Agitated Filter-Dryers, *Org. Process Res. Dev.*, **2013**, *17* (10), 1345-1358.

(10) *PharmTech: Analyzing Content Uniformity*, https://www.pharmtech.com/view/analyzing-content-uniformity (accessed 2023).

(11) Nessel, R. J., Apelian, H. M. & Blodinger, J., Uniformity of Distribution of Cyanocobalamin in Tablet Formulations, *J. Pharm. Sci.*, **1970**, *59*, 254-257.

(12) Langenbucher, F., Statistical Analysis of the USP 18 Content Uniformity sampling plan for tablets, *Pharm. Acta Helv.*, **1972**, *47*, 142-151.

(13) Pederson, A. O. & Torud, Y., Statistical characteristics of the USP content uniformity test, *Pharm. Acta Helv*., **1971**, *46*, 114-120.

(14) Pederson, A. O., Torud, Y. & Waaler, T. J., Influence of the analytical error on the validity of the content uniformity test USP XVII, *Pharm. Acta Helv.*, **1971**, *46*, 21-30.

(15) Orr, N. A. and Sallam E. A., Content uniformity of potent drugs in tablets. *J. Pharm. Pharmacol.*, **1978**, *30*, 741-747.

(16) Saunders, R., The Effect of Particle Agglomeration in Pharmaceutical Preparations. *The Statistician*, **1991**, *40,* 77-86.

(17) Heisel, S., Ernst, J., Emshoff, A., Schembecker, G., and Wohlgemuth, K., Shape-independent particle classification for discrimination of single crystals and agglomerates. *Powder Technol.*, **2019**, *345*, 425-437.

(18) *The Benefits of Continuous Manufacturing*. https://www.pharmaceuticalprocessingworld.com/the-benefits-of-continuous-manufacturing/ (accessed 2020).

(19) Warman, M., (2010) *Chemical Engineering in the Pharmaceutical Industry: R&D to Manufacturing*; John Wiley & Sons, **2010.**

(20) Auger, P. and Bougarel, C., Models in Science, *Diogenes,* **1965,** *13*(52), 1-13.

(21) *Encyclopaedia Britannica: Scientific Modelling*. https://www.britannica.com/science/scientific-modeling (accessed 2020).

(22) Oberkampf, W. L., DeLand, S. M., Rutherford, B. M., Diegert, K. V. and Alvin, K. F., Error and uncertainty in modelling and simulation, *Reliability Engineering and System Safety*, **2002,** *75*(3), 333–357.

(23) Roy, K., Kar, S. and Das, R. N., *Understanding the Basics of QSAR for Applications in Pharmaceutical Sciences and Risk Assessment*, Academic Press, **2015.**

(24) Durrant, J. D. and McCammon, J. A., Molecular dynamics simulations and drug discovery, *BMC Biology*, **2011,** *9*, 71.

(25) Burman, B. C., Cundall, P. A. and Strack, O. D. L., A discrete numerical model for granular assemblies, *Geotechnique*, **1980,** *30*(3), 331–336.

(26) Munjiza, A., Bangash, T. and John, N. W. M., The combined finite-discrete element method for structural failure and collapse, *Engineering Fracture Mechanics*, **2004,** *71*(4–6), 469–483.

(27) *EDEM Research Spotlight: CFD-DEM Simulations of a Liquid-Solid Fluidized Bed with ERT Validation*. https://altairuniversity.com/51138-edem-research-spotlight-cfd-dem-simulations-of-a-liquid-solid-fluidized-bed-with-ert-validation/ (accessed 2021).

(28) Thakur, S. C., Ooi, J. Y. and Ahmadian, H., Scaling of discrete element model parameters for cohesionless and cohesive solid, *Powder Technology*, **2016,** *293*, 130–137.

(29) *Altair EDEM Home Page*. https://www.altair.com/edem/ (accessed 2021).

(30) *EDEM Research Spotlight: DEM modelling of agglomeration of potato starch. https://altairuniversity.com/51109-edem-research-spotlight-dem-modelling-of-agglomeration-of-potato-starch/ (accessed 2021)*.

(31) Marigo, M., Discrete element method modelling of complex granular motion in mixing vessels: evaluation and validation. Ph.D. Thesis, University of Birmingham, Birmingham, U.K., **2012.**

(32) Stein Sören, S., and Jürgen, T., Simulation of a Filtration Process by DEM and CFD, *IJMEM,* **2012**, *1* (1), 28-35.

(33) Abdallah, A., Vincens, E., Magoariec, H., and Picault, C., DEM filtration modelling for granular materials: Comparative analysis of dry and wet approaches, *Int J Numer Anal Methods,* **2024,** *48*, 870-886.

(34) Abdallah, A., Vincens, E., and Magoariec, H., Dry and wet filtration in granular filters: a DEM modeling, *25th French Mechanics Congress*, **2022.**

(35) Qian, F., Huang, N., Lu, J., and Han, Y., CFD-DEM simulation of the filtration performance for fibrous media based on the mimic structure, *Computers and Chemical Engineering,* **2014**, *71*, 478-488.

(36) Liu, X., Ding, X., Chen, C., An, R., Guo, W., Zhang, W., Nan, H., and Wang, Y., Investigating the filtration behaviour of metal felt using CFD-DEM simulation, *Eng. Appl. Comput. Fluid Mech.,* **2019**, *13*, 426-437.

(37) Tamrakar, A., Zheng, A., Piccione, P. M. and Ramachandran, R., (2020) 'Investigating particle-level dynamics to understand bulk behavior in a lab-scale Agitated Filter Dryer (AFD) using Discrete Element Method (DEM)', *Advanced Powder Technology*, **2020,** *31*(1), 477–492.

(38) Zhang, G. and Torquato, S., Precise algorithm to generate random sequential addition of hard hyperspheres at saturation. *Phys. Rev. E*, **2013**, *88*(5), 053312.

(39) Chang, H. and Wang, L., A Simple Proof of Thu''s Theorem on Circle Packing, **2010**. arXiv:1009.4322

(40) Gauss, C.F., Besprechung des Buchs von L.A. Seeber: Untersuchungen über die Eigenschaften der38ositiven ternären quadratischen Formen usw., *Göttingische Gelehrte Anzeigen*, **1831**, *2*, 188–196.

(41) Schoenberg, I.J. and Steinhaus, H. Mathematical Snapshots, *Coll. Math. J.*, **1986**.

(42) Chen LT., Chen, CY., and Chen, HL., FCC or HCP: The stable close-packed lattice of crystallographically equivalent spherical micelles in block copolymer/homopolymer blend, *Polymer*, **2019**, *169*, 131-137.

(43) Jaeger, H.M. and Nagel, S.R. 'Physics of Granular State', Science, **1992**, *20*, 1523–1531.

(44) Hilbert, D. and Cohn-Vossen, S. 'Geometry and the Imagination', *Phys. Today*, **1953,** *6*(5), 19.

(45) Gardner, M. 'New Mathematical Diversions From Scientific American', **1966,** 88.

(46) *David Darling: Cannonball Problem*. https://www.daviddarling.info/encyclopedia/C/Cannonball_Problem.html (accessed 2021).

(47) Bédaride N. and Fernique, T., Density of Binary Disc Packings: The Nine Compact Packings. *Discrete and Computational Geometry,* **2022**, *67*, 787-810.

(48) Heppes, A., Some Densest Two-Size Disc Packings in the Plane, *Discrete and Computational Geometry*, **2003,** *30*(2), 241–262.

(49) Beecroft, P., Properties of Circles in Mutual Contact, *Lady's and gentleman's diary*, **1843.**

(50) Zong, C., From deep holes to free planes, *Bulletin of the American Mathematical Society*, **2002,** *39*(4), 533–555.

(51) O'Toole, P. and Hudson, T., New High-Density Packings of Similarly Sized Binary Spheres. *J. Phys. Chem. C*, **2011**, *115*(39), 19037-19040.

(52) Marshall, G. W. and Hudson, T. S., Dense binary sphere packings, *Beitrage zur Algebra und Geometrie*, **2010,** *51*(2), 337–344.

(53) Hopkins, A., and Stillinger, F., Densest binary sphere packings, *Phys. Rev. E.,* **2012**, *85*, 021130.

(54) Sahimi, M., *Applications of percolation theory*; London: Taylor & Francis. **1994.**

(55) Flory, P. J., Molecular size distribution in three dimensional polymers. I. Gelation, *Journal of the American Chemical Society*, **1941,** *63*(11), 3081.

(56) Chayes, L. and Schonmann, R.H., Mixed percolation as a bridge between site and bond percolation, *Annals of Applied Probability*, **2000,** *10*(4), 1182–1196.

(57) *Wolfram MathWold: Bond Percolation*. https://mathworld.wolfram.com/BondPercolation.html (accessed 2021).

(58) Mcnamara, D., Kolmogorov's zero-one law with applications, **2017,** 1–15.

(59) Stauffer, D. and Aharony, A., *Introduction to Percolation Theory 2nd Edition;* Taylor & Francis Group, **2014**.

(60) Gaunt, D. S. and Sykes, M. F., Series study of random percolation in three dimensions, *Journal of Physics A: General Physics*, **1893,** ***1**6*(4), 783–799.

(61) Li, D., Fu, B., Wang, Y. and Havlin, S., Percolation transition in dynamical traffic network with evolving critical bottlenecks, *Proceedings of the National Academy of Sciences of the United States of America*, **2015,** *112*(3), 669–672.

(62) Boswell, G. P., Britton, N. F. and Franks, N. R., Habitat fragmentation, percolation theory and the conservation of a keystone species, *Proceedings of the Royal Society B: Biological Sciences*, **1998,** *265*(1409), 1921–1925.

(63) Wedekind, J., Reguera, D. and Strey, R., Finite-size effects in simulations of nucleation. *J. Chem. Phys*, **2006**, *125*(21), 214505.

(64) Mehta, A. and Barker, G.C., The dynamics of sand, *Reports on Progress in Physics*, **1994**, *57*(4), 383–416.

(65) Nichols, E.L. and Franklin, W.S., *The Elements of Physics. A College Text-Book*; New York: The Macmillan company, 1896

(66) Pitanga, H., N., Gourc, JP. and Vilar, O. M., Interface shear strength of geosynthetics: Evaluation and analysis of inclined plane tests, *Geotextiles and Geomembranes*, **2009**, *27*(6), 435–446.

(67) *Powder Process: Angle of Repose Overview.* https://powderprocess.net/Powder_Flow/Angle_Repose.html (accessed 2021).

(68) Coetzee, C.J., Review: Calibration of the discrete element method, *Powder Technology*, **2017,** *310*, 104–142.

(69) Birch, M. and Marziano, I., Understanding and Avoidance of Agglomeration During Drying processes: a case study, *Org. Process Res. Dev.*, 2013, *17*, 1359−1366.

(70) *Feeco International: Controlling particle size through agglomeration.* https://feeco.com/controlling-particle-size-through-agglomeration/ (accessed 2022).

(71) Eggersdorfer M. L., Kadau, D., Herrmann, H. J. and Pratsinis, S. E., Fragmentation and restructuring of soft-agglomerates under shear, *Journal of Colloid and Interface Science*, **2010**, *342*, 261–268.

(72) van Wachem, B., Thalberg, K., Nguyen, D., de Juan, L. M., Remmelgas, J. and Ingela Niklasson-Bjorn, I., Analysis, modelling and

simulation of the fragmentation of agglomerates, *Chemical Engineering Science,* **2020**, *227*, 115944.

(73) Zeng, Y., Jia, F., Xiao, Y., Han, Y. and Meng, X., Discrete element method modelling of impact breakage of ellipsoidal agglomerate, *Powder Technology,* **2019**, *346*, 57-69.

(74) Ottoboni, S., Simurda, M., Wilson, S., Irvine, A., Ramsay, F. and Price, C. J., Understanding effect of filtration and washing on dried product: Paracetamol case study, *Powder Technology,* **2020**, *366*, 305-323.

(75) Nolan, G. T. and Kavanagh, P. E., Random packing of nonspherical spheres, *Power Technology*, **1995**, *84*(3), 199-205.

(76) ShuiXiang, L., Zhao, J., Lu, P., and Xie, Y., Maximum packing densities of basic 3D objects, *Chinese Science Bulletin*, **2010**, *55*(2), 114-119.

(77) Li, S., and Zhao, J., Sphere assembly model and relaxation algorithm for packing of non-spherical particles, *Chinese Journal of Computational Physics,* **2009,** *26*(3), 454-460.

(78) Tangri, H., Guo, Y., and Curtis, J., Packing of cylindrical particles: DEM simulations and experimental measurements, *Powder Technology,* **2017**, *317*, 72-82.

3 Methodology

The aim of this work is to produce a model that can produce realistic representations of packed beds of particles formed under gravity that can be used to investigate agglomeration during isolation. This started with a 2D algorithm as it was initially easily produced, before upgrading it to a 3D algorithm.

A new *ab initio* model was developed instead of using pre-existing software as it allows targeting the model towards specific issues that we want to investigate, as whilst current modelling software can achieve many goals, due to their complexity they can take up much more computational time and power than would be needed to solve a single problem within these systems.

The following chapter discusses the stages that the 2D and 3D algorithms went through from initial conception to their current states. It also discusses the other functionalities that the model is able to perform, and the specifics of the experiments performed for data collection. The creation of these algorithms continued all the way up to the end of my project, with the 2D algorithm being finalised after roughly two years, the 3D algorithm finalised after the third year, and the chain particle algorithm being developed in the last year. The algorithm was created using the ForTran programming language, learnt using online resources and literature.[1,2]

3.1 2D Algorithm

3.1.1 Timeline of Model

This section looks through the stages that the 2D algorithm went through and the reasons behind each of the changes, from its initial setup to the algorithm used to produce data discussed in the later sections.

3.1.1.1 Initial Program

The first step taken in producing the model was to create a box such that the boundaries could be edited and it could have particles, made up of covered points within the system, placed inside it. Initially this was done by creating a 2D array, allowing each co-ordinate to have an individual value. The values used were a 0, denoting that its location was empty, or a 1, showing that a particle was present at that location. The array could then be printed, showing the particles using a grid of 0s and 1s. The size of the box could be changed by simply editing the *x* and *y* ranges of the array.

Once the box was set up, the particles could be added. This was done by randomly generating an *x* value, then a loop was initiated, with the starting value being the maximum *y* value present in the array, decreasing towards 0 in intervals of -1. The loop repeated until either the particle reached the bottom of the box, in which case the particle would be placed there, or the falling particle impacted a previously placed particle. Impacts were determined by looking at the array and determining if the falling particle overlapped any co-ordinates in the array containing a value of 1. When an impact occurred, a series of 'if, then, else' statements were run that determined where the falling particle had impacted and how to react, until a suitable position was found for the particle to be placed. For example, if the falling particle had been impacted (i.e. encountered an existing particle in the bed) on its left side, then it would “slide” down to the right to find a stable resting place. An example of the structure formed in this algorithm is shown in Figure 3.1.

Due to the high specificity of the impact determining statements, this method worked for the situations that were found. It is very likely, however, that many interactions were not accounted for, due to the large number of ways two particles, even with set sizes and shapes, can impact. This method was also specific to the shape and size of the particle, as the calculations were made using exact distances, therefore it would not be very useful going forward as if we wished to model a different size or shape of particle, the whole algorithm would need to be rewritten. This was important as the more varied the particles the model can account for, the more useful it will be. Even with the minimal number of interactions actually accounted for, the model was slow as it had to check through each interaction before finding the relevant one. Finding all of the possible interactions for the specific particle shape and size would have taken an inordinately long time and not been useful, therefore, a new method was investigated. In Figure 3.1, the particles added are displayed as integer values from 1 to 9, with the gaps between them shown as 0s.

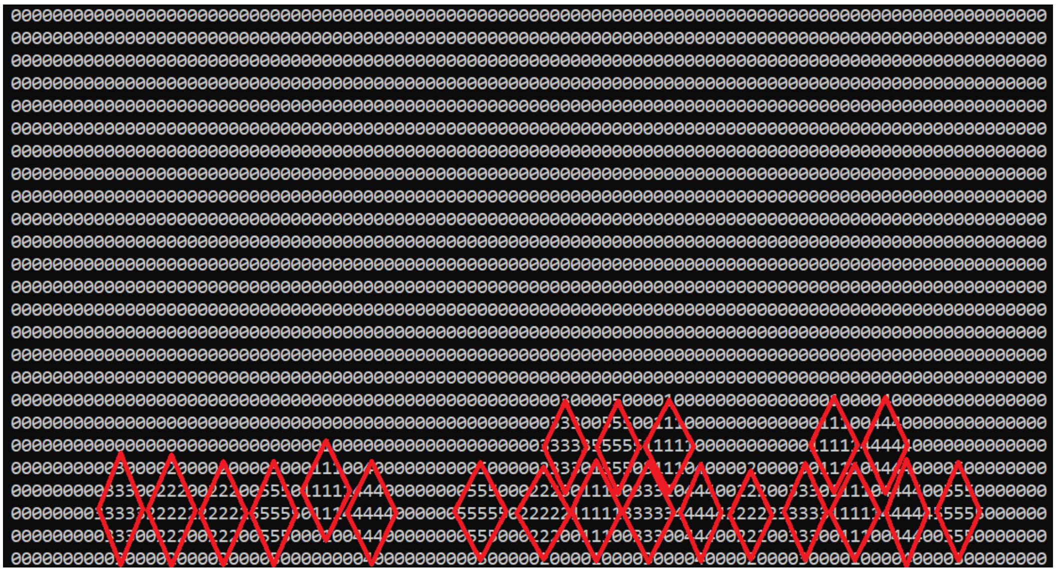


Figure 3.1: An example of a system created by the early 2D algorithm, with the diamond particle shapes, denoted by non-0 values, highlighted

3.1.1.2 Implementation of the Contour Plot Method

In the previous model, the particles were a diamond shape as it was the easiest to draw and stack without calculations. However, in this revised model the particles were altered to be circular. By inputting a radius, the user was able to choose the size of the particles at the beginning of each run. The input of the particle into the array could also be done more simply by using the radius and Pythagoras theorem.

The new particle addition method not being specific to shape or size, rather being variable, made this possible. The size of the overall box was also now based upon the particle size that was entered, and when multiple particle sizes were added, the largest radius was used to calculate the box size. This ensures that a sensible number of particles are able to be added to a system.

In addition to the original array, which showed where the particles were placed, and the space they took up in the same way as the previous model, there was a second array which showed a ‘contour plot’ of the grid. This contour-plot showed the particle bed with no distinction between individual particles, and with a line across the edge of the current particle bed showing the closest “safest spots” that a particle could be added on top of the particle bed. This allowed the lowest point along this line to be identified for the new particle to be placed, within certain bounds of the impact point. The model now also outputted the grid to a text file in a format that could be read by MATLAB; this allowed for improved presentation, as the command line output was difficult to observe for long periods of time.

A new particle would be inputted into the system at a random *x* coordinate at the top of the system and then fall until it impacted with a previously placed particle. At this point, the above mentioned contour plot would be created to determine the nearby low point for the particle to roll to from its impact point.

This model was a considerable step up from the initial model, however it still needed to search through the grid to find the contour plot points, at this point the grid was relatively small, as it would need to be scaled up later this method would also become less and less feasible the larger the grid became.

3.1.1.3 Removing the Grid

To address this, the ‘real’ grid, that denoted each spot with ‘1's and ‘0's, was removed and replaced with arrays that saved the centre points and radii of the particles. For the initial version of this model the contour plot was removed, and instead distance calculations were made between the centres of the current particles and the proposed centre of the new particle. By checking the distance against the combined radius of the particles, it could be determined if they were too close to each other (i.e. overlapping) or not, and so whether the particle could be placed there. However, due to this being the only check present, the particles filled up the grid leaving some unwanted gaps between them, as the only check was whether or not the particles were overlapping, with no preferences for realistic stacking.

The model now had to change how it output the particles, as the previous output, the array, no longer existed to be printed. Instead, the centre coordinates and radii of each particle was output to a file, and the code was written to allow MatLab to take the output and visualise them.

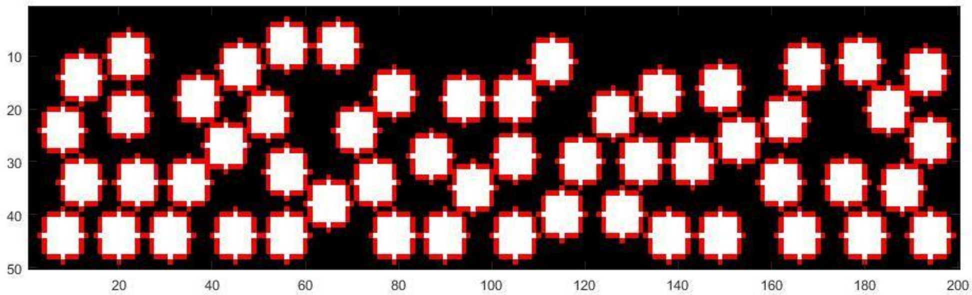


Figure 3.2: A system created by the 2D algorithm where particles are only placed if they contact another particle

3.1.1.4 Sections and Variable Grid

Now that the model was running faster due to the removal of the visual array, the contour plot was reintroduced alongside the new distance calculations, both making sure that the particles were not overlapping and were in more realistic stable positions. However, in order that the contour plot did not have to be created for the whole grid, the large grid was split into 5x5 sections; when a particle's centre point was saved, it was also noted in which section it was placed. Therefore, when a new particle was added, a local contour plot was created of that section, allowing the model to finalise the placement based upon the initial distance calculation. Initially these sections, and the grid itself, had set sizes due to it being simpler at the time. However now that different particle sizes were able to be added, which is discussed further below, the smaller sections, and the grid as a whole, needed to be able to accommodate this variation. Therefore, the sections and grid were changed to have a variable size, dependant on the largest particle radius that had been entered, thus ensuring that both the grid and the contour plot would be able to handle the size of the particles, ensuring a sensible number of particles were present in each grid section. The reintroduction of the contour plot along with the changes to how the particle data was saved, greatly increased the speed at which the model ran as well as improving its accuracy.

3.1.1.5 Top-Down Filling

All iterations of the model, after the first, simulated a particle falling into a box. This worked by looping the particle's location from the bottom to the top of the box, so that the lower points would be found first. However, when working with particles of different sizes, this resulted in smaller particles being placed in gaps between larger particles that should no longer be accessible. At first a method of trying to determine if a space would be underneath another particle was investigated, however this greatly increased the runtime of the model, as well as not always functioning correctly. Therefore, the model was changed so that instead of searching upwards for the first available space, the particles are lowered into the box until they impacted something else. The contour plot would then be used to refine the final position after impacting with the bed, in the same method as in the previous model algorithms.

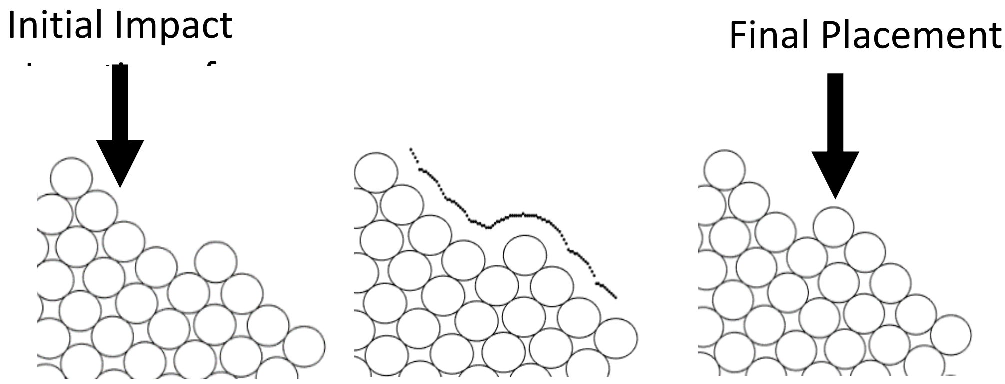


Figure 3.3: The stages a particle goes through when being added to the system. a) Falling at a random *x*-coordinate until impacting the bed. b) The contour plot being formed to show possible points of rest. c) The particle moving down the contour plot to a low point.

3.1.1.6 Score Based Positioning

Previously, the particle would simply pick the lowest point along the contour plot that was within a certain distance of the initial impact point. However, this was very basic and it did not result in particles being placed in inappropriate positions. As shown in Figure 3.4, the closest low point, (b), to the impact point, (a), would not be the correct final position as it would instead roll down the slope to the right and rest at (c).

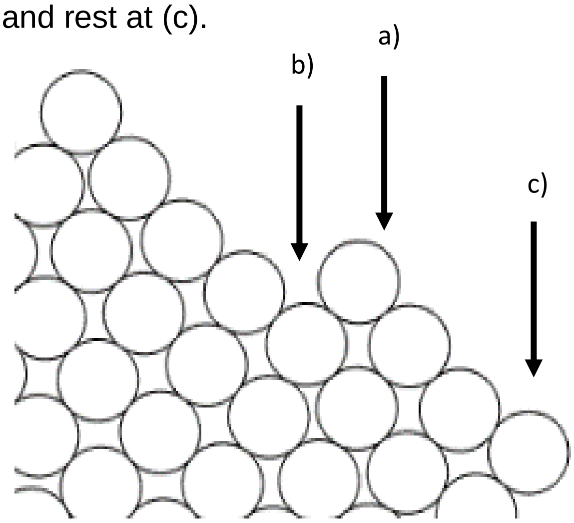


Figure 3.4: An example of a possible incorrect position. a) The point of impact. b) The closest low point. c) The correct resting point.

Therefore, once the contour plot was created, each valid position along it was given a score based upon how close it was to the initial impact point, and how high up in the box it was. Preference was given to being close to the initial impact site and being lower down in the box. Each point was then ordered based upon its score, and the model then looped through them from best to worst until a point passed the final checks. This helped with the realism issue, however it did not fix it completely. Therefore, the score system was implemented in a way that when more criteria were conceived to make the point selection more accurate, they could easily be added to the model.

3.1.1.7 Set Allocatable Array

Once the functionality of being able to add particles of a chosen size was added, the arrays used in the model were made so that their size was allocatable at the beginning of the model, allowing them to be changed to fit the size of the particles being added. The contour plot array had its size changed each time it was created as depending on the location of the particle it might need to search adjacent sections of the box.

Due to the array having to be reset each time a new particle was added, it occasionally caused the model to crash. Even after attempting a debugging of this issue, it still is not fully understood why these crashes occurred, only that they were caused by the re-allocating of the array. Owing to this, from here on in, the contour plot array was set to have the largest size it would need, instead of having its size changed each time it was used.

3.1.1.8 Increase Position Criteria

To increase the accuracy of the selection of the final position of the new particle, a check was added to ensure that the new particle was resting on top of two others. This was first done by confirming that there were two particles close to the new location being tested, and then ensuring the centre point of the new particle would rest in between, above and in contact with them.

The other change was made to ensure that this would work from the beginning of a run was that the model now started with the box having a base layer of particles already present instead of being empty, as a particle landing on the bottom of the box would have failed this check.

3.1.1.9 Sliding downhill instead of jumping to final position

As the score-based positioning was still occasionally resulting in an inaccurate placement for the particles, as sometimes it would jump over particles to get to its resting place shown in Figure 3.4, the way the particles interact with the contour plot was changed. A simpler approach of having the particle jump to the nearest contour plot point to its impact location, and then looking at the height of the contour plot points to either side of it. It then moved to the point that was lower than it, if they are the same height then it moves to a random side, which then repeats until both points either side of the resting point are higher than it. This method means that particles now accurately slide as if under gravity to their final destination without jumping over particles they would normally be stopped by. Note that the algorithm does not consider conservation of momentum or frictional forces explicitly, but instead the model assumes a gentle settling of the particles with motions dampened by the solvent.

3.1.1.10 Real Final Values

Currently, whilst the particles are not placed on a grid as in the method initially used, the outputs are still based upon a coordinate system. This resulted in the MatLab output having some gaps between particles due to the coordinates being integers and drawing circles within the grid. The next step was to edit the model so that once the integer positions for each particle have been found, the integer values are edited to real values to eliminate the unphysical gaps appearing in the visualisation.

This was done by removing the rounding from the calculations and allowing the values to be saved as real variables. This required a change in how the values were then referenced as the real variables were not able to be used as coordinates for the arrays, so instead of referencing the particle number by location, the location was now referenced by the particle number.

3.1.1.11 Final Optimisations

It was at this point that the 2D algorithm was deemed to be working, however a few more optimisations were added so that it ran more efficiently. These included removing looking at smaller boxes as it did not increase the speed of the model and instead resulted in more work for the model to separate the system. This is due to the size of the systems being simulated, as splitting up and reforming the whole systems into smaller boxes, was not efficient due to the small size of the overall system. Were the system to be scaled up to a much bigger size, this method would likely again become more efficient. As work on the 3D algorithm progressed and the stochastic optimisation method was finalised, discussed in section 3.2.1.2, this was retroactively added into the 2D algorithm as although it gave the same results as the equations discussed in section 3.1.2, the stochastic optimisation method was much more efficient.

Finally, more checks were added for instances when the model cannot find a valid spot for the particle to rest; this ensured that there is not a valid point close by that the model has missed.

3.1.1.12 Undergraduate MEng masters project work, Lewis Cartwright and Luke Convery.

The work around the forces present at the contact points of the particles under a shear force was carried out by two MEng students, Lewis Cartwright and Luke Convery, using structures supplied by the 2D model I generated.

When forces are applied on the top and bottom of a system, it results in torque on the outer particles on which the force is applied, that is then transferred through particle-to-particle contacts throughout the system. To balance this torque, the angle of rotation for each particle is required, which then allows the relative stress at each point of contact to be calculated. This was done by using the following steps and equations. The variables used within these equations are defined at the end of this section in Table 3.1.[3,4]

1: Calculate the position of the ends of the springs on each particle (i = 1 to n).

$$x_{i_2} = (x_{i_1} - x_{i_c})cos\theta_i + (y_{i_1} - y_{i_c})sin\theta_i + x_{i_c} \quad (3.1)$$

$$y_{i_2} = (x_{i_1} - x_{i_c})sin\theta_i + (y_{i_1} - y_{i_c})cos\theta_i + y_{i_c} \quad (3.2)$$

2. Calculate the forces caused by the particle-particle interactions (i = 1 to n).

$$F_{h_{ijnew}} = \begin{pmatrix} x_{j_2} - D_{x_j} \\ y_{j_2} - D_{y_j} \end{pmatrix} - \begin{pmatrix} x_{i_2} - D_{x_i} \\ y_{i_2} - D_{y_i} \end{pmatrix} \quad (3.3)$$

3. Calculate the forces caused by particle-wall interactions (i = 1 to n).

$$F_{h_{w_i}} = \begin{pmatrix} x_{w_i} \\ y_{w_i} \end{pmatrix} - \begin{pmatrix} x_{i_2} \\ y_{i_2} \end{pmatrix} \quad (3.4)$$

4. Calculate the resultant force through the sum of all forces acting on a particle (i = 1 to n).

$$F_{R_i} = F_{s_i} + \sum F_{h_{ij}} + F_{h_{w_i}} = \begin{pmatrix} x_{R_i} \\ y_{R_i} \end{pmatrix} \tag{3.5}$$

5. Calculate the torque from the initial shear force on each particle (for i = 1 to n).

$$P_{s_i} = \begin{pmatrix} x_{s_i} \\ y_{s_i} \end{pmatrix} - \begin{pmatrix} x_{i_c} \\ y_{i_c} \end{pmatrix} \tag{3.6}$$

$$\tau_{s_i} = F_{s_i}^{P_{s_i}} = F_{s_{x_i}} P_{s_{y_i}} - P_{s_{x_i}} F_{s_{y_i}} \tag{3.7}$$

6. Calculate the torque caused by particle-particle interactions (for i = 1 to n). Note that $F_{h_{ij}}$ is used here and not $F_{h_{ij_{new}}}$ as only the torque from the particles rotation is calculated here, not the particles displacement.

$$P_{p_{ij}} = \begin{pmatrix} x_{ij_2} \\ y_{ij_2} \end{pmatrix} - \begin{pmatrix} x_{i_c} \\ y_{i_c} \end{pmatrix} \tag{3.8}$$

$$\tau_{p_{ij}} = F_{h_{ij}}^{P_{p_{ij}}} = F_{h_{x_{ij}}} P_{p_{y_{ij}}} - P_{p_{x_{ij}}} F_{h_{y_{ij}}} \tag{3.9}$$

7. Calculate the torque caused by particle-wall interactions (for i = 1 to n).

$$P_{w_i} = \begin{pmatrix} x_{w_{i_2}} \\ y_{w_{i_2}} \end{pmatrix} - \begin{pmatrix} x_{i_c} \\ y_{i_c} \end{pmatrix} \tag{3.10}$$

$$\tau_{w_i} = F_{h_{w_i}}^{P_{w_i}} = F_{h_{w_{x_i}}} P_{w_{y_i}} - P_{w_{x_i}} F_{h_{w_{x_i}}} \tag{3.11}$$

8. Calculate the torque created from the displacement of each particle (for i = 1 to n).

$$\tau_{D_{ij}} = F_{R_i}^{P_{p_{ij}}} = F_{R_{x_i}} P_{p_{y_{ij}}} - P_{p_{x_{ij}}} F_{R_{y_i}} \tag{3.12}$$

9. Calculate the overall torque on each particle from the sum of their torques (for i = 1 to n).

$$\tau_{R_i} = \tau_{s_i} + \tau_{w_i} + \sum \tau_{p_{ij}} + \sum \tau_{D_{ij}} \tag{3.13}$$

10. If the overall torque and resultant force are below 0.001, exit the program and output results.

11. If either value of the overall torque or resultant force is over 0.001, adjust the value of the angle of rotation and the displacement vector (for i = 1 to n).

$$\theta_i = \theta_i + (0.2 \frac{\tau_i}{r_i^2}) \tag{3.14}$$

$$D_i = D_i + \alpha F_{R_i} \tag{3.15}$$

12. Return to step 1.

Table 3.1: Variables used within the above equations to determine the forces present at each contact point in a system when placed under a shear force.

Variable	Symbol	Unit
Angle of rotation for particle *i*	θ_i	radians
Updated *x* or *y* coordinate of the spring location for particle *i*	x_{i_2}, y_{i_2}	mm
Current *x* or *y* coordinate of the spring location for particle *i*	x_{i_1}, y_{i_1}	mm
x or *y* coordinate of the centre of particle *i*	x_{i_c}, y_{i_c}	mm
Updated force between particles *i* and *j*	$F_{h_{ij_{new}}}$	N
Updated *x* or *y* coordinate of the spring location for particle *j*	x_{j_2}, y_{j_2}	mm
Displacement of the *x* or *y* coordinate of particle *i*	D_{x_i}, D_{y_i}	mm
Displacement of the *x* or *y* coordinate of particle *j*	D_{x_j}, D_{y_j}	mm
Force caused by particle-wall interactions for particle *i*	$F_{h_{w_i}}$	N
x or *y* coordinate of the contact point between the wall and particle *i*	x_{w_i}, y_{w_i}	mm
Resultant force acting on particle *i*	F_{R_i}	N
Applied shear force to particle *i*	F_{s_i}	N
Position vector used for shear force torque for particle *i*	P_{s_i}	mm
x or *y* coordinate of the shear force being applied to particle *i*	x_{s_i}, y_{s_i}	mm
Torque due to applied shear force for particle *i*	τ_{s_i}	nm
x or *y* coordinate of the position vector used for shear force torque	$P_{s_{x_i}}$, $P_{s_{y_i}}$	mm
Position vector used for particle-particle torque between particles *i* and *j*	$P_{p_{ij}}$	mm
x or *y* coordinate spring end attached to particle *i* after rotation	x_{ij_2}, y_{ij_2}	mm
Torque due to particle-particle interactions between particles *i* and *j*	$\tau_{p_{ij}}$	nm
x or *y* coordinate of the position vector used for particle-particle torque between particles *i* and *j*	$P_{p_{x_{ij}}}$, $P_{p_{y_{ij}}}$	mm
Position vector used for particle-wall torque for particle *i*	P_{w_i}	mm
Updated *x* or *y* coordinate of the contact point between the wall and particle *i*	$x_{w_{i_2}}$, $y_{w_{i_2}}$	mm
Torque due to particle-wall interactions for particle *i*	τ_{w_i}	Nm
x or *y* coordinate of the position vector used for particle-particle torque between the wall and particle *i*	$P_{w_{x_i}}$, $P_{w_{y_i}}$	Mm
Torque due to the displacement of particle *i* by particle *j*	$\tau_{D_{ij}}$	Nm
Resultant torque for particle *i*	τ_{R_i}	Nm
Torque for particle *i*	τ_i	Nm
Radius of particle *i*	r_i	Mm
Displacement of particle *i*	D_i	Mm
Angle between contact point and x-axis	α	radians

3.1.2 Code Walkthrough

This next section will go through the algorithms used in the final version of the 2D model to show how it works through each stage.

The code is broken into three subroutines: the initial setup; when the particle is falling; and then its final placement. In addition, there is one module that contains all of the universal variables that are carried across all three subroutines.

The main variables contained within the module are: the box dimensions; the number of particles; the radii being used; and the stored positions of the already placed particles.

The first part of the initial setup subroutine sets up the local variables that are required, and then requests the user to start the program. The user is then prompted to enter how many different radii they would like to be present in the system and to enter those radii. Next, the model determines which of the entered radii is the smallest and which is the largest, to use when determining the size of the box. When the program is being looped to produce multiple results, this section is omitted, since the radii is already known and to stop the program being interrupted by prompting the user for inputs.

Now that the size of the particles present is known, the box size can be determined. This is based on the largest particle radius so that a sensible number of large particles can fit, instead of having a system containing too few particles to form a sufficiently sized bed. The box size can then be used to allocate the size to various arrays used later in the algorithm. These include the array that contains the entire contour plot, which is still a grid of the box that contains a point for each integer spot within the box, and the “Ones” array, which stores the coordinates of the valid points on the contour plot, so named as a contour plot point is one of three options, either “0” denoting a blank space, a “-“, denoting being covered by a particle, or a “1” denoting being a valid point based on the distance from the current bed.

In the looped algorithm, the model now enters the section of code that will be looped for a number of times equal to the number of overall beds that has been requested to be simulated.

The next stage of the model is to place the initial bed layer of particles into the box, shown in Figure 3.5. This is done by randomly picking a radius, from the inputted radius options, and an *x* coordinate within the boundaries of the box. The model confirms that this position, using the particles radius as its *y* coordinate, is not already covered by another particle. As the particle is resting upon the base of the bed, no other conditions are required, so once this check is passed the particle location is saved, and this section of the algorithm looped to place the rest of the initial bed layer. This loop goes for a sufficiently large number of iterations, currently set at 10000000. Due to the possibility of there still being a position where a particle could still rest upon the bed, the model then loops across all of the bottom layer of the bed, using the smallest particle radius as the *y* coordinate, checking if there are any more places for a particle to fit.

Following this the program starts looping the second subroutine to add in the rest of the particles to the box, shown in Figure 3.6. This is done as many times as needed until the box is full, or the number of particles specified has been reached.

After initialising the second subroutine’s local variables, the first check made is whether the box is full or not, as this check comes before adding a new particle in every loop. Each time the algorithm tries to place a particle beyond the roof of the box, a counter is iterated. Once this value reaches a sufficiently large value the box is deemed full, and no more particles added.

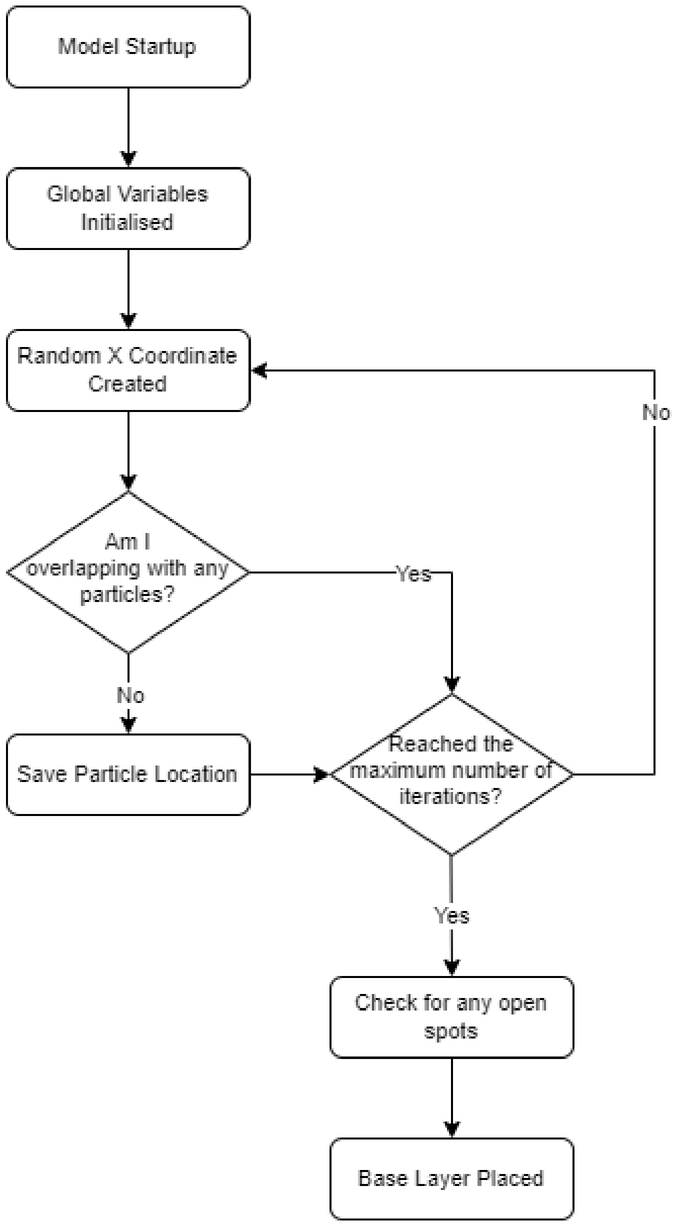


Figure 3.5 - Flowchart showing the stages the algorithm goes through to produce the base layer of particles for a bed system

For each particle added, a random particle radius is chosen from the list of entered radii and a random starting spot is chosen at the top of the bed, by generating an *x* coordinate within the bounds of the box size. The third subroutine is called at this point whilst the particle is falling. This subroutine takes the starting position of the particle and iterates the *y* coordinate downwards one step at a time. At each point the model confirms that it is not touching another particle allowing the loop to continue. Once an impact does occur, the model saves the location of the impact and moves back into the main second subroutine.

At this point the model creates a contour map of the system, to locate the highest points that a particle can rest upon. The particle jumps from the location of impact to the nearest of these points, and then compares the height of the two points either side of it. The model then moves in the direction with the lowest y value, simulating gravity, until it reaches a point where the contour points on both sides are higher than it, so it rests there.

However, at this point the location of the particle is still saved as an integer, resulting in gaps between particles due to rounding. As a result, using the location of the two particles it is resting on, and the distances between them, the model calculates the triangle that the three particles make to determine the final real values of the new particles coordinates. The distances between the original particles and the new particle are the sum of the radii of the particles. Knowing these three distances, a triangle can be formed between the centre of the three particles of which the angles can then be calculated. The gradient of the lines connecting the new particle with the old particles can then be calculated, then allowing the final determination of the centre point of the new particle.

Checks are then run to confirm that: the new particle is resting upon the old ones instead of attempting to balance over an edge; that the new particle is not overlapping with any old particles; and that it is contained within the box. Having passed these tests, the coordinates are then saved into the list, and the model then resets the appropriate values and loops back to the start of the particle addition subroutine.

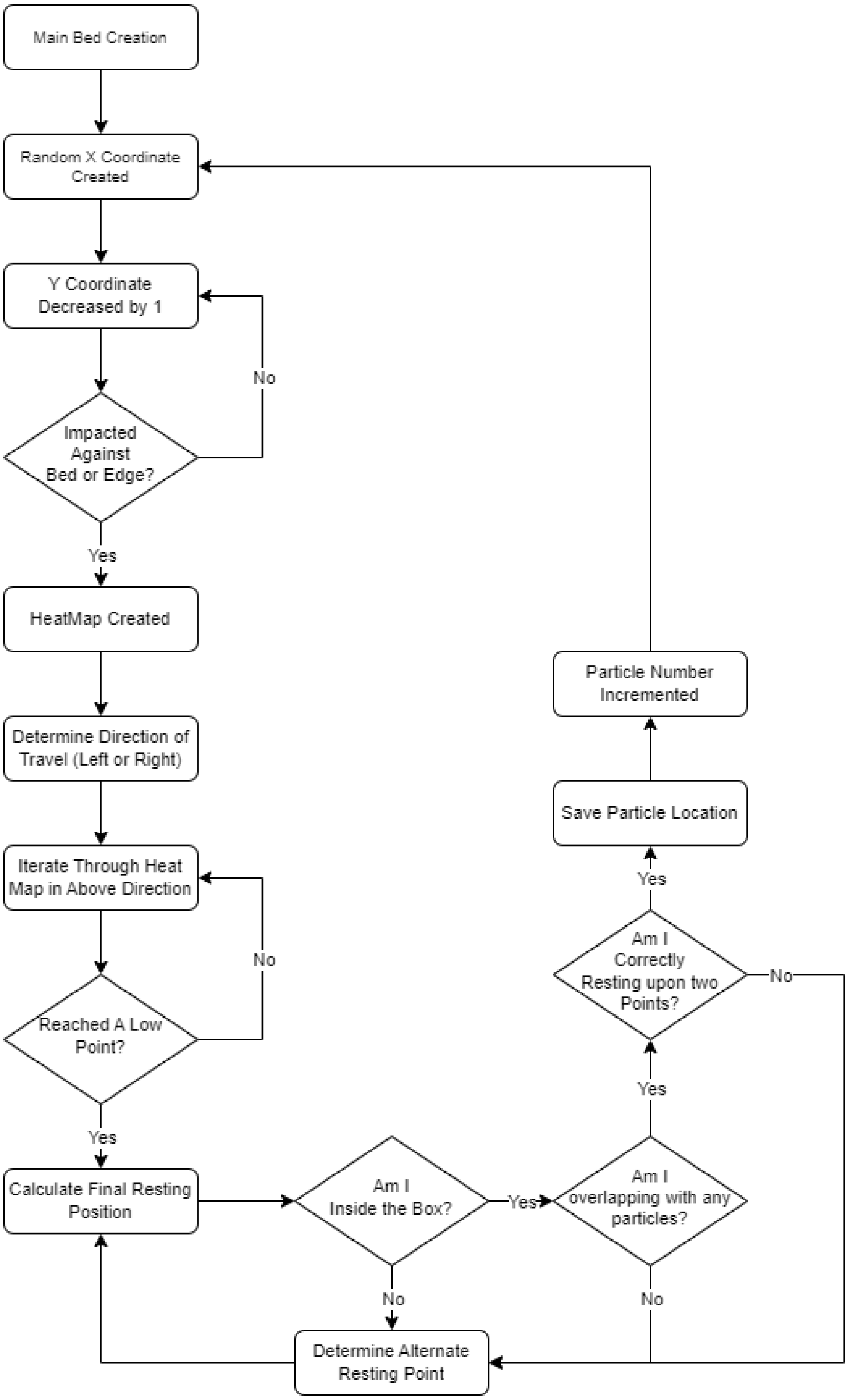


Figure 3.6 - Flowchart showing the stages the algorithm goes through to add particles to fill up a bed system

Once the box is full, or a specified number of particles have been added, the model returns to the initial subroutine where the user is prompted as to whether they want to save the list of particle locations, their contacts, the contour map from any particle addition, or calculate the particle fraction of the system. Currently, the model saves these files to the same folder as it is contained in.

A final query then confirms the user understands the model is about to end.

3.1.3 Simulations Run

All of the runs completed using the 2D algorithm were completed on ARCHIE-WeSt[5], a regional supercomputer centre based at the University of Strathclyde.

500 systems were created for each of the following systems: radius 10, radius 10 and 20, and radius 10 and 50 (henceforth when referring to particle sizes in a system, the notation r_p = “radius” will be used. Binary mixture systems shall be referred to as r_p = “radius a”, “radius b”). The ratio of addition for each radii in these system is 1:1.

Each of these systems had its packing fraction calculated, the number of contacts each particle had determined, and the size of individual voids present calculated. The algorithms that determine these values are described in section 3.3.

100 runs were also created for investigating percolation with r_p = 10, 20 at the following ratios of addition Large:Small particles: 1:1, 1:1.5, 1:2, 1:2.25, and 1:2.5. The percentage of these structures that contained a percolation chain was determined.

Particle sizes in these and future runs were chosen as round numbers that could easily be used for different size ratios, as the model can handle particles with non-rounded values, however is it much easier to discuss particle ratios of 10:20 than e.g. 17:34.

3.2 3D Algorithm Methodology

As mentioned above, the program was modified so that it could replicate the packing of particles in 3D as well as 2D and produce images as well. A 3D version of this model would allow us to gain a much better insight into how the packed bed forms, and its properties, due to the increase in accuracy that the third dimension brings, and also due to it being a more realistic simulation of what would be happening in a real-life experiment.

3.2.1 Timeline of Changes

This section looks through the stages that the 3D algorithm went through and the reasons behind each of the changes, from its initial setup to the algorithm used to produce data discussed in the later sections.

3.2.1.1 2D to 3D changes

Whilst changes had to be made to create the 3D algorithm from the 2D version, it retained its previous structure of three subroutines and a module. The first thing changed from the 2D algorithm was the addition of the z axis into every stage of the model, and variables to account for the new coordinates in the particles' locations.

Another change from the 2D algorithm is that the particle is now looking for three particles to rest on instead of just two. It was decided that the possibility of a perfect square forming and requiring four particles to be rested on was exceedingly rare and not worth adding into the model.

With the extra dimension also comes the possibility of a particle resting in a corner spot, so the model is now able to detect when a new particle is close enough to two walls, and therefore only needs one particle to rest on, and then be up against the corner. A discussion was held with my supervisors as to whether to implement periodic boundary conditions, and at this stage it was decided to leave its implementation until later. Unfortunately, this did not come to pass as focus was shifted to work on the chain particle algorithm.

3.2.1.2 Stochastic placement

The main detail added into the 3D algorithm, different from the 2D version, is the change from calculating the final real position of the new particle; instead determining it through a stochastic optimisation function. This function works by taking the contour plot point closest to the impact and making small adjustments, in each direction, until a position is reached that satisfies the previous 2D algorithm conditions, i.e., resting on the correct number of particles, resting in between their centre points. This ensures that there are no errors during the calculation making it much more reliable, as previously on occasion the model returned a null value using the old method.

A version of this algorithm is shown in Figure 3.7, with some altered variable names for ease of presentation. Lines 1 to 7 set up the initial variables with their desired values, with *newCoords* containing the current position of the particle to be adjusted, *stochDists* containing the distances between that particle and the two particles it will be adjusted to be resting upon, *sumDist* being the sum of those two distances, and *dx* being the size of the adjustments being made. The outer loop, controlled by the integer “*a*”, is the number of times the adjustments will be scaled down, which occurs at lines 11 to 12. Each loop the adjustment factor is reduced by a factor of 10.

The main inner loop, controller by the integer “*c*”, is how many adjustments are made at each scale. For each adjustment, a random number is determined between -1 and 1, which is then multiplied by the adjustment scale to produce a value which is then added to the current x or y value. The adjustments made to the x and y values of the particle are separate, shown in lines 13 to 16.

In lines 19 and 20, the distances between the particle being adjusted and the resting particles are recalculated for its new position. *PartCoords(b,1)* and *PartCoords(b,2)* being the x and y values of resting particle *b* respectively. The radii of the two particles for each distance are also subtracted, to give the distance between their edges, rather than their centres. If the distance is less than 0, it means the particles are now overlapping and this adjustment is not saved, accomplished by setting the "*ibad*" variable to 1, shown in line 22. The other check made to allow the new position to be saved is that the sums of the distances between the resting and new particles is lower than at the previous location, as the aim to is reduce this value to 0, without going under it. If these conditions are met, the new locations are saved, shown in line 28, the sum distance stored, line 30, and then the new adjustment made.

Once all the loops have been completed, the final positions calculated are saved into new variables, lines 34 and 35, that are then used going forward.

```
1	newCoords(1) = TempX
2	newCoords(2) = TempY
3	newCoords(3) = TempZ
!	These variables store the current integer position of the particle to be adjusted

4	stochDists(1) = FinalPartDist1,2)
5	stochDists(2) = FinalPartDist(2,2)
6	stochDists(3) = FinalPartDist(3,2)
!	These variables store the distances between the new particle and the particles it will
be resting upon

7	sumDist = stochDists(1) + stochDists(2) + stochDists(3)
!	This variable stores the sum of the above distances, and is the value we are trying to
minimise

8	dx(1) = 10 * RadLarge
9	dx(2) = 10 * RadLarge
10	dx(3) = 10 * RadLarge
!	These variables store the starting amounts by which the position will be adjusted

11	do a = 1, 10
!	This loop determines the number of times the adjustment amount will be shrunk
12		do b = 1, 3
13			dx(b) = dx(b) / 10 ! This shrinks the adjustment value
```

```
        end do
        do c = 1, 500
!   This loops determines the number of adjustments made to the particle
            do b = 1, 3
                call random_number(RX)
                stochxnew(b) = newCoords(b) + dx(b) * (2*RX-1)
            end do
!   For each coordinate (x, y, z), they are adjusted by a random fraction of the total adjustment value

            ibad = 0

            do b = 1, 3
!   The new distances between the particles are calculated based on the adjusted positions
                stochDists(b) = ((PartCoords(b,1) - stochxnew(1))**2) + ((PartCoords(b,2)-stochxnew(2))**2)
                stochDists(b) = sqrt(stochDists(b)) – RadT – MLr(FinalPart(b,1))
                if (stochDists(b) < 0) then
!   If the particles are overlapping then a variable (ibad) is set to 1, to ensure this adjustment is rejected
                    ibad = 1
                end if
            end do

            stochynew = stochDists(1) + stochDists(2) + stochDists(3)
!   The new total distance is calculated

            if (stochynew < sumDist .and. ibad == 0) then
!   If this distance is smaller than the previous distance and there are no overlaps, the new location is saved
                do b = 1, 3
                    newCoords(b) = stochxnew(b)
                end do
                sumDist = stochynew ! And the new total distance saved
            end if
        end do
    end do
!   This process repeats with the adjustment distance shrinking each time to obtain more specific adjustments until a precise location is determined for the new particle

    NewX = newCoords(1)
    NewY = newCoords(2)
    NewY = newCoords(3)
!   The final particle location is then saved
```

Figure 3.7: Stochastic optimisation algorithm used for shifting particles from the integer spot on the contour plot to the real location resting on top of their nearest particles

3.2.1.3 Extra checks

Some extra checks were also added to ensure that the model is able to find the correct resting point, as this was where the model was having most of its issues due to the higher complexity of the 3D contour map. These involved allowing the model to change the resting particles to look at other nearby options until it found the particles that it should be being rested on. In the case of an overlap, the model replaces one of the current resting particles with the particle being overlapped with, as an overlap meant that the new particle should be resting on the overlapped particle. As shown in Figure 3.8, particle D is being added to the system and incorrectly attempted to rest upon particles A and B. This causes an overlap with particle C, so the algorithm swaps C with the closest of particles A or B, which in this case is particle B, so that it becomes a resting particle. This then allows particle D to rest correctly. Figure 3.8 is a 2D representation of the 3D issue for the ease of visualisation, however these checks were also added in future versions of the 2D code used for the chain particles algorithm.

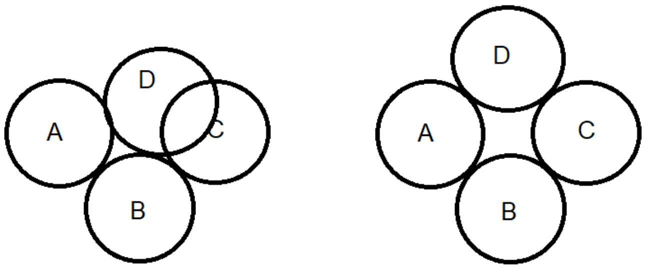


Figure 3.8: Representation of an particle placement on the left where particle D has incorrectly rested on particles A and B, but then is corrected in the right image to be resting upon particles A and C.

Further checks were required for when the particle is near the edge of the box. Previously there has been a binary statement for if a particle was resting against an edge or not, meaning that if the particle finds itself one spot away from the edge, it would still be looking for three particles to rest upon. However, now it is able, when near the edge to swap between looking for an edge resting spot or not, once it has exhausted its other possibilities.

3.2.2 3D Code Walkthrough

This next section will go through the code used by the final version of the 3D model to show how it works through each stage.

As with the 2D code, it is broken into three subroutines: the initial setup; when the particle is falling; and then its final placement, and one module which contains all of the universal variables that are carried across all three subroutines.

The main variables contained within the module are: the box dimensions; the number of particles; the radii being used; and the stored positions of the placed particles.

The first part of the initial setup subroutine sets up the local variables that are required, as mentioned in the 2D algorithm section, and then requests the user to start the program. The user is then prompted to enter how many different radii they would like to be present in the system and to enter those radii. The model then determines which of the entered radii is the smallest and largest, to use when determining the size of the box. When the program is being looped to produce multiple systems, this section is omitted as the radii are already known.

The next stage of the model is to place the initial bed layer of particles into the box. This is done by randomly generating x and z coordinates, confirming that there is no other particle already overlapping, and then saving the particle there. The *y* coordinate of each of the particles is equal to their radius. A final check is then run to confirm that there are no available positions by looping through the box, looking for a position that a particle can fit in.

After this the program starts looping the second subroutine to add in the rest of the particles to the box. This is done as many times as needed until the box is full, or the number of particles specified has been reached.

After initialising the second subroutines local variables, the first check made is whether the box is full or not, as this check comes before adding a new particle in every loop. A random particle radius is then chosen from the list of entered radii and a random starting position is chosen at the top of the bed. The third subroutine is called at this point whilst the particle is falling. The particle iterates downwards one coordinate at a time, and at each point the model confirms that it is not touching another particle. Once this does occur, the model saves the location of impact and moves back into the main second subroutine.

At this point the model creates a contour map of the system, to locate the highest points that a particle can rest upon. The particle jumps from the location of impact to the nearest of these points and then starts moving along them in a downwards direction, to simulate gravity. As in the 2D model, once it reaches a point where all adjacent contour points are higher than it, it rests there.

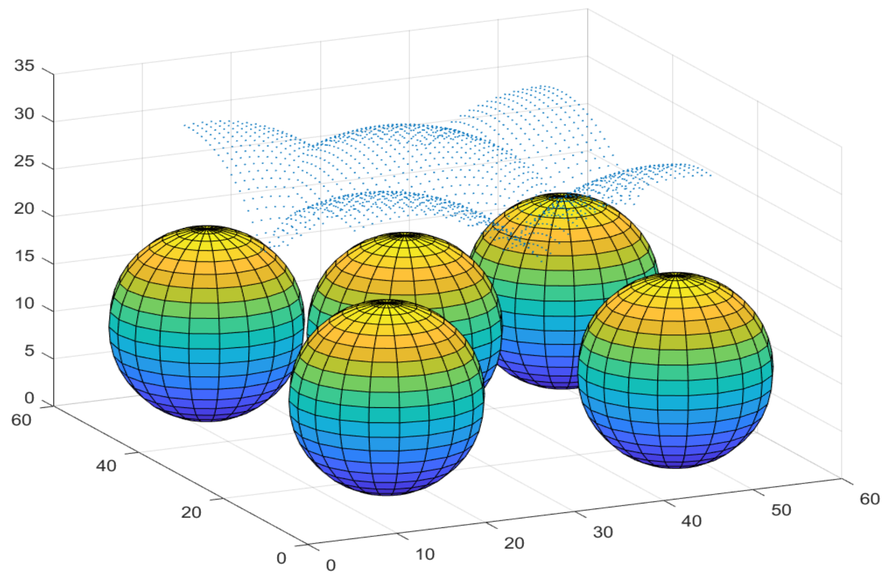


Figure 3.9: Contour map showing possible points for a new particle in a small system of 3D particles

However, at this point the location of the particle is still saved as an integer, so there would be gaps between particles due to rounding. Therefore, as described earlier, the model uses a stochastic optimisation function to move the final location around in increasingly small increments, to try and reduce the distance between the new particle and all the particles it is resting on to 0.

Checks are then run to confirm: that the new particle is resting upon the old ones instead of attempting to balance over an edge; that the new particle is not overlapping with any old particles; and that it is contained within the box. Having passed these tests, the coordinates are then saved into the list, and the model then resets the appropriate values and loops back to the start of the particle addition subroutine.

Once the box is full, or a specified number of particles have been added, the model returns to the initial subroutine where the user is prompted as to whether they want to save: the list of particle locations; their contacts; the contour map from any particle addition; or calculate the particle fraction of the system. The latter two of these were removed in the finalised version of the algorithm, as the contour map was used for error checking, and an alternative method was used for calculating the packing fraction. Currently, the model saves these files to the same folder as it is contained in.

A final query then confirms the user understands the model is about to end.

3.2.3 Simulations Run

All of the 500 runs completed for each system using the 3D algorithm were performed on ARCHIE-WeSt[3]. The ratio of addition for each radii in these system is 1:1.

Each of these systems had its packing fraction calculated and the number of contacts for each particle was determined.

100 binary mixture systems were also created for investigating percolation with r_p = 10, 20 at the following ratios of addition Large:Small particles: 1:1, 1:2, 1:3, 1:4, 1:5, and 1:6. The percentage of these structures that contained a percolation chain was determined.

3.3 Other Functionalities and Data Collection

Throughout creating the model, there have been functionalities added and removed that have been separate to how the model runs but have given options for the user to process the information produced.

At the start of the program, it requests the input of the radius that will be used for the particles in the model. The model also asks how many different particle sizes are needed, currently allowing for between 1 and 5, so there can be variation in the sizes present. Certain versions of the model also allow the user to enter a mean and standard deviation, to allow the model to place particles with sizes of a standard distribution based upon the inputted data instead of a binary sized system, allowing more realistic systems to be created. However this was left as a separate option within the algorithm and not investigated further.

Different methods of particle addition have been investigated, with the packing being ordered from the bottom to the top, as well as placing the particles randomly throughout the box with no need for them to be in contact with each other. This can be used to simulate the particles in suspension in solution, however at this time it has not been investigated further in favour of refining the packed particle bed approach.

The program also has the capability to calculate the fraction of the box that is either voids between particles or occupied by particles, as well as the sizes of the individual voids present in the system. These values can be used to compare against known values to confirm the realism of the model, as well as gain more information about the structure of the system created.

The algorithm counts each particle in the system by its radius, and then works out the total area, or volume in 3D, of particles of each radius, as shown in Equations 3.16 and 3.17. When summed, this value represents the total area of the system that is present as particle surface. This is then divided by the total area, or volume, of the box to gain a fraction of the system that is then covered by particles.

$$Area = \ (p_1 * (\pi * \ r_1^2)) + \ (p_2 * (\pi * \ r_2^2)) + \cdots \ (p_n * (\pi * \ r_n^2)) \tag{3.16}$$

$$Volume = \ (p_1 * \left(\frac{4}{3}\pi * \ r_1^3\right)) + \ (p_2 * \left(\frac{4}{3}\pi * \ r_2^3\right)) + \cdots \ (p_n * \left(\frac{4}{3}\pi * \ r_n^3\right)) \tag{3.17}$$

where *p* is the number of particles of radius *r*, and *n* is the number of different radii present in the system.

The model can also determine the location of the points of contact between each of the particles, as well as noting which particles are in contact with each other. This is done by looking at the distance between a particle and each other particle in the system. If the distance is equal to the sum of the two particles radii, then they are in contact. Each particle is numbered in order of when it was added to the system, so a file can be outputted showing each particles contacts using those assigned numbers, for example particle 1 is in contact with particle 3.

A section of the 2D model is also capable of determining the shapes that groups of particles make up within the system to form a void, and then calculate the area of that void. As mentioned above, the model can determine which particles are in contact with each other, this can now be used to investigate the shapes that chains of the particles form. The algorithm starts looking for three vertex shapes, where vertices can be a particle or an edge, and then increments the number vertices, up until ten. This limit was created for time management purposes as this algorithm could theoretically look for shapes with an infinite number of vertices, however ten was deemed sufficient to find most, if not all shapes, and did not use too much computational time. Starting from particle one, the algorithm loops through the list of its contacts, for each particle in contact, the algorithm then loops through its contacts. This is done recursively, looking through the system until a chain starts and ends with the same particle. Once a chain has been found, the recursive loop unwinds, storing the particle number at each step so that once it has fully unwound, the chain creating the shape is fully saved. This shape is then checked against two criteria to ensure that it a valid shape to save permanently, the first of which is that it must be a unique shape that has not already been found, i.e. a shape could be found multiple times starting from each of its vertices, and be saved as 1,2,3, 2,1,3, etc, even though they make up the same shape. The other criteria is that there are no particles within the shape found, as therefore the area calculated for that shape would not be entirely void. This is done by creating the outline of the shape between each of the particles centre points. Looping through every other particle in the system, a line is drawn from its centre to the left most edge of the box, and the number of times it crosses one of the shapes outer lines is counted. If the total is odd, then the particle is inside the shape, and if it is even then it is outside the shape. Now that the shape has been deemed valid, it is saved, and its area can be calculated using Equation 3.18.

$$Total\ Shape\ Area\ = \frac{1}{2}\sum_{i=1}^{n}(x_i y_{i+1}) - (x_{i+1} y_i) \qquad (3.18)^6$$

Where x and y are the centre coordinates of each particle.

Once the total area of the shape between the particles has been determined, the sectors of each of the particles that overlap with this shape are calculated and subtracted from the total area, thus leaving behind the void area.

$$Particle\ Sector\ Area = \frac{\theta}{360} * \pi r^2 \qquad (3.19)$$

Where Θ is the inner angle of the sector and r is the radius of the particle.

The algorithm used to find percolation structures uses the same recursive loop as above, detailed in Figure 3.10, however instead of looking for loops back to the starting particle, the chains only start from particles that are in contact with the lefthand edge and terminate when they have reached a particle in contact with the righthand edge. The same check is performed to ensure that each chain found is unique before they are saved.

One of the main alternate functionalities is to have the model produce systems that are not circular/spherical. This was done by merging circular particles together to form chains, which is discussed in greater detail in Chapter 6.

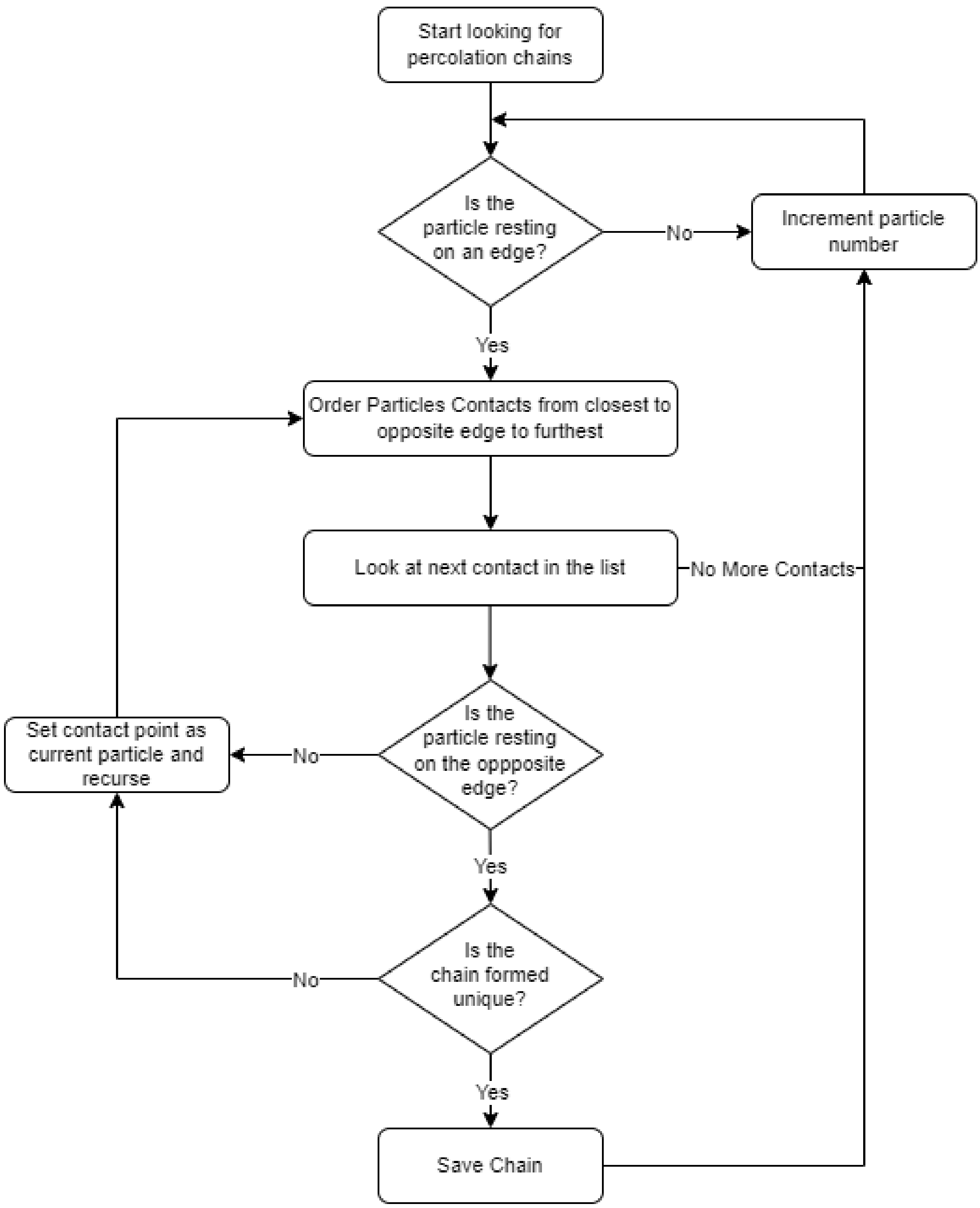


Figure 3.10: Flowchart showing the stages the algorithm goes through to adjust a particles location from an integer to real value using stochastic optimisation

3.4 Summary and Conclusions

This chapter has discussed the algorithms produced to simulate the packing of particles in both 2D and 3D under gravity. The algorithms are able to produce realistic representations of these systems which we have been able to analyse, the data obtained from which is discussed in the following two chapters.

These algorithms will be saved on the University of Strathclyde Pure repository.

3.5 References

(1) A. Balfour and D. H. Marwick. *Programming in Standard ForTran 77*. Heinemann Educational Books, **1979**.

(2) W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling. *Numerical Recipes in Fortran 77 Second Edition.* Cambridge University Press. **1993**.

(3) L. Cartwright, Modelling Stress and Fracture in Agglomerated Structures, University of Strathclyde, Glasgow, 2021.

(4) L. Convery, Modelling the Fracture and Breakage of Agglomerates Under an Applied Shear Force, University of Strathclyde, Glasgow, 2022.

(5) *Archie-WeSt*. https://www.archie-west.ac.uk/ (accessed 2020).

(6) M. A. de Souza, H. R. Gamba and H. Pedrini. *Multi-Modality Imaging: Applications and Computational Techniques.* Springer. **2018.**

4 Results and Discussion – 2D Algorithm

This chapter discusses the systems created by the 2D algorithm discussed in Chapter 2. Specifically investigating the different properties of these systems with emphasis on features that might affect the strength of the particle bed structure when placed under stress. We start by looking at the packing of the systems created, before investigating the sizes of the voids and number of contacts each particle has. Finally, the forces present within the systems and how breakages caused by the application of shear stress were investigated.

4.1 Binary System Beds

The radii 10, 20 and 50 were chosen as they given radius ratios of 1:2 and 1:5, which would show either side of the range of particle size distributions. Ratios in between these ones were planned on being tested but were put to the side to focus on completing the later algorithms.

The packed bed with only r_p = 10 present shows a mostly regular square lattice structure, as shown in Figure 4.1a, even with the irregularity of the initial placement of particles at the bottom of the bed. In the centre of the bed, it can be seen that the average number of contacts is four, since each particle contacts two existing particles below it, and two particles are then placed above. However, there are significant edge effects with the hard walls of the box due to the small size of the systems and lack of periodic boundary conditions.

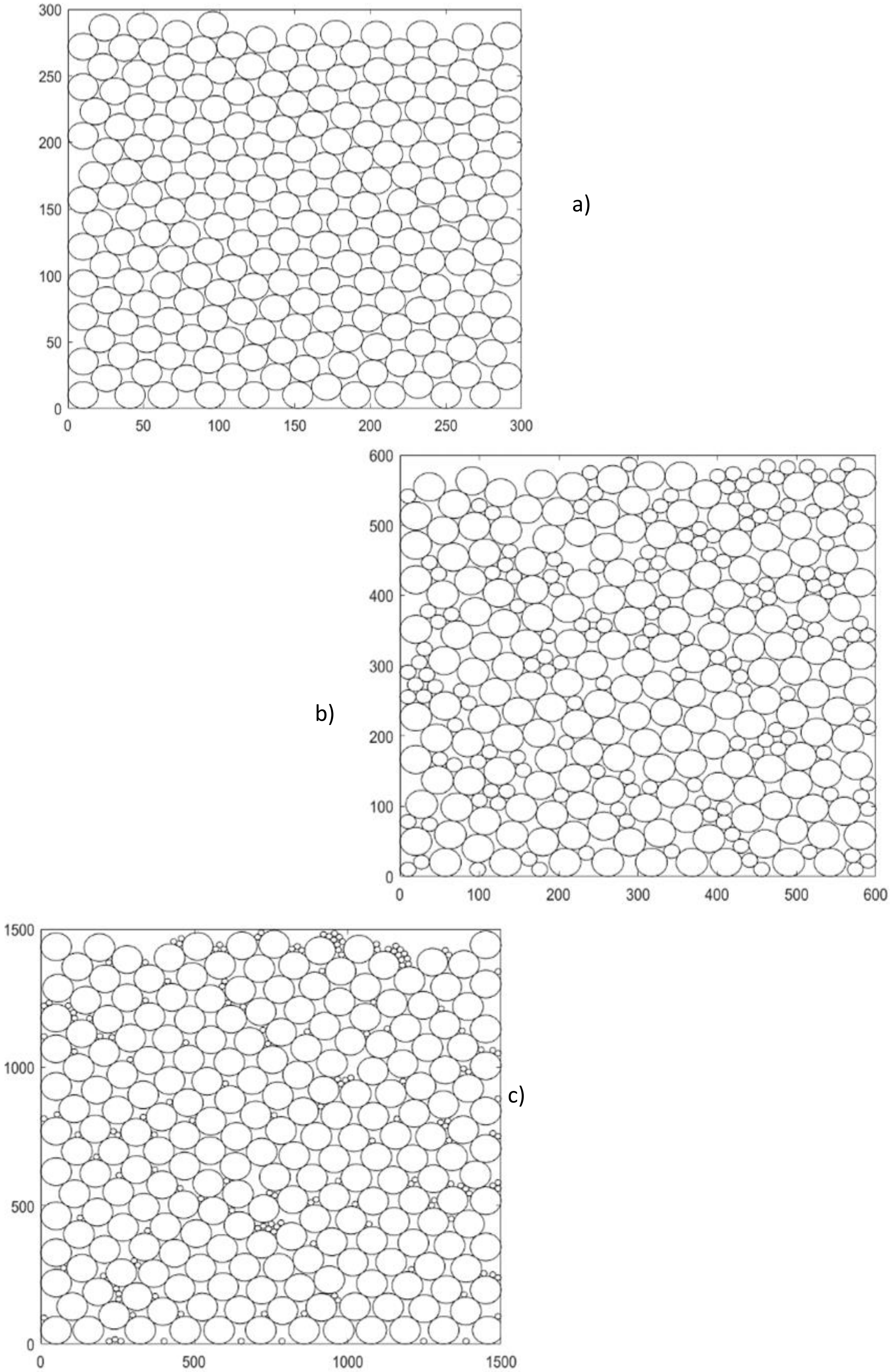


Figure 4.1: Packed beds of particles across four systems with different radii particles present. a: r_p = 10. b: r_p = 10, 20. c: r_p = 10, 50.

The irregularity of the packing increases once the bed also includes larger r_p = 20, as shown in Figure 4.1b. As the r_p = 10 are not small enough to fit inside the voids created by the r_p = 20, as with the r_p = 10 falling above the critical size ratio value, 0.41:1[1], discussed in the literature review, they instead contribute to the increased irregularity in void shape and size and compel the larger r_p = 20 to shift from a regular packing structure to accommodate for the smaller particles between them.

This effect is still apparent in the bed containing r_p = 10, 50, however to a lesser extent, as shown in Figure 4.1c. Due to the larger difference in particle size, two and occasionally three of the smaller particles can be seen to fit in between the larger particles without greatly affecting the placement of the particles landing above them. There are still instances of irregularity that spawn from the overabundance of smaller particles overfilling what might otherwise be a void, thereby forcing the addition of the next large particle to the side, preventing it from capping the putative void.

Note that the smaller particles filling in amongst the voids of the larger particles would increase the difficulty of washing the system, and the smaller particles forming clumps in between the larger particles would help bind them together, increasing the likelihood of agglomerates forming.

4.2 Packing Fractions

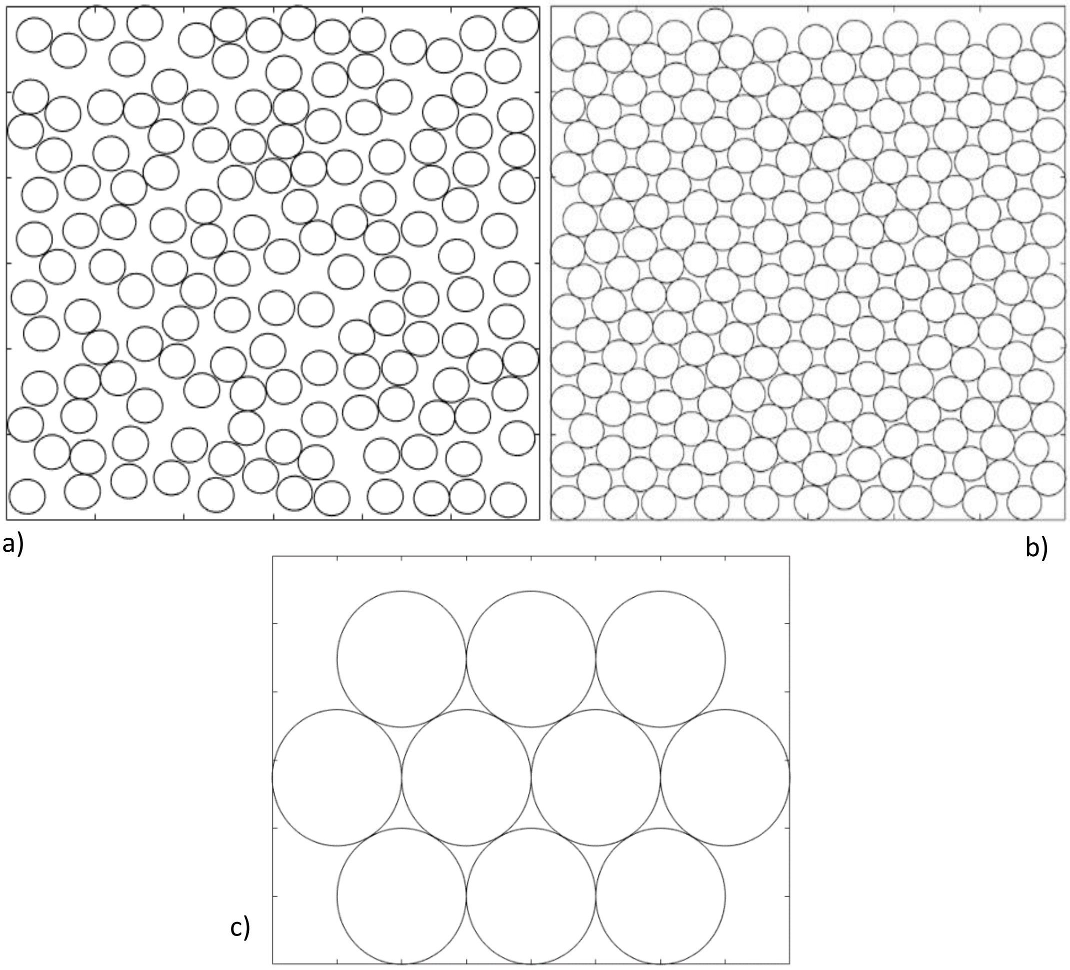
Table 4.1 shows the results of 500 runs of each system of different sized particles. Within these systems, the number ratio of large:small particles was 1:1.

Table 4.1: The Packing Fractions in Packed Bed Systems with different particle radii present.

	Packing Fraction		
Particle Radii	Minimum	Average ± Standard Deviation	Maximum
10	0.757	0.766 ± 0.007	0.806
10, 20	0.731	0.779 ± 0.005	0.791
10, 50	0.770	0.781 ± 0.005	0.796

As the difference between the sizes of the particles present increases, the average packing fraction increases. The system with only r_p = 10 has the lowest packing fraction, as whilst it is a partially ordered structure, with regions of short-range order, i.e. small sections of the system where the particles have packed efficiently, there is no way to fill in the voids between the particles. In the systems containing a larger size of particles, r_p = 20, 50, the r_p = 10 particles are now able to sit in the voids created by the larger particles, thus giving these systems a higher packing fraction. The difference between the r_p = 10, 20 and r_p = 10, 50 systems is likely due to the way that the particles pack together. This means that in the r_p = 10, 50 system, as previously mentioned, the r_p = 50 are able to form a partially ordered structure, with the smaller particles more able to fill the voids in between. Whereas in the r_p = 10, 20 systems, the larger particles are pushed out further from a regular structure by the smaller particles, due to their closeness in size. When the particle radius ratio between smaller and larger particles is 0.41:1,[1] or 10:24.39, scaling the 0.41 value to 10, my smallest particle radius, the smaller particles can fit perfectly inside the voids created by the larger particles. For any size ratio less than this value the smaller particles can fit into the voids between larger particles. As the size of the void created by a set of same sized particles in a regular pattern is proportional to the size of the particles, once you have passed this ratio the void fraction will remain similar for the larger particles, as the r_p = 10 do not affect the void formation and simply fill up space within them. The further variation in particle fraction would then come from the difference in the number of smaller particles that were placed within the voids, as it would hypothetically be possible to change the ratio between large and small particle in order to get an extremely densely packed bed where enough smaller particles are placed in amongst the larger ones, so that all the voids were filled with closely packed small particles, with the larger particles still maintaining a fairly regular packing pattern.

The theoretical maximums for these binary systems is unknown, as they are not using ratios for which the compact packing exists.[1] As adding more smaller particles to the systems shifts the larger particles out of an ordered hexagonal packing arrangement, the highest packing possible for the r_p = 10, 20 and r_p = 10, 50 systems is likely to be one in which the large particles pack on their own in a hexagonal lattice, and then the smaller particles are used to fill in any space left at the top of the box where large particles no longer fit.



a) b) c)

Figure 4.2: Examples of packing in systems with different particle addition methods. a) Random Sequential Adsoprtion (RSA). b) My Model. c) Triangular Lattice.

Packing fractions have been researched previously using random sequential adsorption (RSA) models, which calculate the maximum packing fraction to be roughly 0.547.[2] Our values exceed this by about 0.2; which is expected as while the packed beds presented here have a degree of randomness in the placement of the particles, the particles settle under gravity to create denser packing than with RSA, where particles are added at random without overlap until it is no longer possible.[3]

The highest possible packing fraction for a bed of circular particles of the same size is roughly $\frac{\pi}{\sqrt{12}} \approx 0.9069$,[4] so our values fall comfortably below this. This is because the model will never achieve perfect packing in a triangular lattice due to the random nature of the structure, especially the randomness of the base layer of particles. A comparison of these packing methods and one of our systems is shown in Figure 4.2, demonstrating the differences in how packed the structures are.

4.3 Number of contacts between particles

The number of contacts each particle has was also investigated as it is a parameter that can give us more information about how densely packed a bed is. It is also another metric through which my model can be compared to expected values to confirm that the simulations run give usable results.

As shown in Figure 4.3, across the different systems, particles will most often have two to five contacts. Particles with fewer contacts than this are infrequent, as having zero contacts requires being one of the initial particles placed on the bottom of the box ending up with no particles laying on top, and one contact being a resulting of a particle resting against a wall and one other particle.

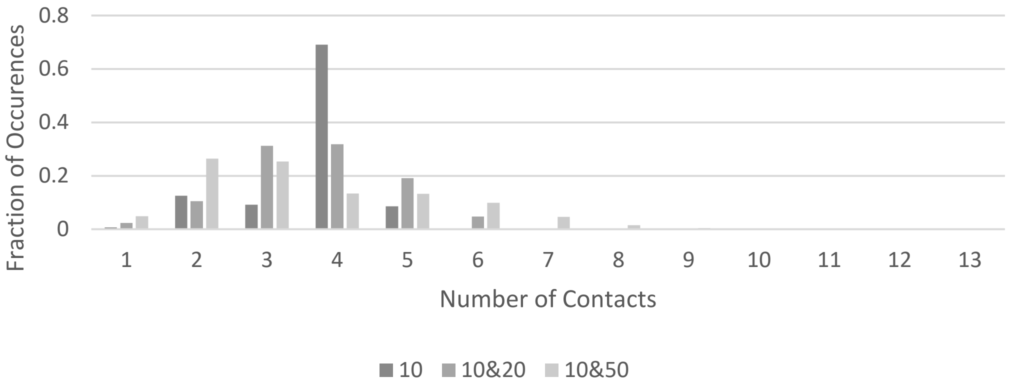


Figure 4.3: The frequency of the number of contacts per particle across the investigated systems.

In the systems with only r_p = 10, there are no particles with greater than six contacts, as required by the geometry of packing. In a perfectly ordered system, each of the particles would have six contacts, as they would form a triangular lattice arrangement, and thus six is the maximum number of contacts possible. As our systems have a degree of disorder within them due to the randomness of the initial particle placements, it is rare for this to occur by chance, as shown by the frequency of occurrence bar in Figure 4.3 for six contacts being minimal compared to the other contact amounts. Therefore, in the middle of the bed, the average number of contacts will be four, since each particle added to the system creates two new contacts each shared by the two particles.

As larger particles are added into the system with the r_p = 10, the number of contacts the particles are able to have with smaller particles increases, as the increased circumference of the larger particles allows for more contact points to be made. Figure 4.4a shows the frequency of each number of contacts for r_p = 10 across each of the systems created. Figure 4.4b shows the frequency of each number of contacts for larger particles, r_p = 20, 50, present across each of the systems created.

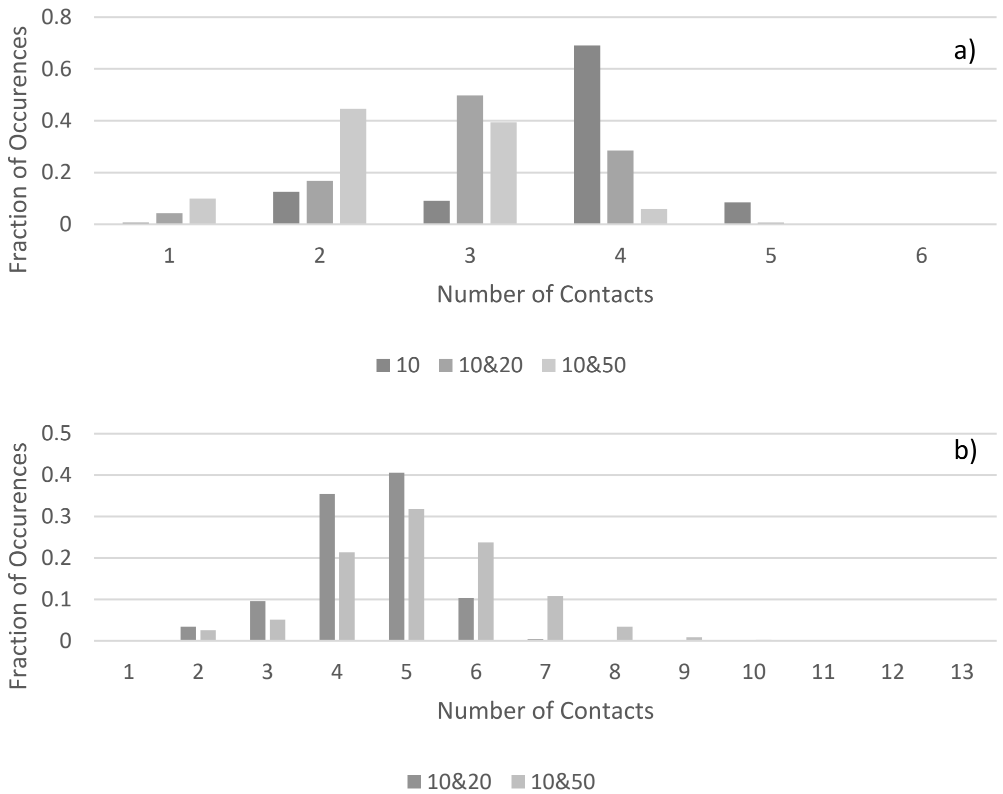


Figure 4.4: The frequency of the number of contacts per particle, differentiated by the radius of the particle, across the investigated systems. a (top): Contact number frequency of r_p = 10 in all three investigated systems. b (bottom): Contact number frequency of r_p = 20, 50 in the investigated binary systems.

When comparing the number of contacts of just the r_p = 10 (Figure 4.4a), in the systems in which they are mixed with larger particles, there is a clear difference in the number of contacts they make. In the r_p = 10, 20 systems, the peak moves to three contacts, with roughly half as many particles having four contacts, then very few one or two contact particles. This is due to the increased irregularity in the systems, pushing the particles further away from the six contact “perfect” structure of the triangular lattice. Recall that the beds created with r_p = 10 only have a fairly regular structure, whereas systems with larger particles present are more disordered (see Figure 4.1). The r_p = 10, 50 systems have similar sized peaks for both two and three contacts, with very few particles having one or four. This increase in two contact particles will be due to small particles that are resting inside a void created by larger particles, therefore having no contacts from above due the void being capped off above them.

When looking at the larger particles present within their systems (Figure 4.4b), their graphs both follow a similar pattern, with increasing frequency up to five contacts, but then decreasing from that point. The presence of the smaller particles allows for these larger particles to make many more contacts due to their increased circumference, hence the shift towards the higher number of contacts. The particles with lower numbers of contacts, such as two and three, will be due to those sitting at the bottom of the box and at the edges, as contacts with the edges of the system were not counted.

4.4 Individual Void Areas

Individual voids are determined as described in section 3.3, with loops of particles found that start and end with the same particle, as shown in Figure 4.5.

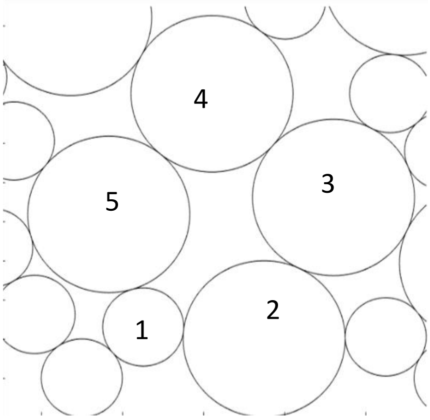


Figure 4.5: The order of particle looped through to find a shape that creates a void addition to create a void.

As shown in Table 4.2, the size of the smallest void present does not vary across the different beds, which is due to the high likelihood of a triangle of r_p = 10 existing across all the beds formed, therefore the smallest void would have little variation of the area formed by these particles.

Table 4.2: The smallest, average, and largest void areas in packed bed systems with different particle radii present.

Particle Radii	Void Areas			Average Void Area Scaled by Largest Particle Area
	Min.	Average	Max.	
10	16	78 ± 2	930	0.25
10, 20	16	190 ± 6	3300	0.15
10, 50	16	610 ± 23	10000	0.08

The average and largest void sizes increase as the width of the size distribution is increased, which is expected as there will be voids formed solely by larger particles therefore having larger gaps in between them. However, when the void areas are scaled to be proportional, by area, to the size of the largest particle area present in the system, the scaled sizes decrease with increased size distribution. This is because the voids in between the particles are proportional to particle area, however the larger particles allow the smaller particles to fit in between them filling up the gaps, whereas in the systems with more similarly sized particles, the gaps remain empty.

The sizes of the voids at the edges of the box are also included in the calculations, which contribute to the large maximum void sizes, as these voids will be bigger due to the flat surface of the box making up an edge, instead of the curved edge of a particle.

4.5 Percolation Structures

We hypothesise that the existence of percolated structures[5] in the packed beds are relevant to its structural properties. The presence of these structure may inform the strength the beds have when looking at how contacts within the system are broken, discussed in section 4.8, as well as having implications on the porosity on the bed for other investigations that could be done using this model in future research.

As discussed in section 2.3.2, for our purposes, a percolation chain is a chain of connected large particles joining the left and right sides of the box. In Figure 4.6, structures formed with varying number ratios of r_p = 20 to r_p = 10 are shown. Percolation pathways connecting large particles only from one side of the box to the other are also shown where they exist. The results of multiple runs are reported in Table 4.3. 100 runs were completed for each ratio of the different particle sizes, and the presence of percolation structures within the bed systems determined. The shortest chains, by number of particles, are counted where they exist.

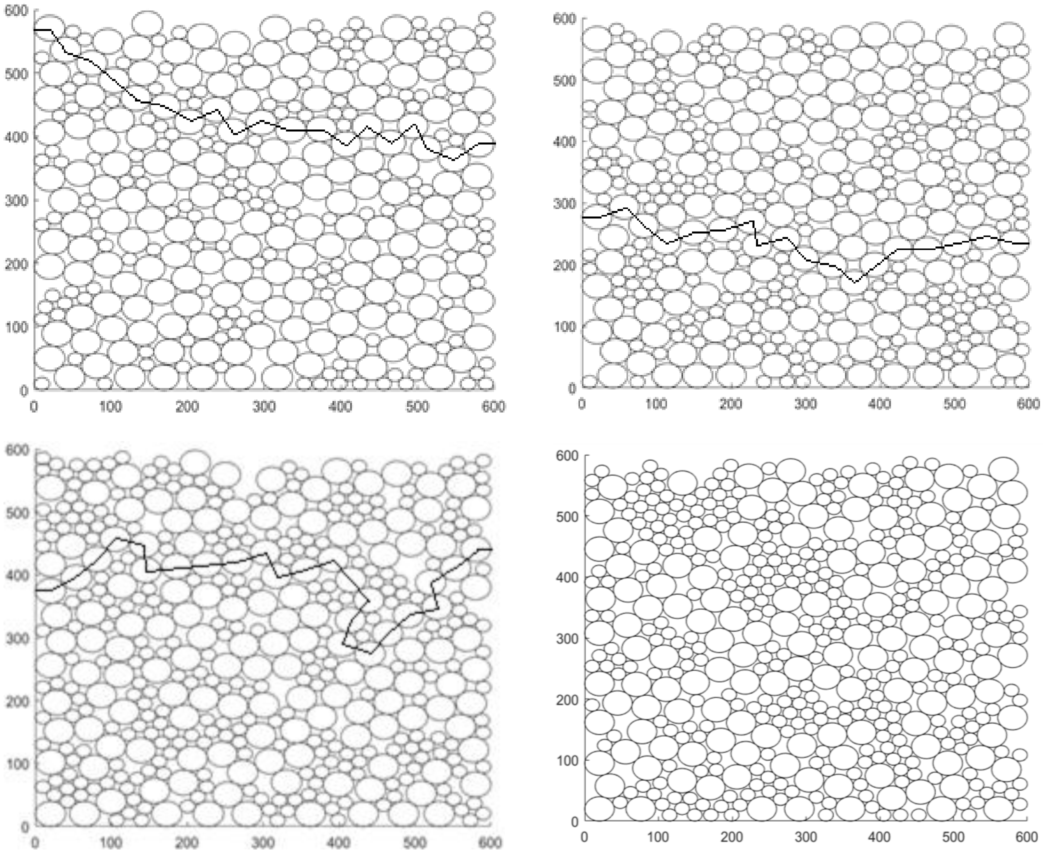


Figure 4.6: The r_p = 10, 20 system with different number ratios of large-to-small particles. Top left 1:1, top right: 1:1.5, bottom left 1:2, bottom right 1:2.5.

Table 4.3: The percentage of particle system runs that contained an edge-to-edge percolation chain and the shortest chain lengths present in systems with various number ratios of large-to-small particles in the r_p = 10, 20 system.

Particle Addition Ratio (Large:Small)	Percentage of Systems with Percolation Chains Present (%)	Avg Chains	Max Chains	Min Length	Avg Length	Max Length
1:1	98	102.73	400	18	66.28	108
1:1.5	82	113.19	319	16	41.98	81
1:2	59	62.29	300	17	32.10	66
1:2.25	33	27.27	350	17	32.69	62
1:2.5	23	17.48	200	18	30.04	59

For the purpose of determining if a percolation chain was unique, they were counted as unique chains if they were not made up of the exact same particles. For example, the blue and green chains shown in Figure 4.7 would be counted as separate chains, however the red and orange chains would be treated as the same chain and therefore only counted once.

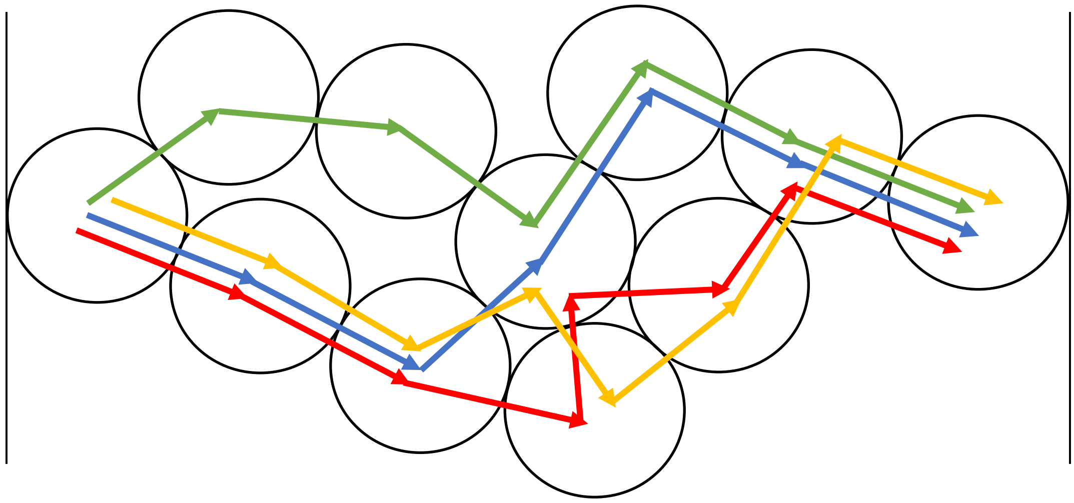


Figure 4.7: Example percolation chains showing unique chains (blue and green), and chains treated as identical (red and orange).

As shown in Table 4.3, as the ratio of larger particles within the system decreases it becomes more difficult for percolation chains to form, however even at the lower ratios there are still some chains present, indicating that we have not yet reached the percolation threshold for this system. The site percolation threshold for a regular triangular lattice is 0.5[6], and ~0.59 for a square lattice, however due to the irregularity of our structures, and the addition of smaller blocking particles, our value would likely be lower, due to these additional constraints making a chain from edge-to-edge less likely to form. The minimum possible length of a chain is 15 particles, so the minimum lengths in Table 4.3 are very close to this value, showing that almost direct chains across the bed are forming at all these ratios.

The number of chains within each system also sees an overall decrease across the systems investigated, along with the length of the chains found. This is again supported by the chains becoming more difficult to form, therefore less are able to be made. The chain lengths do start to become more consistent at the higher particle proportion ratios. This is because in the lower particle proportion ratios, longer chains are able to be formed due to the greater number of contacts between pairs of large particles. However, once there are less contacts, the pathways are more restricted and so the shorter route becomes the only route.

4.6 Finite Size Effect

Runs could also be completed using larger box sizes to investigate the finite size effect[7] within the model. As the packing of the particles will differ against the edge of the box compared to the centre of the bed, a larger box size will negate the effect of the edges so we can determine a more consistent packing fraction. Roughly 20 runs were completed on the system containing a 1:1 ratio of r_p = 10, 20 with a box four times the normal size, one of which is shown in

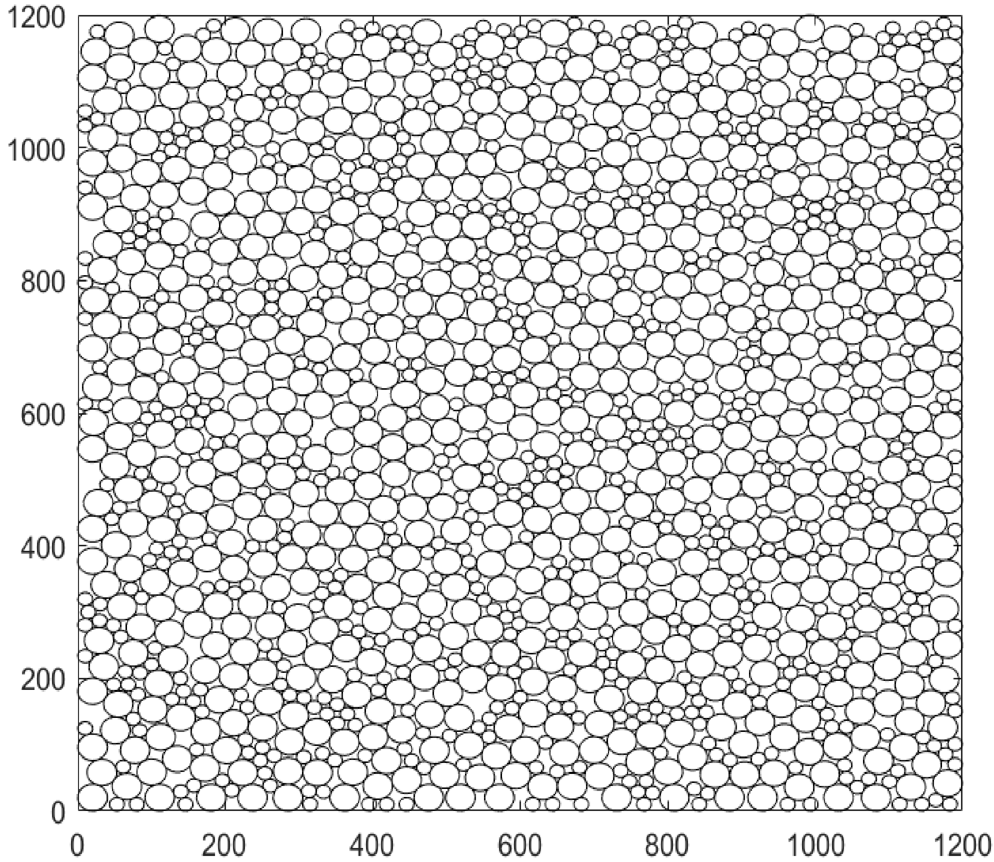


Figure 4.8: r_p = 10, 20 system at a 1:1 addition ratio in a larger box size

Figure 4.8. The average packing fraction across these systems was 0.787 ± 0.003, which is more densely packed than the systems in the smaller box sizes. There was also less variation between the larger systems than between the smaller systems. This shows initially that the properties of the systems in the smaller box sizes are getting affected by the finite size effect, therefore more runs should be completed as part of future work to fully comprehend the effect of box size on these systems and a system size where the effect is minimal.

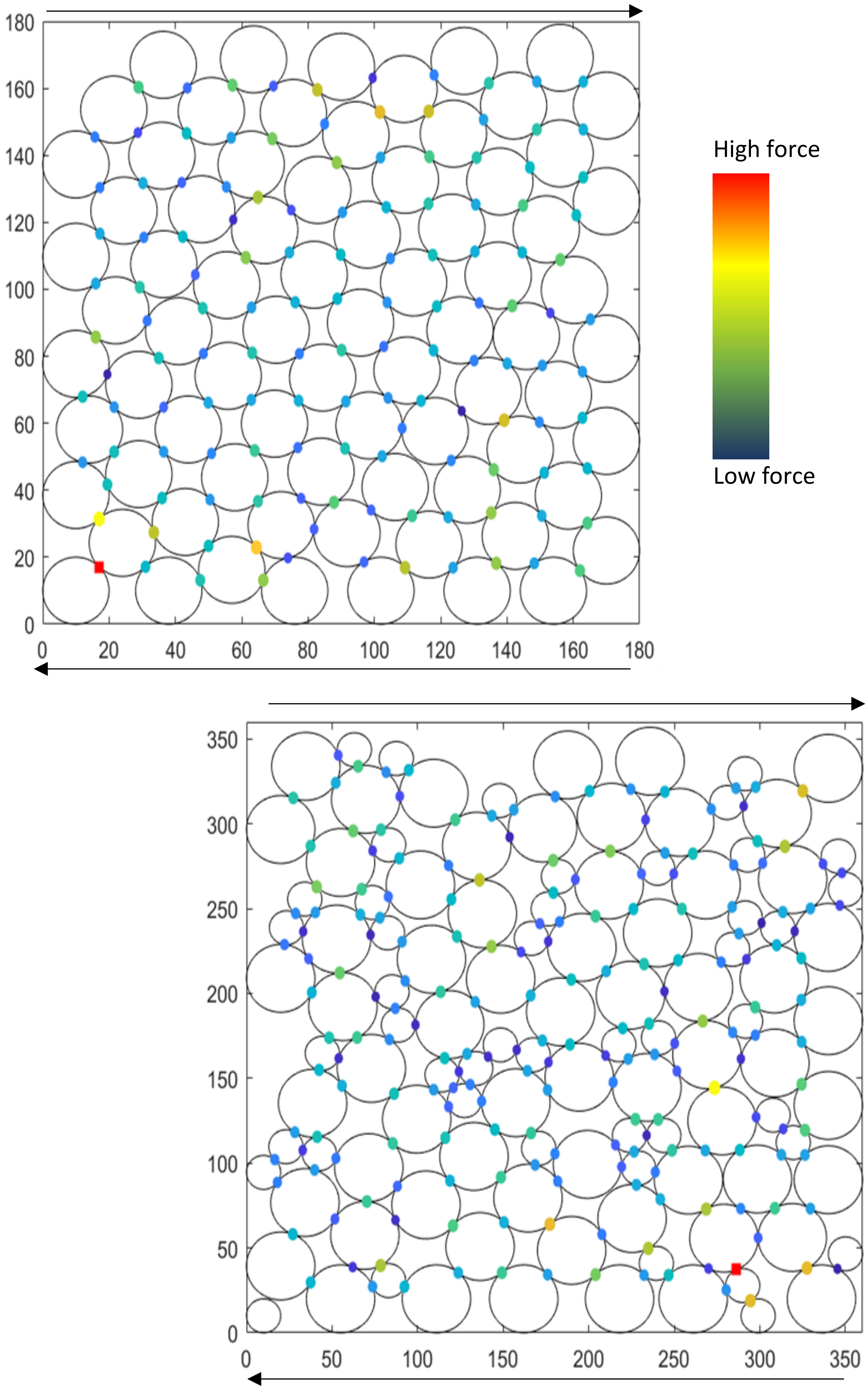
4.7 Bed Fragmentation

The algorithm discussed in this next section was produced by two MEng students at the University of Strathclyde working alongside my PhD project.[8,9] Their projects involved taking the structures produced by my model and calculating the forces present at the contact points between the particles, using the steps described in section 3.1.1.12.

Figures 4.9a and b show the forces present at the contact points between particles within four structures with different sized particles present. The structures used for these investigations are smaller than those used for the packing and percolation work discussed above as currently running these systems takes a long time due to the algorithm being written in VBA. The forces are represented by the coloured shapes at the contact points, with the darker blue colour representing a lower force, then colour shifting through green to yellow to represent a higher force present. The red square marks the contact point with the largest force present.

A shear force was applied to the systems with a 1-unit force across the top in the positive *x* direction and a 1-unit force across the bottom in the negative *x* direction. These forces were split equally between the particles on the top and bottom of each of the systems, so if there were 5 particles resting on the bottom layer of a system, 0.2-units of force would be applied to each.

a)



b)

Figure 4.9: Forces present at the contact points within packed beds of particles across four systems with different radii particles present. The arrows show the direction of the shear forces applied to the systems. a: r_p = 10. b: r_p = 10, 20.

As shown in Figure 4.9a, and mentioned previously when discussing packing fractions, the system with just r_p = 10 present is packed in a relatively ordered fashion when compared to the other systems. Because of this the forces are similar across the whole system, with forces being higher in the areas with more irregularity. The point with the highest force is present in the bottom left of the system, most likely due to the particle on the base of the bed having the shear force directly acting upon it, and only having one contact point for the force to be distributed through.

In Figure 4.9b, the system with r_p = 10, 20 present, the structure is more disordered and therefore there is less consistency between the forces present at the contact points. The highest force is again present near the base of the bed, on a particle pressed between two base layer particles and another larger particle above it. The large void to the right of the particle also means that the increased disorder of the area of the system means the force can not spread out as easily, resulting in it accumulating on the adjacent particle.

Figures 4.10a and b show the same systems as Figures 4.9a and b, however they show the order in which contacts break upon multiple runs of the algorithm. After each run is completed, the contact point with the highest force present in the system is removed from the next round of calculations, simulating the bond being broken and allowing us to investigate the cracks that could form through the structure. If this breakage resulted in a top or bottom particle that had the shear force applied to it no longer having any contacts, the force was removed from this particle and redistributed evenly between the other particles on its level.

Both of the systems show a mostly continuous chain of breakages through the system. Figures 4.10a and b both contain cracks through the systems near the top or bottom of the bed, likely due to this being close to where the shear force is being applied. The bottom of Figure 4.10a's bed is more irregular than the top therefore it is expected for it to be less stable and have higher forces present between the particles. In Figure 4.10b, the top and bottom of the bed are closer in regularity, however the bottom of the bed does contain larger voids so the packing is not as tight, resulting in higher forces there.

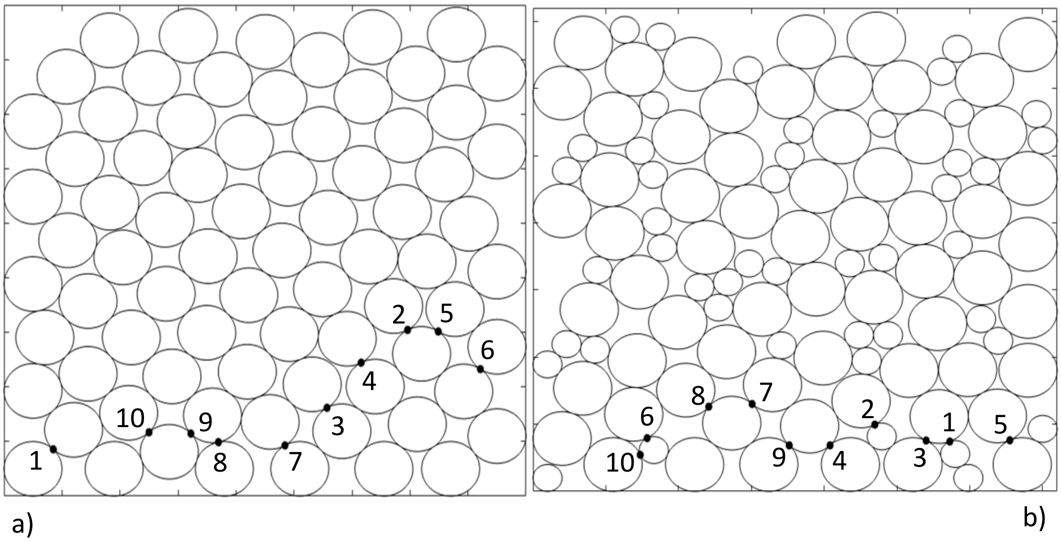


Figure 4.10: The order of contact breakage within packed beds of particles across four systems with different radii particles present. a (left): r_p = 10. b (right): r_p = 10, 20.

Figures 4.11a and 4.11b show the contact point forces present in two structures, one which contains a percolation chain (4.11a) and one which does not (4.11b). As shown previously, the forces present in the ordered areas of the systems are more consistent compared to the forces in areas with less ordered packing. The forces are also higher near the top and bottom of the systems, where the shear force is being initially applied, and then spreads out closer to the centre of the systems.

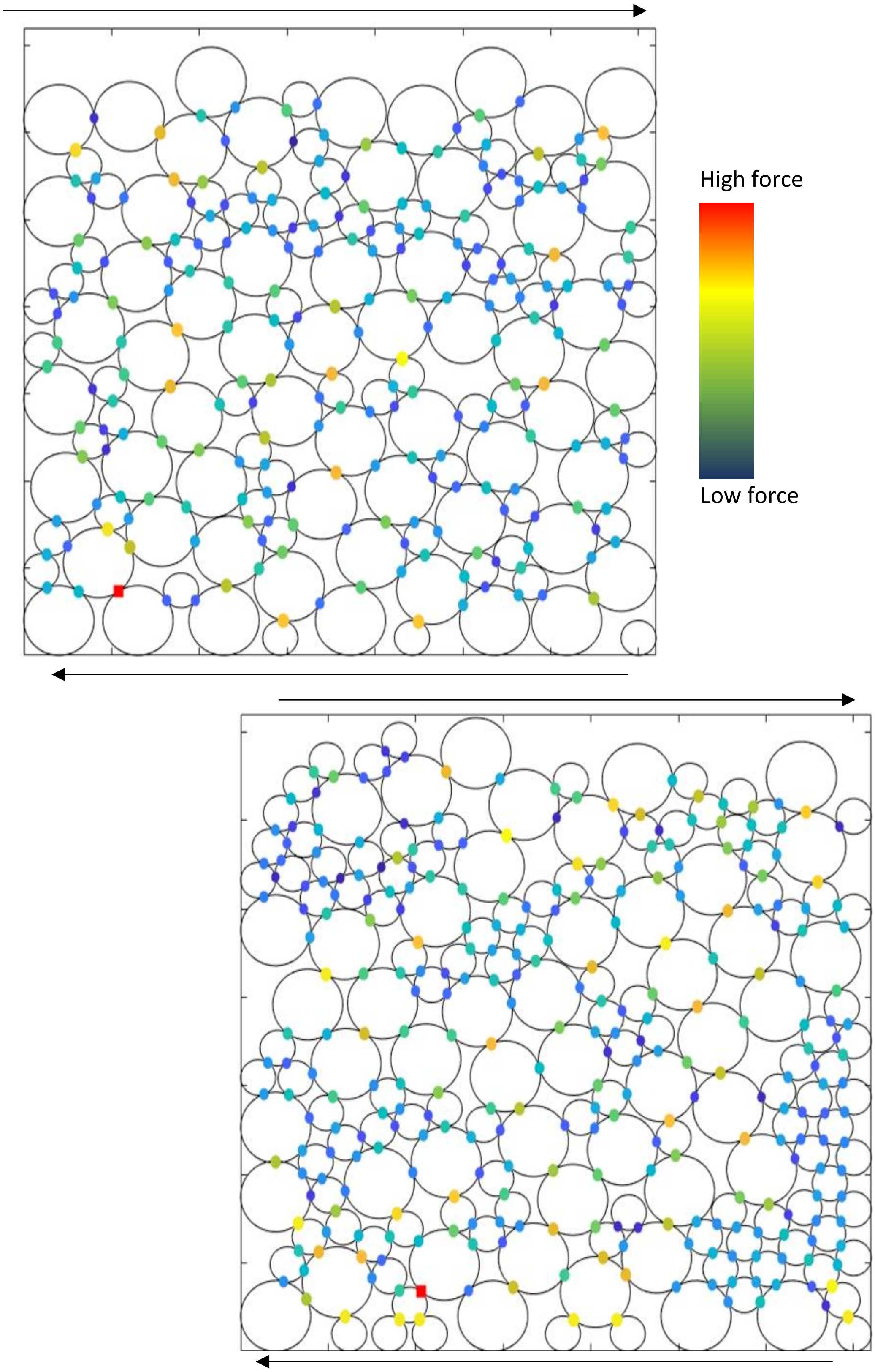


Figure 4.11: Forces present at the contact points within the r_p = 10, 20 system with different number ratios of large-to-small particles. The arrows show the direction of the shear forces applied to the systems a: 1:1, b: 1:2.5

Figure 4.11b has more ordered areas due to a higher proportion of smaller particles present allowing them to pack more tightly together, compared to Figure 4.11a which has a more equal ratio of particle sizes, causing the structure to be more irregular.

Figures 4.12a and 4.12b show the systems from Figures 4.11a and 4.11b, having gone through the same analysis as in Figures 4.9a and b. The initial contact breakages in both structures occur at the base of the beds, likely due to the proximity to the application of the force. However, once these initial contacts are broken, the next contacts to break form a chain closer to the centre of the beds. In Figure 4.12a, the breakage initially follows the percolation chain through the centre of the bed from contacts three through six. Contacts seven and eight then break the percolation chain which moved downwards, maintaining the horizontal fragmentation created by the previous contacts breaking, and ending with contact nine breaking on the left edge of the bed. The last contact then breaks as still in line with the previous breakages but back on the righthand side of the bed.

Figure 4.12b contains no percolation chain, however the fragmentation also created a chain within the centre of the bed. Compared to Figure 4.12a however, the chain breaks in smaller clusters between contacts three and four, five and six, and then seven to ten.

Both breakage chains occur in the regions of the beds that have larger voids and more irregular packing structures, as these areas are less stable within the bed due to the system finding it harder to distribute the forces evenly.

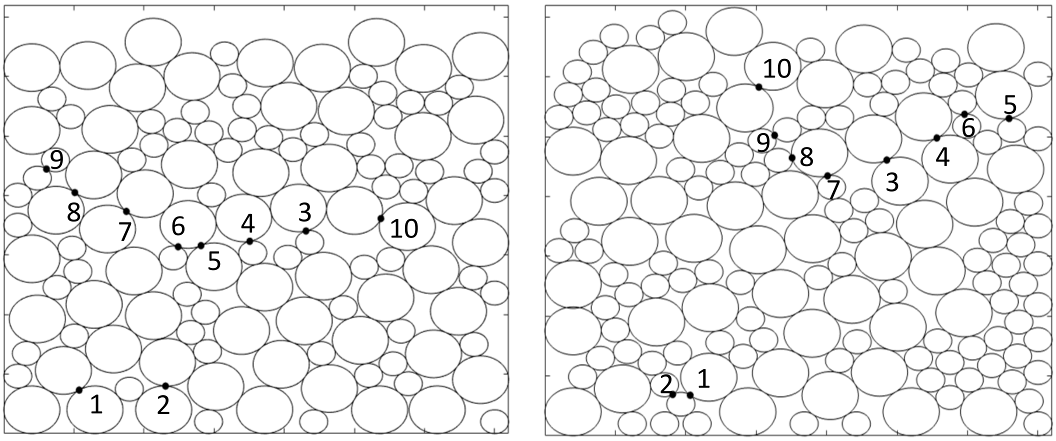


Figure 4.12: The order of contact breakage within a system with a percolation chain (a: left) and a system without a percolation chain (b: right).

4.8 Summary and Conclusions

When comparing the packing fractions for my simulated systems to calculated values from literature, it was found that my values fell in the expected range being higher than RSA packing fractions, due to the addition of gravity simulated in the system, however they had a lower packing fraction than the calculated maximum, as there is a degree of randomness stemming from the initial base layer of particles that causes the system to shift out of a perfectly ordered lattice. There is an increase in the average packing fraction as the particle size distribution is increased, due to the ability of the smaller particles to fit inside the voids created by the larger particles, thereby decreasing the void area of the system.

When investigating the number of contacts that each particle had, the starting assumption was that in the single sized particle system, each particle would have four contacts, two from particles being rested on, and two from particles resting on it. This was found to be the case, with the majority of particles having four contacts, however there were particles with fewer contacts, likely due to the finite size effect in the small system, and some particles with more contacts, in areas where the particles were packed closer together.

The introduction of larger particles within the binary systems, caused a decrease in the number of contacts per small particle, as the systems became more irregular. In the r_p = 10, 50 systems, the majority of small particles had two or three contacts, as they were likely placed in between two larger particles, with potentially one more small particle, before the void was capped off. The larger particles in the binary systems have a similar contact graph to the single sized system, with the peaks being around four to six particles. This is shifted upwards from the four shown in the single sized particle system due to the additional contacts with smaller particles in the voids between the larger particles.

When calculating the sizes of the individual void areas between particles, it was found that when the average void area across a system was scaled by the largest particle area present within that system, it decreased when the particle size distribution increased. This is expected because, as previously discussed, the smaller particles in these systems are able to fill in void spaces between larger particles, so reducing the sizes of the voids within the system.

When looking for the presence of percolation chains, in this thesis defined as a chain of large particles connecting the left- and right-hand side of the box, it was found, as expected that increasing the number of small particles within the system would disrupt the formation of percolation chains. There was always a system that did not contain a percolation chain, even at a ratio of 1:1 large:small particles, and as the ratio of smaller particles increased the percentage of systems that contained a percolation chained decreased massively from 98% to 23% across the systems tested. The average number of chains per system, and the average length of chain per system also decreased as the number of small particles in a system increased.

When looking at the forces present at the contact points between particles, it was found that the forces were much more evenly spread in the system with the same sized particles compared to the binary particle system. This is likely due to the packing in the binary system being less uniform, therefore creating a less stable structure, with areas of higher stress within the system where contacts are more likely to break. The order of breakage contacts do follow chains in both same sized and binary particle systems, as once a contact breaks the contacts around it are put under more stress as there is one less contact for the forces to be shared between.

When investigating the contact breakage of systems that did contain a percolation chain, it was found that the order of contact breakage followed the percolation chain through the system almost fully. However, more investigation will need to be done into these systems to determine if there is a link between the presence of percolation chains and contact breakage paths as this initial system could be due to random chance.

The 2D algorithm was created as an initial starting point due to the simplicity of creating it, and it has been shown to be capable of producing realistic systems of packed bed particles formed under gravity. It has also shown the possibility of different applications it can be applied to, such as further investigation of how the beds break apart and how the existence of percolation structures affects this. However, it was intended to be a stepping stone into the 3D algorithm, the results of which are discussed in the next chapter.

4.9 References

(1) Bédaride N. and Fernique, T., Density of Binary Disc Packings: The Nine Compact Packings. *Discrete and Computational Geometry,* **2022**, *67*, 787-810.

(2) Zhang, G. and Torquato, S., Precise algorithm to generate random sequential addition of hard hyperspheres at saturation. *Phys. Rev. E*, **2013**, *88*(5), 053312.

(3) Andrienko, Y., Brilliantov, N. & Kurths, J. Complexity of two-dimensional patterns. *Eur. Phys. J. B,* **2000**, *15*(3), 539–546.

(4) H. Chang and L. Wang, **2010**. A Simple Proof of Thu''s Theorem on Circle Packing. arXiv:1009.4322

(5) Sahimi, M., *Applications of percolation theory*. London: Taylor & Francis. **1994.**

(6) Stauffer, D. and Aharony, A., *Introduction to Percolation Theory*. 2nd ed. Boca Raton: Taylor & Francis Group, **2014**.

(7) Wedekind, J., Reguera, D. and Strey, R., Finite-size effects in simulations of nucleation. *J. Chem. Phys*, **2006**, *125*(21), p.214505.

(8) L. Cartwright, "Modelling Stress and Fracture in Agglomerated Structures," University of Strathclyde, Glasgow, 2021.

(9) L. Convery, "Modelling the Fracture and Breakage of Agglomerates Under an Applied Shear Force," University of Strathclyde, Glasgow, 2022.

5 Results and Discussion – 3D Algorithm

This chapter discusses the systems created by the 3D algorithm discussed in Chapter 2. This chapter investigates the different properties of these systems in contrast with similar systems in two dimensions. We start by looking at the packing of the systems created, before investigating the sizes of the voids and number of contacts each particle has.

5.1 Visual Inspection

Figure 5.1 shows some examples of 3D packed beds, with different radii of particles present, that were created using the model presented in Section 2.3.

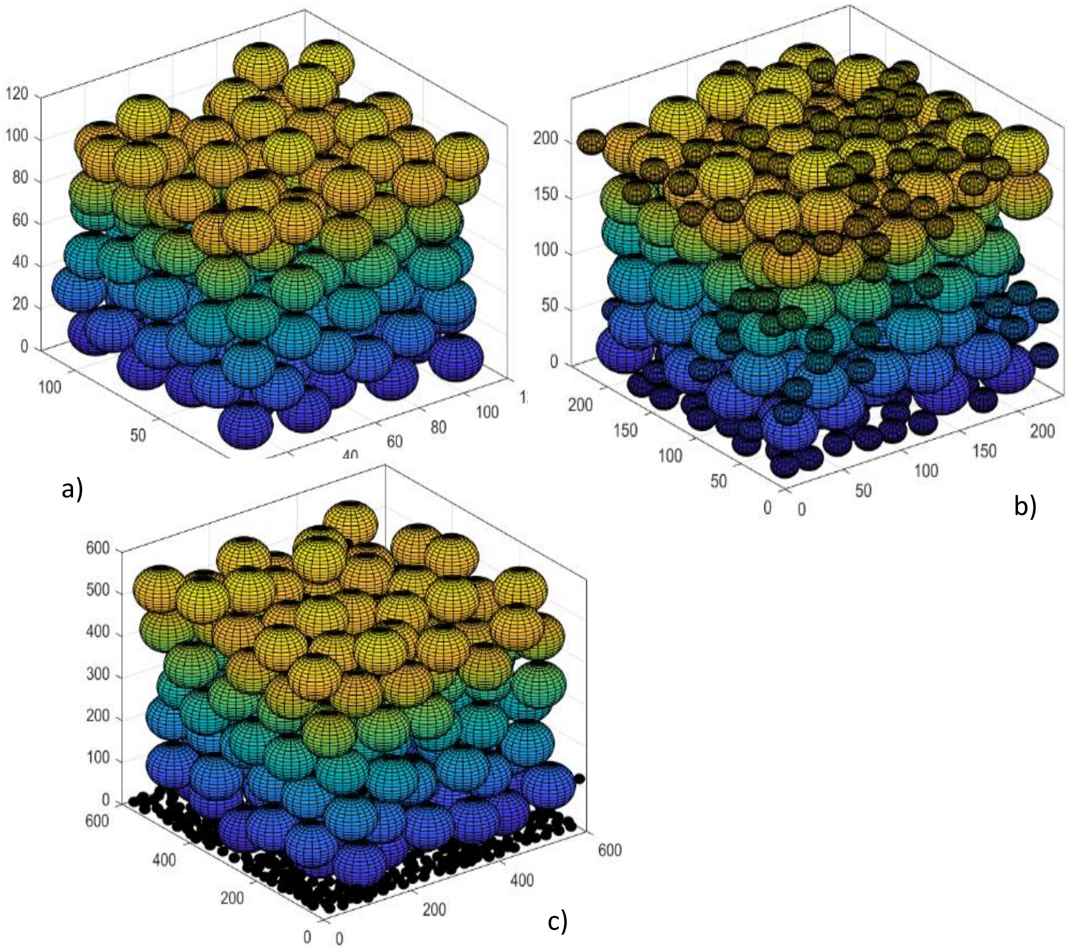


Figure 5.1: Packed beds of 3D particles across three systems with different radii particles present. a (top left): r_p = 10. b (top right): r_p = 10, 20. c (bottom middle): r_p = 10, 50. Note that the size of the bounding box increases with the largest particle dimension.

The 3D systems have much lower packing fractions when compared to the 2D systems, which is expected when comparing the maximum possible packing fractions, ~0.9 in 2D and ~0.7 in 3D, as there is an extra dimension leading to more variability in particle placements. This makes the chance of an ordered system where each sphere has twelve contacts extremely unlikely, therefore an “ordered” system for our cases would be one where each sphere has six contacts, with both three above and three below. As even the systems with spheres of the same size present are disordered, the addition of smaller particles does not affect the order of the system significantly due to arrangement of the particles in the first layer. However, we can see, in contrast to the 2D systems, that particles which are sufficiently small are now able to fall through the gaps between the larger particles. In Figure 5.1b, the small particles are not small enough to fit between most of the gaps between the larger particles and therefore there are still small particles throughout the height of the system. However, in Figure 5.1c, the small particles are now small enough compared to the large particles to be able to fall through the majority of the gaps between them, resulting in a collection of small particles at the base of the bed. This is a significant observation since it is consistent with the widely held belief that the presence of fine particles which may be transported through a bed of particles are responsible for significant increases in the filter cake resistance when filtering particle suspensions with a wide particle size distribution.[1]

5.2 Packing Fractions

The packing fractions determined for the 3D systems follow a different pattern to that of the 2D systems. As shown in Table 5.1, the systems become more packed with the introduction of a larger sized particle, with the system that contained r_p = 10, 20 particles having a larger packing fraction than the other systems. As with the 2D systems, the smaller particles are able to fill in the voids between the larger particles when they are present, however the 1:1 addition ratio of the particles sizes means that whilst the r_p = 10, 50 systems could have a higher packing fraction if the voids were filled with smaller particles, there are not enough small particles placed within the systems to fill the voids, thus leaving the r_p = 10, 20 systems with a higher packing fraction.

Table 5.1: The Packing Fractions in 500 generated 3D Packed Bed Systems with different particle radii present.

	Packing Fraction		
Particle Radii	Minimum	Average	Maximum
10	0.434	0.460 ± 0.007	0.475
10, 20	0.480	0.497 ± 0.006	0.513
10, 50	0.452	0.468 ± 0.006	0.486

These values are lower than the highest packing fractions that have been calculated in systems of same sized spheres. There are two lattices that can occur to achieve the highest packing fraction[2], which is $\pi/_{3\sqrt{2}} \approx 0.74048$[3]. These two lattices, as seen earlier in Figure 2.3, are face-centred cubic (FCC) and hexagonal close-packed (HCP). It has been found that the highest packing fraction in 3D binary sphere packings, such as our r_p = 10, 50 system, in which the smaller particles are able to pass between the voids formed by the larger particles is 0.8617.[4]

Other examples of packing types and their maximum densities are: random close packing, 0.6400[5]; the tetrahedral lattice, $\pi\sqrt{3}/_{16} \approx 0.3041$[6]; and the loosest possible density that has been found is 0.0555[7]. Our values fit between these as expected, as they are lower than the more packed systems due to our inherent randomness but more packed than the more irregular systems due to the presence of the simulated gravity forcing particles downwards to pack more tightly. Another reason behind our systems having a lower packing fraction than the higher density packing methods is due to the small box size being used for our systems, resulting in significant edge effects reducing the packing fraction.

5.3 Number of Contacts

In the FCC and HCP lattices discussed above, the expected number of contacts for each sphere is twelve, with three below, six on the same plane, and three above. However even the slightest irregularity causes the spheres on the same plane to be further away and no longer in contact with each other. Therefore, the number of contacts that each sphere would have in a regularly structured system would be six, accounting for the three touching spheres above and below.

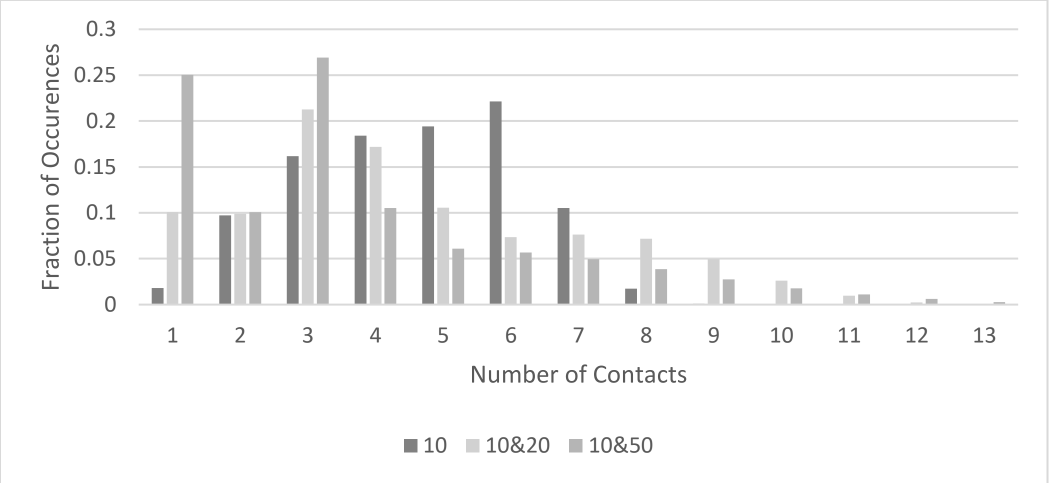


Figure 5.2: The frequency of the number of contacts per particle has across the investigated 3D systems.

As seen in Figure 5.2, the systems containing only r_p = 10 do show the most frequent contact number is six, however not by a large margin. Due to the large amount of disorder in these systems, the number of contacts ranges all the way from one to ten contacts in the single particle size bed. Whilst there are some particles with contact numbers close to the FCC and HCP lattice value of 12, they are extremely outnumbered by the number of particles with 6 contacts or less. This shows how the packing of these systems is far away from the ordered packing of the HCP and FCC lattices, due to the randomness of the placement of the particles.

The beds with different sized particles present show a maximum at three contacts, also with high occurrences of four to seven contacts. These are still around the expected value of six, with the lower ones being particles in contact with the edge of the box, due to the finite size effect.[8] The lower end of the contact values is also due to particles that are in contact with the edges of the box, as contacts between particle and boundary are not counted, as well as smaller particles resting inside voids capped by larger particles. As part of future work, discussed further in Chapter 8, future investigations would go into the finite size effect so that these data points will not impact the averages as much.

As shown in Figure 5.3a, there is a large variance between the number of contacts each of the smaller particles across the three investigated systems has. The increased number of particles with one contact in the r_p = 10, 20 and r_p = 10, 50 systems, is due to the higher box area, and therefore more small particles falling to the bottom of the box, and only having a single contact with a particle resting above them. The large number of small particles with three contacts is due to a small particle resting on three larger particles with the void then capped above by another large particle, not allowing the smaller particle now trapped inside the void to gain any more contacts.

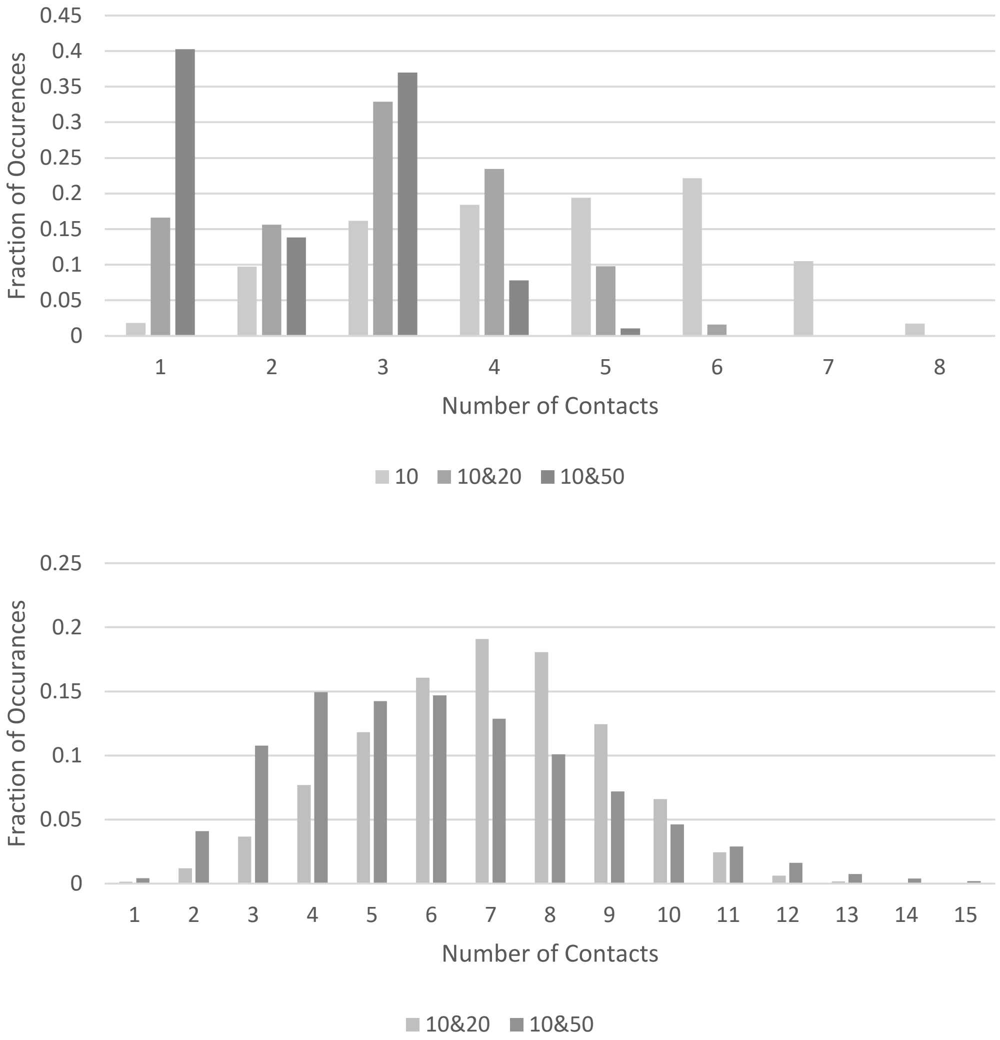


Figure 5.3: The frequency of the number of contacts per particle, differentiated by the radius of the particle, across the investigated 3D systems. a (top): Contact number frequency of r_p = 10 in all three invesitated systems. b (bottom): Contact number frequency of r_p = 20, 50 in the investigated binary systems.

Figure 5.3b shows the difference between the number of contacts of the larger particles in the r_p = 10, 20 and r_p = 10, 50 systems. The r_p = 10, 50 data in Figure 5.3b is similar to the r_p = 10 data in Figure 5.3a, as similar to the 2D systems they pack similarly, however the r_p = 10, 50 graph has a slower decline at the higher end of the number of contacts due to the smaller particles now present that will also be resting upon them.

These graphs give more information about my systems, and is another metric by which the simulated systems can be compared when subjected to shear forces, which would hopefully be performed as part of future research using this model.

5.4 3D Percolation Structures

We also investigated the presence of percolation structures in the 3D systems. With the addition of the third dimension, chains spanning the box in either the x- or z-direction are sought using the same method as discussed in Section 2.2.3 for the 2D structures.

Table 5.2: Data on Percolation Structures in 100 generated 3D structures containing r_p = 10, 20 in various proportions

Particle Proportion (Large:Small)	Systems with Percolation Chains Present (%)	Minimum Number of Percolation Chains Present	Average Number of Percolation Chains Present	Maximum Number of Percolation Chains Present
1:1	100	7	15.46 ± 4.94	26
1:2	100	2	11.04 ± 5.38	27
1:3	98	0	6.91 ± 5.17	26
1:4	84	0	4.36 ± 3.26	14
1:5	59	0	2.01 ± 3.04	19
1:6	36	0	0.89 ± 1.82	9

The 3D data shown in Table 5.2 show the same pattern as the 2D data in Table 4.3, with a higher frequency of percolation structures present when the number of large and small particle present are similar. However, there are still many more percolation structures present at higher ratios in the 3D systems compared to the 2D systems, with only 59% of systems containing a percolation structure in the 2D system with a ratio of 1:2 large:small particles, but the 3D system with the same ratio having 100% percolation chain presence. This is because in 3D the particles tend to have more contacts, giving more options for the larger particles to connect to each other across the system. As shown, even at a ratio of 1:6 the percolation threshold has not been found and more percolation chains are being found than in the 2D system with a third of the ratio. Another factor is that whilst the radius ratio is the same in both the 2D and 3D systems, the area/volume ratio is not, as in the 2D systems the 10:20 area ratio is also 1:4, however in 3D the volume ratio between radius 10:20 particles is 1:8, giving them much more surface area to make contacts with other particles in the system, and thus form percolation chains.

Table 5.3: The minimum, average and maximum lengths of percolation chains in 100 generated 3D structures containing r_p = 10, 20 in various proportions

Particle Proportion (Large:Small)	Minimum Length of Percolation Chain Present	Average Length of Percolation Chain Present	Maximum Length of Percolation Chain Present
1:1	7	7.56 ± 0.94	18
1:2	7	7.79 ± 1.02	16
1:3	7	7.94 ± 1.31	15
1:4	7	7.80 ± 1.11	17
1:5	7	8.14 ± 1.18	15
1:6	7	8.02 ± 0.98	12

Again, the minimum length of chain is close to the minimum possible but slightly above, with the 3D box dimensions being 6 particle diameters. The average chain lengths across the different ratios are consistent, likely due to the smaller size of the box resulting in much longer chains being unable to form. The maximum chain lengths are also relatively consistent compared to the 2D data, still with a small decline, again likely due to the comparatively smaller box size. Producing more systems in larger boxes for all these tests on a larger scale is the next step for this area of the project, and is discussed more in the Future Work section, as further investigation can then be undertaken into how these percolation structures may reflect fracture patterns within the systems.

5.5 Summary and Conclusions

When investigating the packing fractions of the 3D systems created by my model, I found that whilst they followed the same pattern as the 2D systems, they overall had much lower packing fractions, with a difference of about 0.3. My systems packing fraction values did rest between the precalculated minimum and maximum packing fraction values for single sized and binary particle systems, showing that initially these values are realistic.

The 3D contact graphs also show similar patterns to the 2D contact graphs, however the 3D graphs are much more affected by the finite size affect, as well as the ability of the small particles to fall through voids to the bottom of the box. This results in a large number of small particles with very few contacts, especially in the binary r_p = 10, 50 systems. The single sized r_p = 10 system would have each particle having 12 contacts if it was perfectly packing in a FCC or HCP lattice, however as previously stated we expect our particles to have 6 contacts. There is a peak at this value, however the lower contact number values occur almost as frequently, showing that the structure is quite far removed from the ordered lattices, especially with very few particles having above 7 contacts. The larger particles that have more contacts will also be due to the small particles resting around them, rather than the packing of the systems shifting towards one of the ordered lattices.

When investigating the presence of percolation chains within the 3D structures, it was found that they are much more prevalent in 3D systems than in 3D, likely due to the increase in contact area available, thus more contacts being formed, in 3D systems. When comparing across the 3D systems, the expected pattern of finding less percolation chains occurring the more smaller particles are in the system is found, with the minimum, average and maximum number of chains per system decreasing as well as the average length of a chain increasing as it becomes harder to make a direct chain across the system.

The 3D algorithm that has been produced is able to simulate more realistic systems that the 2D algorithm, purely given the extra dimension that exists in laboratory experiments. The 3D algorithm has also been shown to replicate phenomena experienced in laboratory experiments, such as smaller particles filtering to the bottom of a system, through the gaps made by larger particles. Given the various parameters investigated here, this model can hopefully be used in future research to see the effect on these systems when the parameters are further varied.

5.6 References

(1) Hennemann, M., Gastl, M., and Becker, T., Influence of particle size uniformity on the filter cake resistance of physically and chemically modified fine particles, *Separation and Purification Technology*, **2021,** *272*, 118966.

(2) Gauss, C.F., ‘Besprechung des Buchs von L.A. Seeber: Untersuchungen über die Eigenschaften der109ositiven ternären quadratischen Formen usw.’, *Göttingische Gelehrte Anzeigen*, **1831**, *2*, 188–196.

(3) Schoenberg, I.J. and Steinhaus, H. Mathematical Snapshots, *Coll. Math. J.*, **1986**.

(4) De Laat, D., De Oliveira Filho, F., and Vallentin, F., Upper bounds for packings of spheres of several radii, *Forum of Mathematics, Sigma,* **2014,** *2*, E23.

(5) Jaeger, H.M. and Nagel, S.R. ‘Physics of Granular State’, Science, **1992**, *20*, 1523–1531.

(6) Hilbert, D. and Cohn-Vossen, S. ‘Geometry and the Imagination’, *Phys. Today*, **1953,** *6*(5), 19.

(7) Gardner, M. ‘New Mathematical Diversions From Scientific American’, **1966,** 88.

(8) Wedekind, J., Reguera, D. and Strey, R., Finite-size effects in simulations of nucleation. *J. Chem. Phys*, **2006**, *125*(21), 214505.

6 Non-Spherical Particle Chains

As discussed in Section 2.3.6, when we are modelling chain particles, we are doing so by attaching multiple circular particles together to form the chain. This decreases the complexity of the algorithm as it can be created by building upon the previous 2D algorithm, however this approach does mean the chain particles are not accurate to how a rectangular or needle-like structures would pack, due to the ridges present in our chain structures. The following algorithm was created as an extension for the 2D algorithm due to its simplicity. A 3D version of the algorithm has also been started but is not complete due to time constraints.

6.1 Initial Code Edits

A new section was added into the algorithm that took place immediately after a particle was added into the system. Instead of looping back to the start to place another particle, the model created a localised contour plot based around the most recent particle, which followed its edge. This places the new particle with its edge on the centre of the previous particle, overlapping with it to form a chain, but still being unable to overlap with other particles.

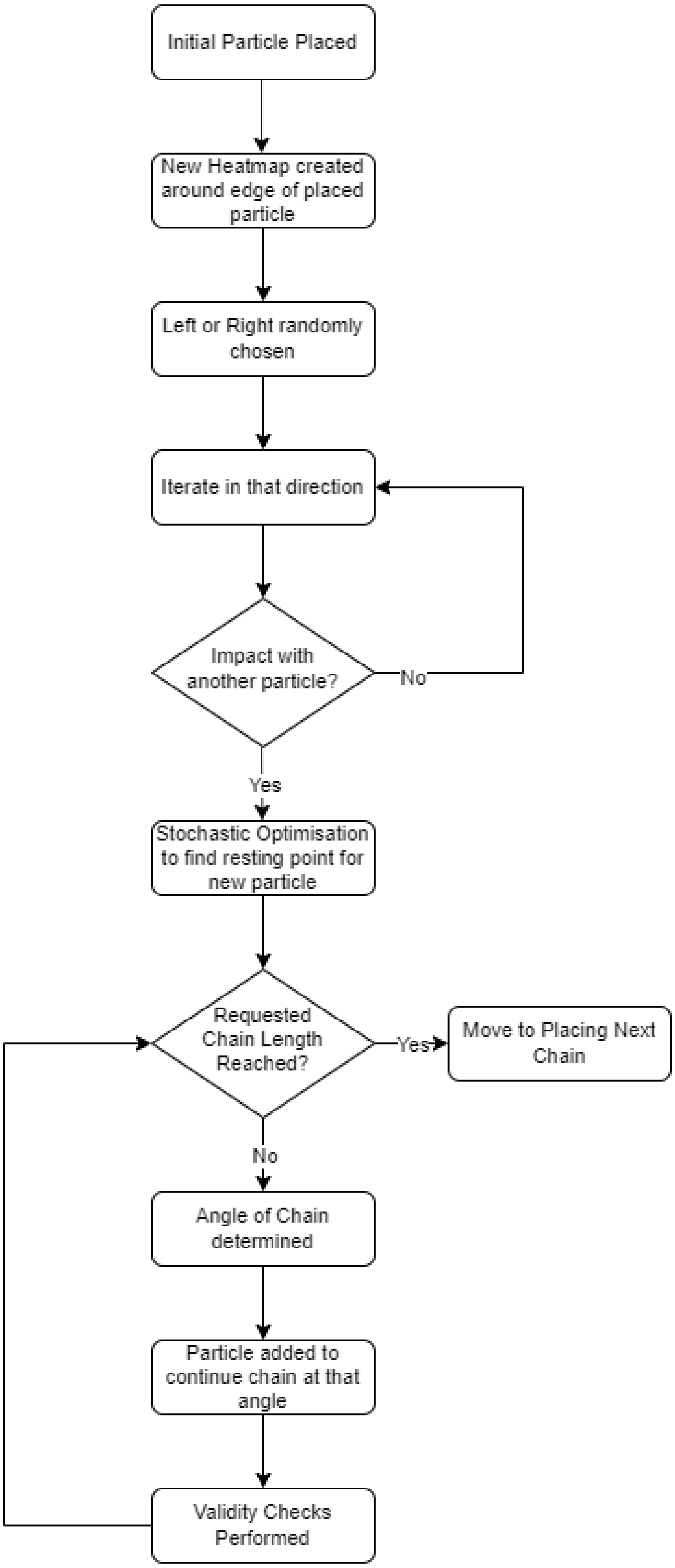


Figure 6.1: Flowchart showing the stages of adding particles to form a chain particle from the initial particle

Once this contour plot is formed, the model then randomly picks a direction for the next particle in the chain to fall to from the centre of the particle, as it is assumed the chain is falling vertically and then tilting to one side upon impact. It then moves along the contour plot as before until it impacts with another particle. It then follows the algorithms previous path of particle impact, where the new particle is being balanced on one particle and joined with another, using stochastic optimisation to determine the real coordinates for its final position.

For a system of chains containing only two particles, the model would then loop back to the start as before and repeat these steps. However, for chains containing three or more particles, a new step is added.

As the chains direction has been determined by the first two particles, the contour plot is no longer required to place further particles in the chain. Instead, the radius of the particles and the direction that they are being placed in can be used to simply place the rest of the particles in the chain along its line.

However, as each particle in the chain is placed, it checks that it is not overlapping with any other particles or outside the box. In either of these cases, the offending particle is moved to be balanced on what it is impacting, as shown in Figure 6.2, whilst still being the correct distance away from the original particle in the chain. The particles in between these two on the chain are then readjusted to ensure they are all connected and in a straight line. This process is looped until the chain is the requested length of particles, and then the model loops back to the very start to place a new chain.

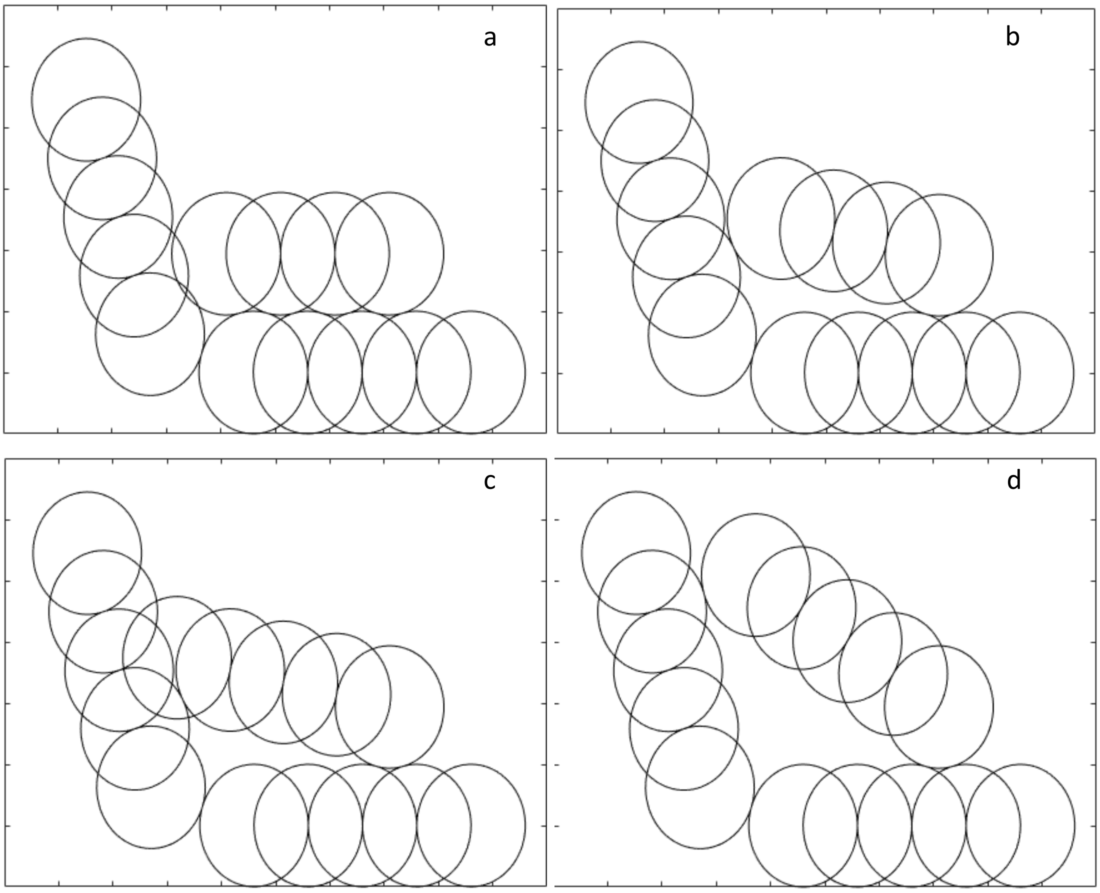


Figure 6.2: Steps taken to adjust a growing chain to balance on an overlapping chain in the event the growth of the new chain causes an overlap with an old chain. A) New chain (currently 4 particles) overlapping with a previously placed chain b) New chain readjusted to be balancing instead of overlapping c) New chain (now 5 particles) overlapping with a previously placed chain d) New chain readjusted to be balancing instead of overlapping.

6.2 Chain Particle Complexities

As the length of the chain increased, the complexity of the systems being formed also increased, with new possible balancing possibilities between particles becoming possible.

Due to the first particle being placed on its own, there was now the possibility of a particle being placed in a situation where a chain could not grow from it.

To combat this, in this instance, the particle would slide back along the path it took to get to its final position, until a chain was able to be formed. This then ran in to the issue that previously the first particle being placed needed to be balancing on two other particles, however in these situations, it is possible, and indeed correct, for the initial particle to be only in contact with one particle, and then be balanced by another contact further along the chain. Therefore, once the particle has determined it needed to slide upwards from a trapped position, some additional leeway was added to allow the first particle to have only one contact, provided the following conditions were true:

a) The base particle of chain had a contact on the opposite side to the direction the particle was leaning, i.e. if the particle is leaning right, the contact is on the left.

b) The contact that is furthest from the initial particle is also beyond the centre point of the chain.

c) There is at least one contact point, not on the initial particle, that is on the underside of the particle.

With these additional conditions, stable chain structures were able to be produced.

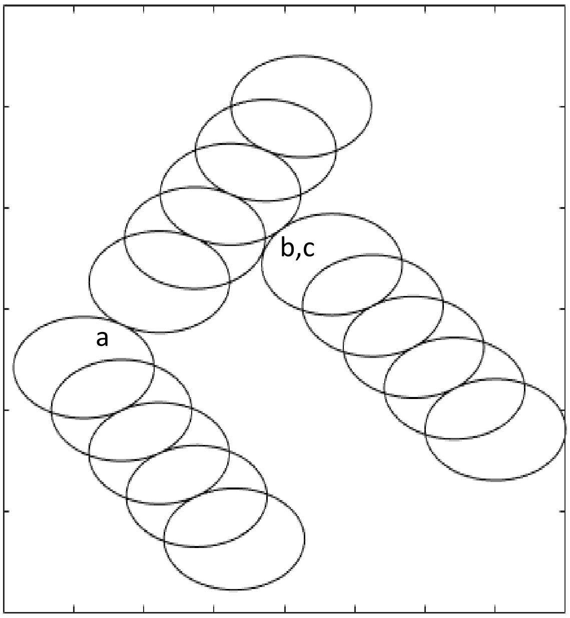


Figure 6.3: Chains labelled with the conditions a, b and c listed above that must be fulfilled to allow the placement of a chain with only have one contact at its base

6.3 Results

All of the 500 runs completed for each of the systems, listed below, using the 2D chain algorithm were performed on ARCHIE-WeSt. The ratio of addition for each different chain or particle type in these systems is 1:1. n_p refers to the number of particles that made up an individual chain.

Each of these systems had its packing fraction calculated and the angle of each chain was determined.

Systems investigated:

- A) r_p = 10, n_p = 2
- B) r_p = 10, n_p = 3
- C) r_p = 10, n_p = 4
- D) r_p = 10, n_p = 5

- E) r_p = 10, n_p = 5 and r_p = 10, n_p = 1
- F) r_p = 10, n_p = 5 and r_p = 5, n_p = 1
- G) r_p = 10, n_p = 5 and r_p = 5, n_p = 5

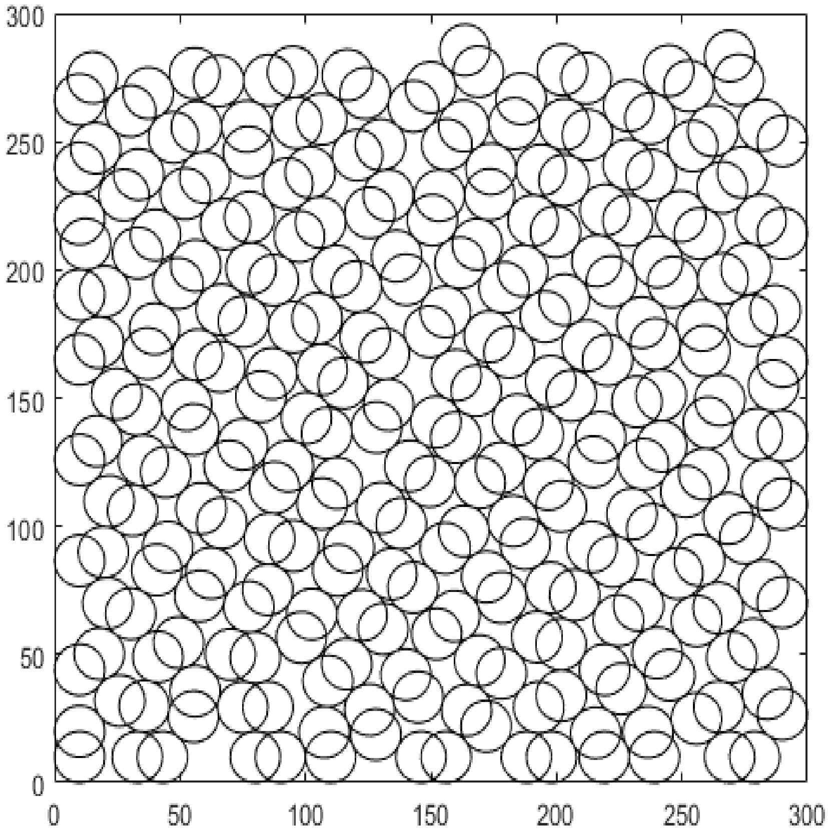
6.3.1 Visual Inspection

Examples of each of the different systems produced, A to G, are shown in Figures 6.4, 6.5 and 6.6.

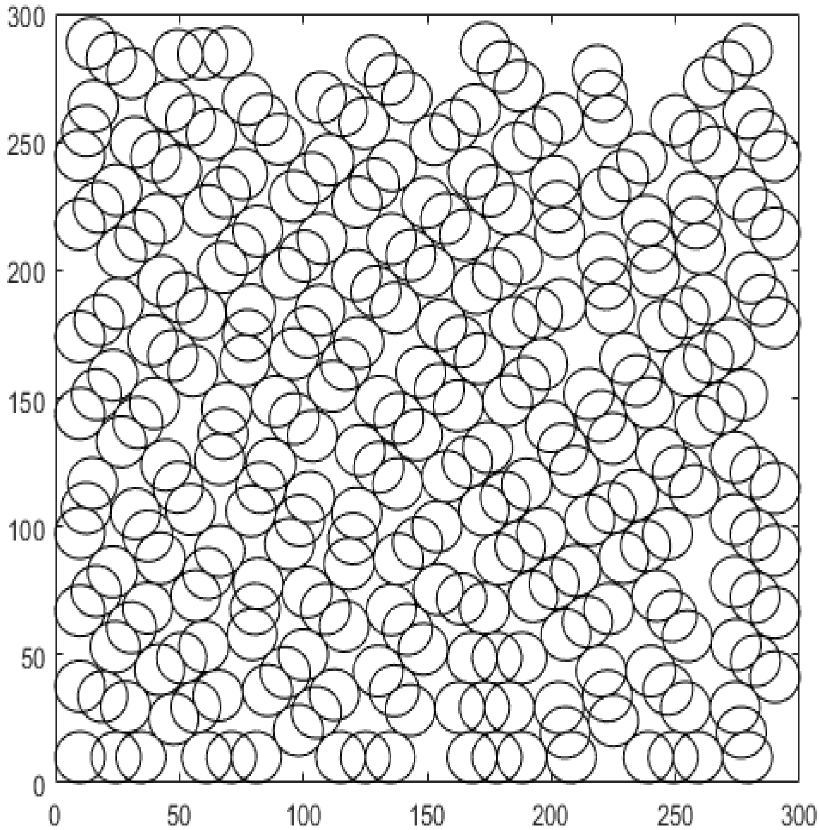
Figure 6.4 shows systems A to D, where the n_p increases from system to system but remains consistent within each system. In each of these systems, once a chain has fallen, it often results in the chains placed above it adopting the same angle, creating sections within the system with chains stacked together, an example of which in the System D example at x = 300 and y = ~150 to y = ~350. As the chain length gets longer, these groups of chains at the same angle take up much more of the systems. These clusters are broken up when a chain is placed to the side of it but leans over enough to cause the next chain that would want to join the cluster to lean differently.

Some perfectly vertical chains are present leaning up against the wall of the box in the base layer of particles, which is a very unstable position. This is because when I added a section of code to the algorithm to favour falling away from the edge of the box if a chain found itself directly up against it, I forgot to also add that section of code to the part of the algorithm that placed the base layer of chain particles. Therefore these vertical chains will only appear in the bottom left and right corner of the box, using this algorithm, but this issue would be easily solved in a future version.

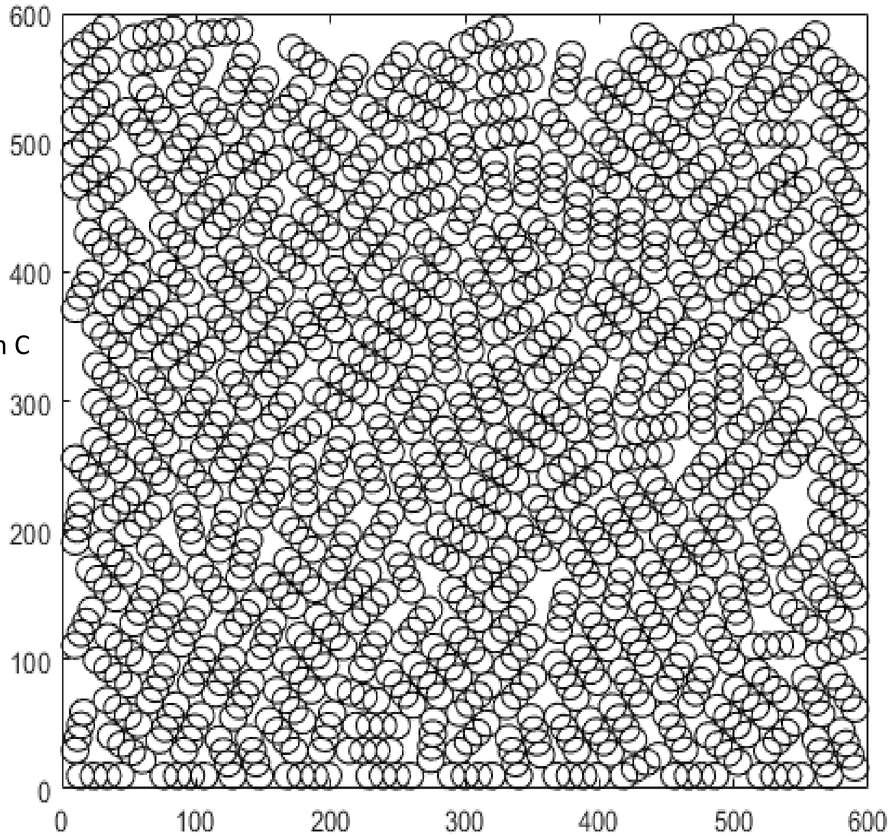
System A



System B



System C



System D

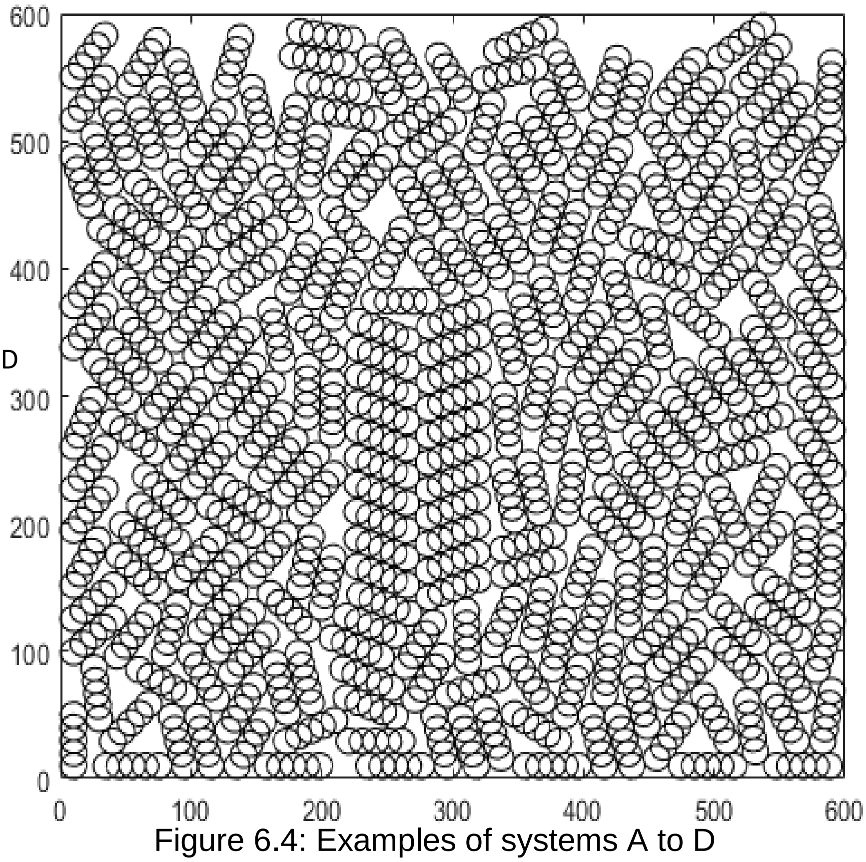


Figure 6.4: Examples of systems A to D

System E

System F

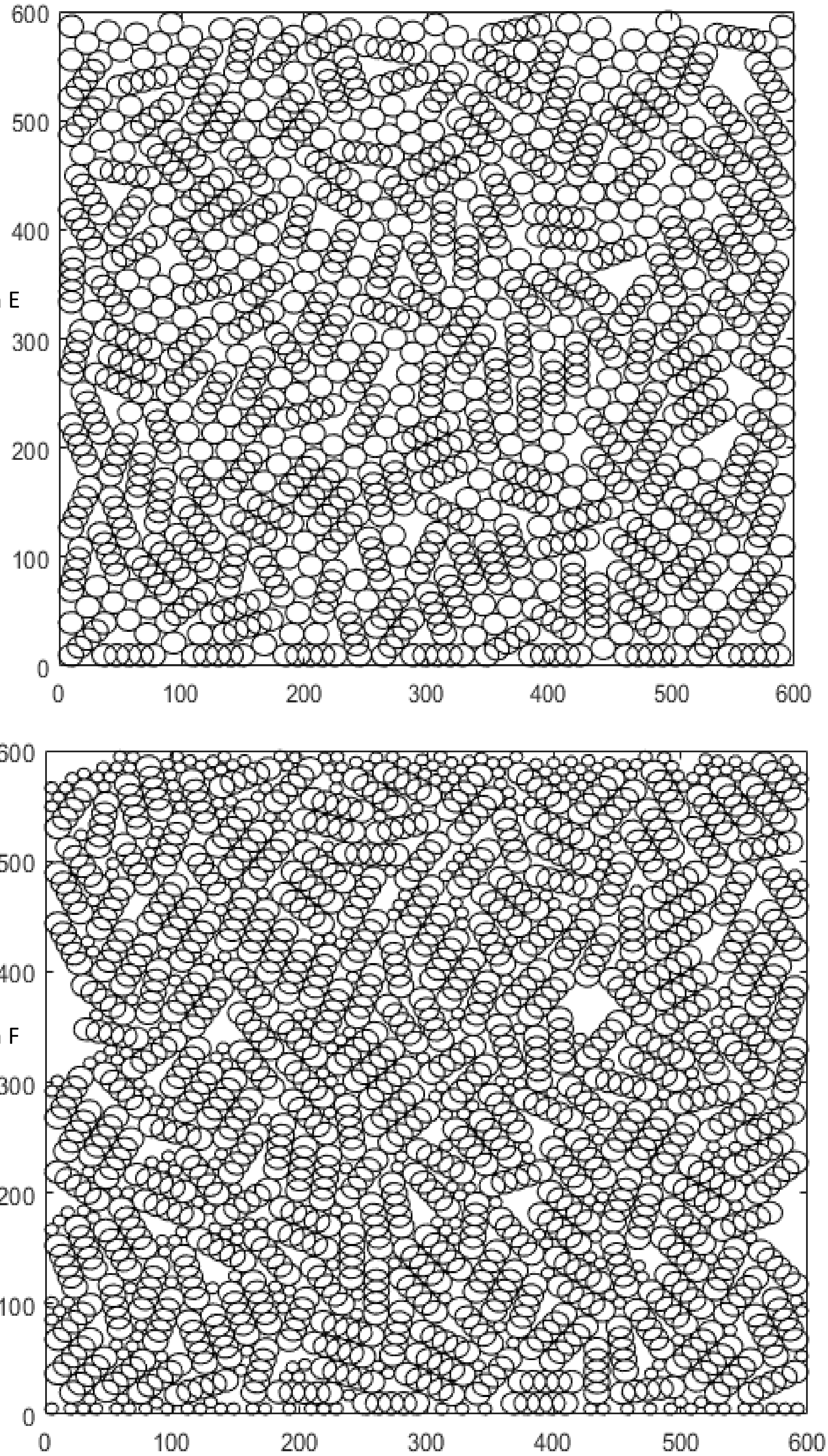
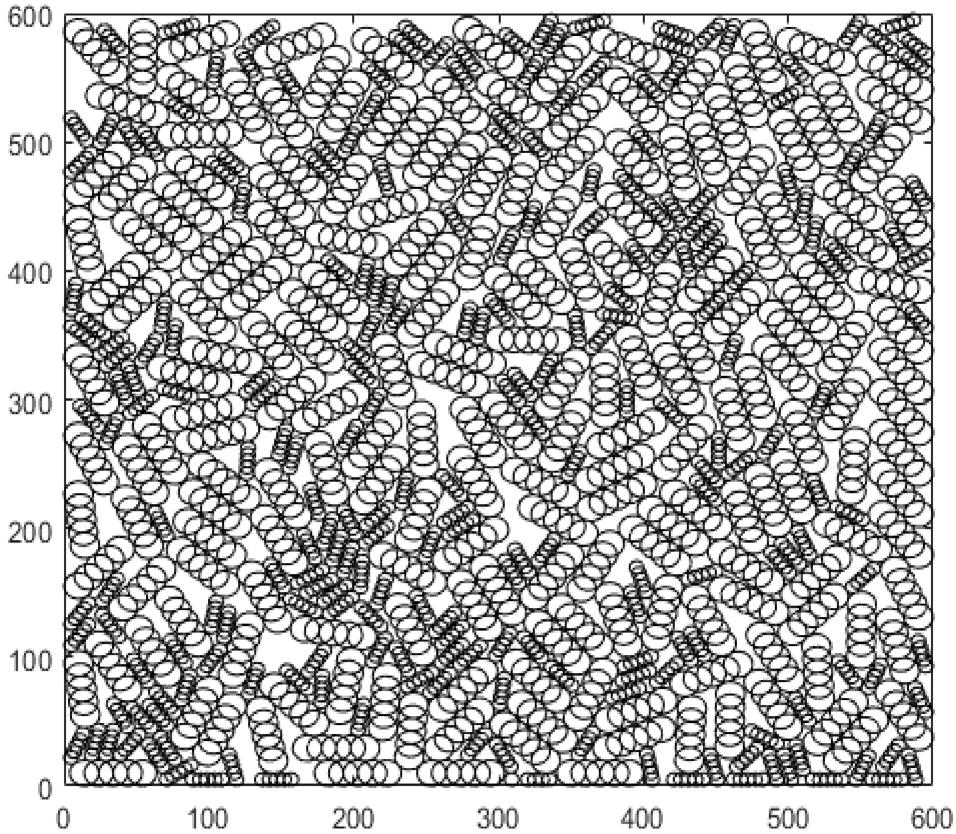


Figure 6.5: Examples of systems E and F

Figure 6.5 shows systems E and F, where single particles have been added into a system containing 5 length chains of r_p = 10. The single particles make forming clusters of aligned chains much more difficult, especially when they are the same size as the chain, as their placement on top of a cluster will immediately stop its continuation. This shows that whilst processing these chain-like particles, the introduction of small particles can dissuade clusters of chains from joining together, and therefore reduce the size of the same angled groups that form.



System G

Figure 6.6: Example of system G

Figure 6.6 shows an example of system G, which has some similarities to system E as it contains smaller particles, however being chains instead of single particles causes them to have much less ability to fit into voids between the larger chains, instead each size of chain causing the other to be unable to join clusters as they block regular placement of each other.

6.3.2 Packing Fraction

Table 6.2: The Packing Fractions in the investigated 2D chain systems.

System	Packing Fraction
A	0.790 ± 0.008
B	0.772 ± 0.019
C	0.795 ± 0.011
D	0.781 ± 0.018
E	0.795 ± 0.006
F	0.788 ± 0.013
G	0.800 ± 0.006

The packing fractions across each of the investigated systems remain relatively consistent compared to the original 2D algorithm with no real pattern found throughout systems A to G, although there is a small increase in packing fraction for the systems that introduce another size of chain or single particles. As observed before, these single particles are able to pack in between the chain particles, filling in voids.

The differences in packing between systems A to D are likely due to the randomness inherent within these systems, as there is an additional factor of randomness added to these systems in the form of the rotation of the chains, which was not present in systems only containing single particles. This is shown by the increased standard deviations in these systems compared to the regular 2D systems discussed in Chapter 4. Research has gone into the packing of 3D chain shapes, such as cylinders and spherocylinders, which have a maximum packing fraction at a specific ratio of height to diameter.[1] Spherocylinders, which are closer to the particles which I am simulating due to the curved ends, have been found to have a peak packing fraction of 0.6896 at a ratio of 0.35, with packing fractions then having a small decrease on increasing ratio. This could explain why there is no discernible pattern across systems A to D, as the ratios explored here are 1.5 to 3, which is far above the ratios explored within this research. In the research into cylindrical packing, a peak packing fraction of 0.7185 was found at a ratio of 0.9, which is still much below the ratios we investigated. Future research could look into simulating ratios closer to these to determine if a similar pattern was found, thought our systems are in 2D and have the slight difference in shape with rounded edge along the sides of the particles.

Another study looked at the simulation of cylinders compared to chain structures made up of “glued spheres” to mimic a cylindrical shape, similar to the particles investigated here.[2] It was found that the systems with cylindrical particles had higher packing fractions that the systems with glued sphere particles, found to be due to the higher volume each individual cylindrical particle has compared to the glued sphere particles. As the number of spheres used to make a particle was increased but the overall dimensions of the shape kept the same, a trend was found where the packing fraction increased, becoming closer to the packing fraction found in the true cylindrical particle systems. However, even with more spheres making up a particle, the packing fractions do not reach the same packing efficiency as the true cylindrical particles.

The introduction of singular particles in systems E and F, increase the packing fraction to be on the higher side of the range of packing fractions previously seen in systems A to D. System E has a packing fraction on par with the highest of systems A to D, likely due to the addition ratio of the particles still being 1:1, so whilst the particles in system E would affect the structure more, an equal amount of particles to the small particles in system F, cover more space. If the small particles in system F were added in a greater amount, the packing fraction would likely be higher as they would be able to fill in many more of the voids, that shown in Figure 6.5 are still quite empty.

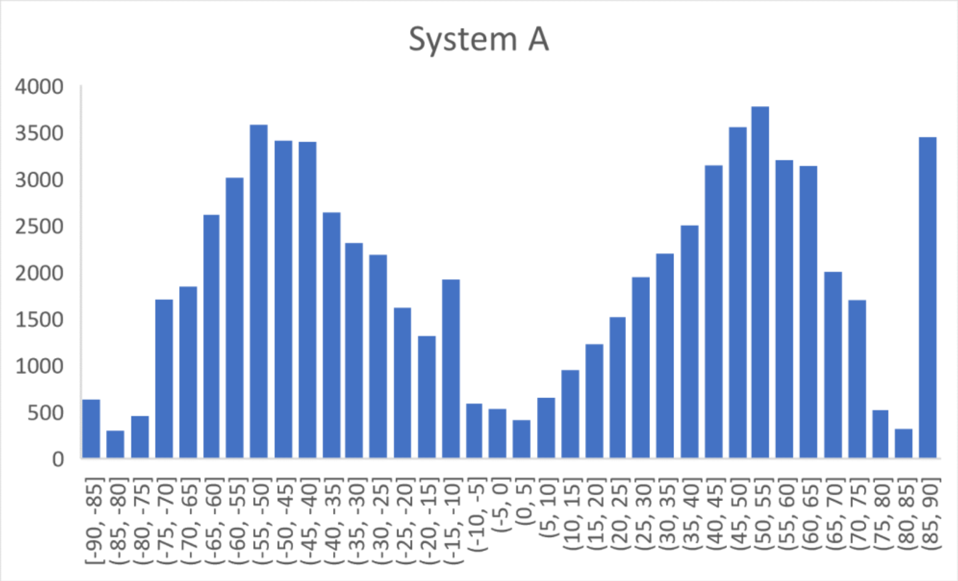
The addition of smaller chains in system G does increase the packing fraction to be above the range shown in systems A to D, again due to the small chains ability to fit in gaps that the larger chains cannot, however as expected they cannot fill in voids as well as single particles of the same size. Research has also gone into the packing of binary mixtures of cylindrical particles[3], mixing particles with a height to diameter ratio (AR) of 1 with particles with an AR of 2 and 3 separately. It was found that as a higher percentage of AR 1 particles were added to a system, the packing fraction increased for both the systems containing AR 2 and AR 3. The AR 1 and AR 2 mixture showed a smaller increase in packing fraction when increasing the percentage of AR 1 particles present but had higher packing fractions than the AR 1 and AR 3 systems overall. These patterns match other research carried out both in physical[4] and simulated[5] experiments. As my binary mixture chain system contained chains that did not differ by AR, being 3, but instead by the scale of their height and diameters this data is useful to investigate but can not be directly compared to my systems. Future runs on my model can be done to investigate particles with different AR to see if similar patterns emerge.

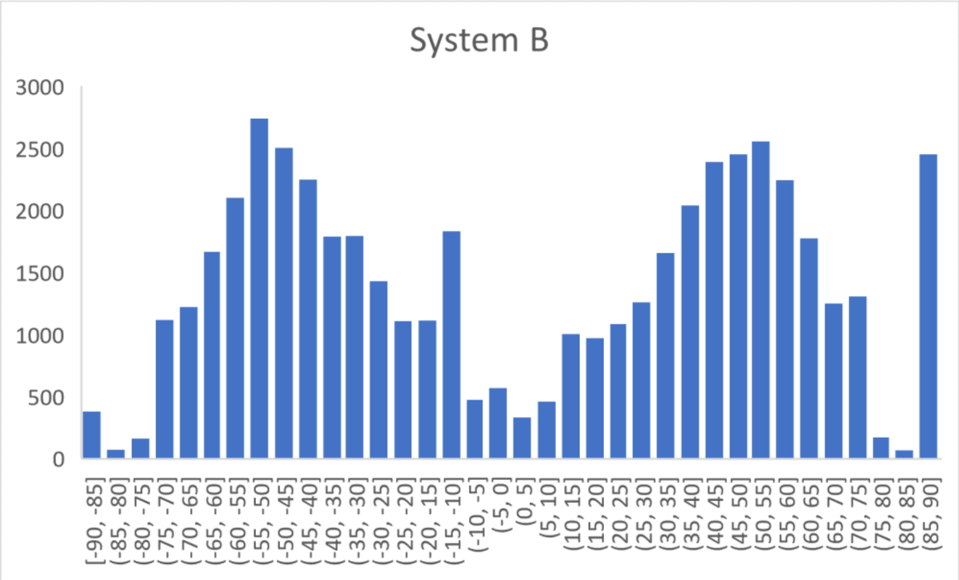
6.3.3 Chain Angles

A new property that can be investigated within the chain systems is the angle at which each chain is lying, with 0^o being directly vertical, then -90^o being lying flat to the left and $+90^o$ being lying flat to the right.

Figure 6.7 shows histograms of the chain angles present in systems A to D. Each graph shows a clear curve with two peaks, starting at the 50^o to 55^o mark in System A and moving closer to 0^o as the chains get longer, ending up at the 30^o to 35^o mark in System D. This shows the chains are becoming more vertical the longer they become, which is likely due to the increased complexity of the systems leading to less orderly packed systems, as a more ordered system of chains would be made up of chains resting horizontal on top of each other.

The peaks at 90º exist as tan(90º) is an undefined value, therefore when the calculations gave an error it was recorrected to be a 90º value, thus no corresponding peak appears at the -90º mark.





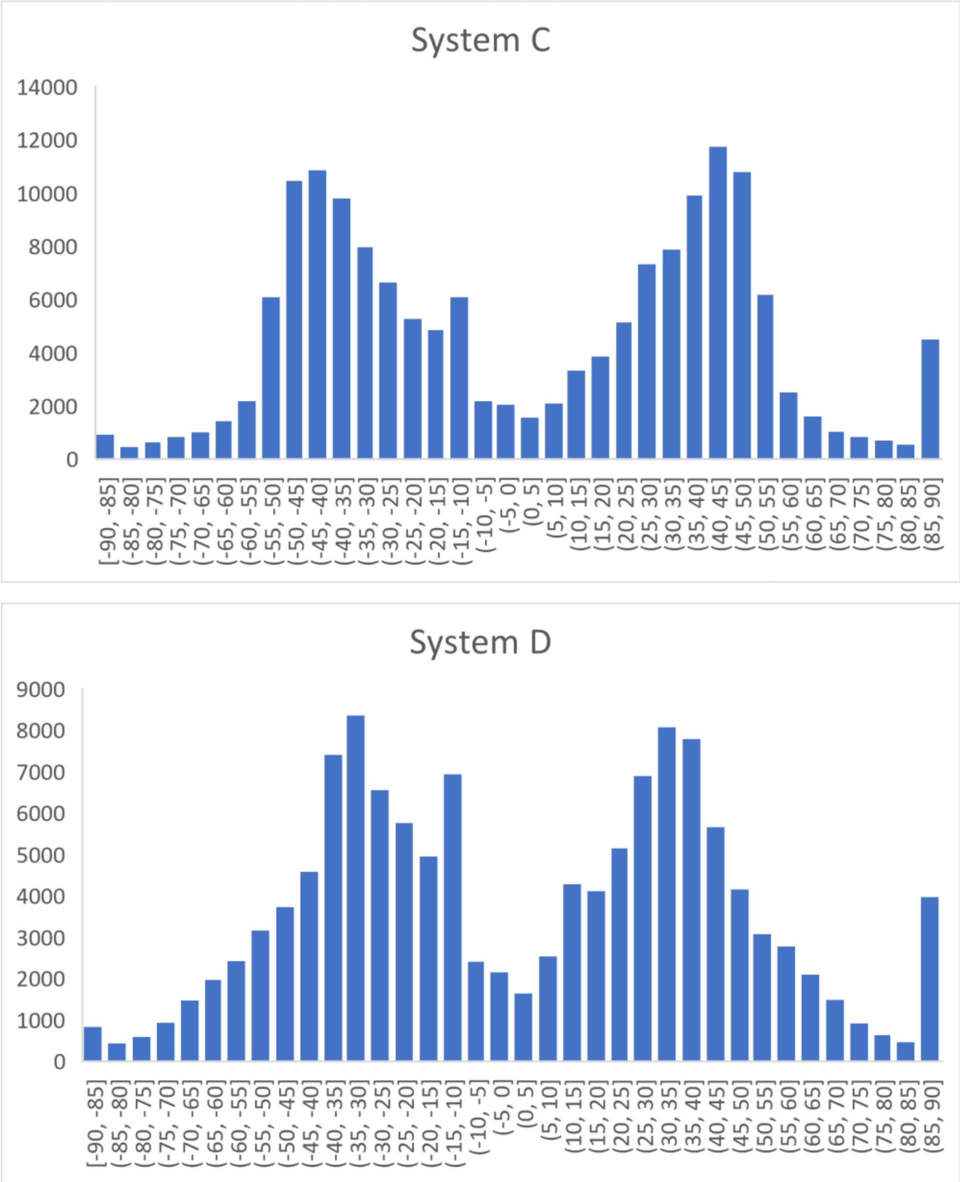


Figure 6.7: Histograms of the angles of chain placement across systems A to D

There is also a peak in the [-10^{o}, -15^{o}] bin in each of the systems. This is due to the large number base layer particles that are directly next to another particle and leaning against it, having an angle of -14.48^{o}, as shown in Figure 6.8. The abundance of these chains comes from the setup of the base layer, as after the model has run for a large number of times to try and fill the base layer, to confirm it is filled, the model runs across the base layer from left to right looking for empty space, as discussed in section 3.1.2. This results in initial particles being placed directly adjacent to horizontal chains, and when they tilt to the left, have an angle of -14.48^{o}. This could be fixed by increasing the counter used when placing random particles, or by altering the algorithm to be able particles landing on the edge of the box, instead of requiring an initial layer of particles.

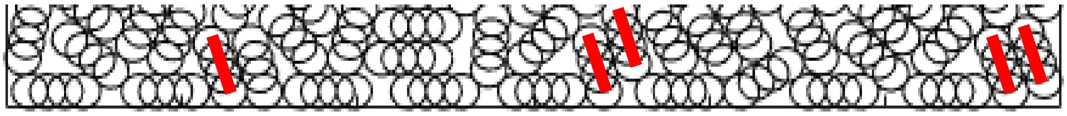


Figure 6.8: Base layer particles with angle -14^{o} particles highlighted with a red line marker.

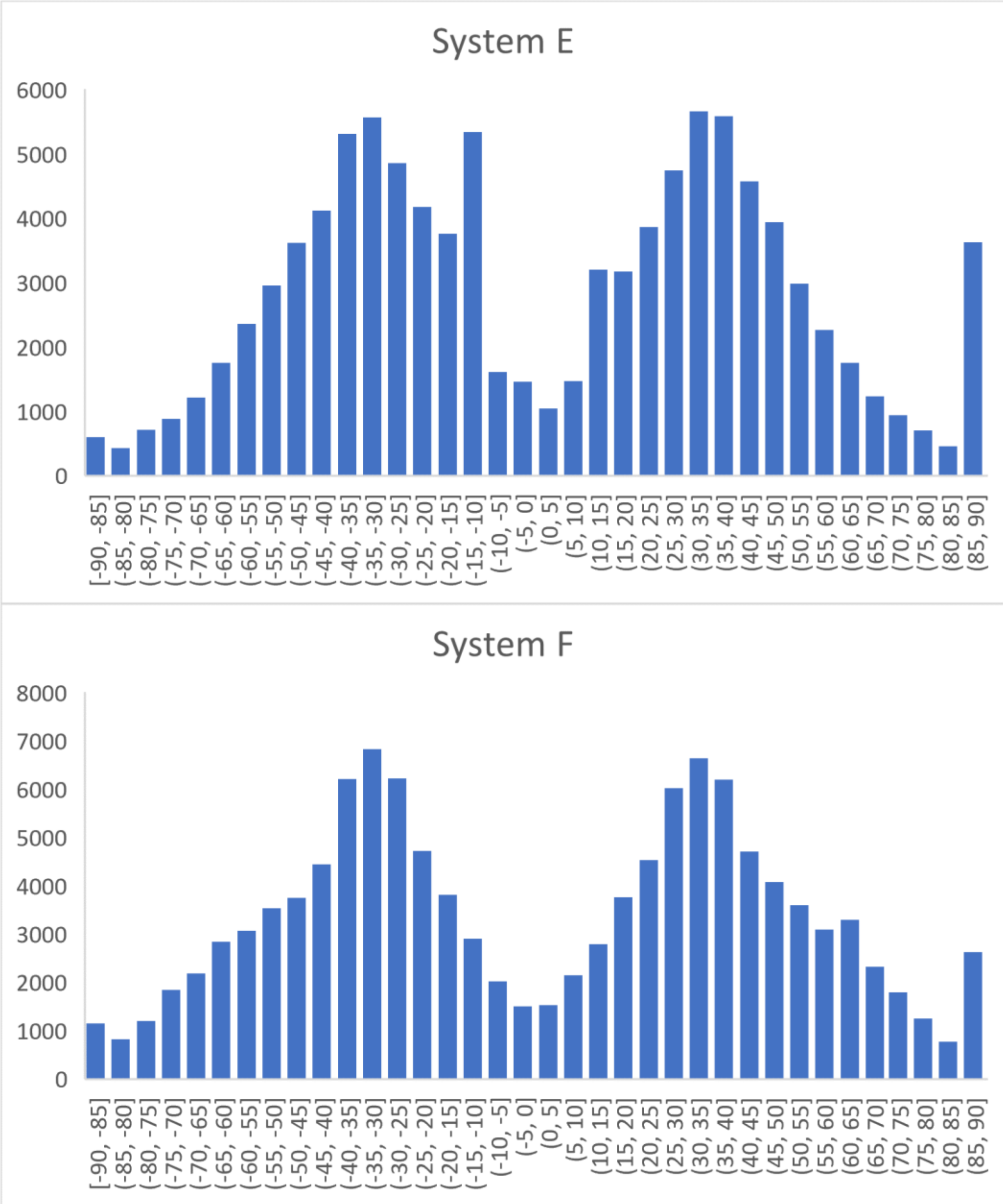


Figure 6.9: Histograms of the angles of chain placement across systems E and F

The introduction of single particles in the chain systems, E and F, does not appear to have much of an effect on the angle histograms with the peaks still being at ± 30^o to 35^o. System F does not have a peak in the [$-10^{o,}$ -15^o] bin, as the small single particles are being used to fill in the gaps in the base layer, meaning there is not an abundance of chains in the base layer resting at the specific -14.48^o angle.

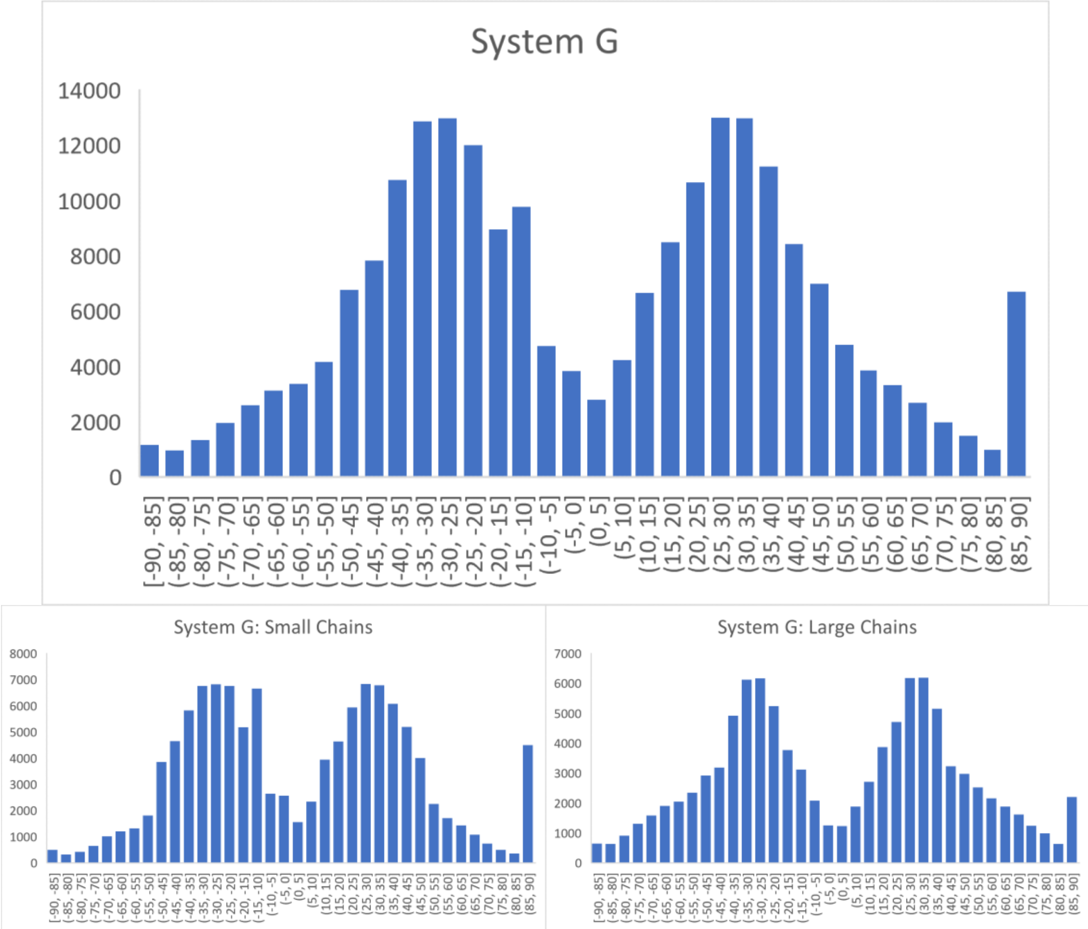


Figure 6.10: Histogram of angles of chain placement in system G overall and separated by chain size

The system G histograms, Figure 6.10, shows the same pattern in its histogram as is within systems A to D, though with the peaks being at ± 25^{o} to 30^{o}, showing the chains are more vertical than in the previous systems. This is more vertical than system D just containing length 5 chains, showing that the introduction of the smaller chains decreases the order of the system, resulting in the chains being pushed further from a regular horizontal packing.

When looking at the angles of the chains in system G separated by size, the previously observed pattern remains, with the smaller chains in these systems creating the peak at -10^{o} to -15^{o}, as the smaller chains are used to fill in gaps in the base layer if they are not filled randomly. Research on the angle of cylinders resting in binary mixtures[3] has been done, and found the opposite trend that was reported here, with the cylinders preferring to lay horizontal over vertical. This is likely due to a few factors, including my systems being 2D and the literature systems being 3D, the system size, with my systems still affected by the finite size effect, and remaining issues within my code, such as what leads to the peak at -14^{o}. Were the code to be updated and system size increased, my model would hopefully also follow the reported pattern, with chains resting more horizontal than vertical.

6.4 Summary and Conclusions

The expansion of the algorithm allows us to investigate systems containing particles that are closer to real needle-like particles, therefore it can be used to make more accurate simulations compared to perfectly circular particles. It was found that when small singular particles are added into a system of chain particles, they reduce the size of clusters of those chain particles, which could be used to lessen the effect of agglomeration in similar systems.

The packing fractions in the investigated systems remained consistent, however with an increase when singular particles are added into the systems, as noted in previous chapters, they are able to fit into the voids created by the chains.

When investigating the angles of the chains within the investigated systems, it was found that increasing the length of the chain caused the particles to become more vertical, likely due to the increased complexity caused by larger particle structures, moving the system away from a more ordered system with chains lying horizontal on top of each other. The preference to vertical over horizontal in my systems is potentially also due to the curved edges of individual particles in the chains still being present, allowing new chains to rest partway along them, instead of sliding along the chains to the edges and so becoming more horizontal.

Due to time constraints, these were the only systems that were able to be investigated, however there are many more experiments that could have been done given more time. Such as increasing the chain length further, investigating a wider range of particle radii within the same system, as well as chains that are made up of different sized particles instead of all the same. This algorithm does give us a good starting point from which to further investigate these systems, so going forward these additional systems would also be looked into.

6.5 References

(1) Shuixiang, L., Zhao, J., Lu, P., and Xie, Y., Maximum packing densities of basic 3D objects, *Chinese Science Bulletin*, **2010**, *55*(2), 114-119.

(2) Tangri, H., Guo, Y., and Curtis, J. S., Packing of cylindrical particles: DEM simulations and experimental measurements, *Powder Technology,* **2017**, *317*, 72-82.

(3) Iniyatova, G., Yermukhambetova, A., Boribayeva, A., and Golman, B., Approximate packing of binary mixtures of cylindrical particles, *Micromachines*, **2023**, *14*, 36.

(4) Zou, R., Lin, X., Yi, A., and Wong, P., Packing of cylindriocal particles with a length distribution, *Jounral of the American Ceramic Society*, **1997**, *80*, 646-652.

(5) Gan, J., and Yu, A., DEM Simulation of the packing of cylindrical particles, *Granular Matter*, **2020**, *22*, 22.

7 AstraZeneca Placement

During the fourth year of my PhD project, I spent 12 weeks on a placement at the AstraZeneca[1] Macclesfield campus, where I compared the commercial Ansys Rocky DEM[2] software with my own model, as well as then using the commercial DEM software in a Design of Experiments (DoE)[3] approach to determine parameters for an AZ Compound.

AstraZeneca are currently in the early stages of investigating various DEM modelling software packages, looking at which one best suits their needs at various stages of chemical and pharmaceutical development and processing. The main aim of my placement was to investigate Rocky DEM and its suitability for use modelling AstraZeneca particles and processes. My personal aims for this placement were to gain some professional experience working in an industry setting, as well as being able to get hands on experience with industry standard modelling software and be able to compare it to my own model.

Within the pharmaceutical industry Rocky DEM modelling software is used to investigate behaviours of particles, such as particle breakage, with varying particle sizes, and shapes in systems of different geometries.[4]

The parameters investigated were: the Basic Flowability Energy (BFE, mJ); the Rolling Resistance (no units); the Static and Dynamic Friction Coefficients (no units); the Young's Modulus (Pa); and the Coefficient of Restitution (no units). The BFE of a powder is a measure of its flow properties when it is in a loosely packed state, in our case defined by the energy required for the mixing blade to move downwards through the powder.[5] This can be used to quantify how changing various parameters of the particle affect how it flows. The Rolling Resistance quantifies the resistance that occurs when a particle rolls, either over another particle or a surface of the bounding box. The Static Friction Coefficient measures the friction that exists between two surfaces whilst they are at rest, whilst the Dynamic Friction Coefficient denotes how much friction will occur when two surfaces are sliding over one another. The Youngs' Modulus is a measure of the elasticity of the particles, denoting how much a particle shape will be affected by the forces being placed upon it. The Coefficient of Restitution is the ratio between the final and initial relative speeds between two particles after they have collided.

7.1 Familiarisation with Rocky DEM

As discussed in Section 2.2, there are multiple different types of models used within computational sciences. For these experiments I am using a DEM package called Ansys Rocky DEM. For new users of Ansys Rocky DEM, there are a series of tutorials that guide the user through various examples to familiarize them with the software and how to use it. Relevant topics include: setting up equipment geometries; establishing particle interaction characteristics; performing simulations and data visualization. Some of the advanced topics address capabilities which are relevant to this research including: particle addition; motion frames which allow geometries placed within the system to move; wear arising from multiple particle impacts; and particle breakage from an impact.

The Rocky DEM training material also included a workshop on creating custom particle shapes including fibres and multiple particle sizes. Although during my short industrial placement, I did not get the time to run simulations using non-spherical shapes, it was useful to see how Rocky DEM handled them, I was however able to use different sized particles in my Rocky simulations.

Another useful element in the training material was a simulation of a conical double screw vacuum dryer which involved multiple stacking motions, as geometries were rotating on multiple axes, both vertically and rotationally, and enabling thermal modelling within the system to see how the heat propagated through the system.

The Rocky training materials gave me a working understanding of the software such that I could use it for simulations.

7.2 Comparing Rocky DEM with my model

My model uses a simpler algorithm, described in Chapter 3, compared to Rocky DEM, with my model only simulating gravity when placing particles one at a time. Rocky however involves many particles moving within the system at the same time, and for each particle in each timestep has to identify that particle's neighbours, calculate the forces that they exert on each other, whether through contacts, electrostatic forces, or other interactions, as well as other forces present in the system, such as gravity, as shown in Equation 7.1.

$$\sum F_{net} = \sum F_{body} + \sum F_{surface} = m\frac{dv}{dt} \tag{7.1}$$

where F is Force, m is mass and t is time.[6]

Once all of these forces have been determined, they can be used to calculate how they affect an individual particle's velocity so the model can calculate the position and rotation of that particle in the next timestep, based on its old position and current new velocity, as shown in Equations 7.2 and 7.3.

$$v_{new} = v_{old} + \int_{t}^{t+\Delta t} \frac{\sum F_{net}}{m} dt \tag{7.2}$$

$$x_{new} = x_{old} + \int_{t}^{t+\Delta t} v_{new}\, dt \tag{7.3}$$

where F is Force, m is mass, t is time, v is velocity and x is position.[6]

These calculations are completed for every particle in the system, the timestep iterated to the next one, and then the whole process repeated until the simulation end time is reached or all particles have left the range of the system. Note that in principle, the simulations conserve energy however frictional forces damp motions so that thermal energy of the particles must be considered. Each Ansys Rocky DEM run used the Hysteretic Linear Spring Model, first proposed by Walton and Braun[7], which is an elastic-plastic (repulsive and dissipative) normal contact model.[6] The means that, unlike my model, the particles are "soft" and can be compressed when coming into contact with another particle. The model is implemented in Ansys Rocky DEM using the following equations 7.4 and 7.5.

$$F_n^t = \begin{cases} \min(K_{nl}s_n^t, F_n^{t-\Delta t} + K_{nu}\Delta s_n) \text{ if } \Delta \mathrm{s_n} \geq 0 \\ \max(F_n^{t-\Delta t} + K_{nu}\Delta s_n, \lambda K_{nl}s_n^t) \text{ if } \Delta \mathrm{s_n} < 0 \end{cases} \tag{7.4}$$

$$\Delta s_n = s_n^t - s_n^{t-\Delta t} \tag{7.5}$$

where F_n^t and $F_n^{t-\Delta t}$ are the normal elastic-plastic contact forces at the current time t and at the previous time $t - \Delta t$, respectively, where Δt is the timestep. Δs_n is the change in the contact normal overlap during the current time, otherwise described as the change in the size of the overlap between two particles, illustrated in Figure 7.1. It is assumed to be positive as particles approach each other and negative when they separate. s_n^t and $s_n^{t-\Delta t}$ are the normal overlap values at the current and at the previous time, respectively. K_{nl} and K_{nu} are the values of loading and unloading contact stiffnesses, respectively. λ is a dimensionless small constant. Its value in Ansys Rocky DEM is 0.001. The part of the expression in which this constant is active ensures that, during the unloading, the normal force will return to zero when the overlap decreases to zero.[6]

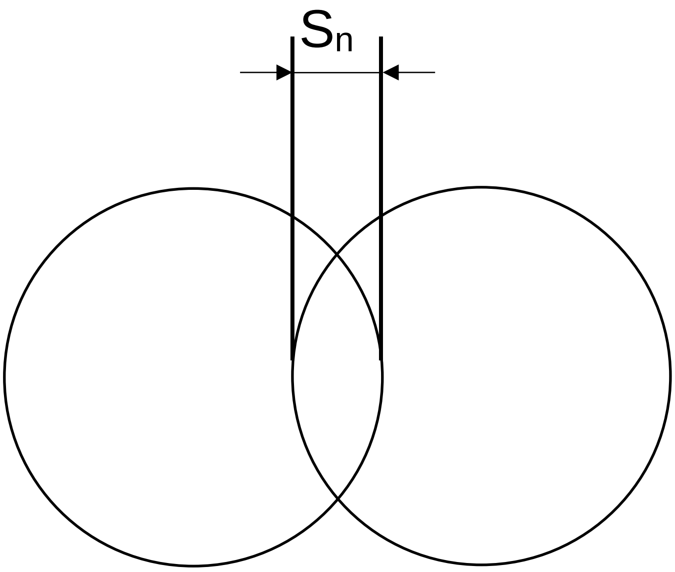


Figure 7.1: Two particles overlapping during a simulation with the contact normal overlap (S_n) labelled

As Rocky incorporates many more parameters than my model, it can be used to simulate much more complicated systems, but at the cost of having large parameters spaces that can be complicated to explore. It also has the cost of taking much more computational power and greater time to complete each run. Each of the Rocky simulations was run on the AstraZeneca supercomputer and took around one to three hours to complete each, with my model being run on a on the ARCHIE-West supercomputer and each run taking around a few minutes each.

7.2.1 Method of Particle Addition Comparison

Initial comparisons were completed between Rocky DEM and the model I created, using the same scale between particle and box size, with one box side being six particle diameters. Runs were completed in Rocky DEM using both the volume and continuous fill methods to also be able to analyse the difference between them.

The Volume fill method involves choosing a point within the system as an origin point for the particles, which for my systems was the centre of the box.

A particle is placed at this origin point, and then all subsequent particles are placed attached to this origin particle to form a ball of particles. No forces are accounted for during these additions, instead, once the model has added the requested number of particles, the forces are applied to the system, and the particles settle. Compared to this, the continuous fill method acts much more similarly to my model, with each particle entering the system from a random point in a designated inlet, which for my systems is the top of the box, and forces are immediately applied to them so they can settle. In Rocky DEM, multiple particles are added at the same time however, resulting in the possibilities of impacts and interactions as the particles are falling, instead of just once they impact the pre-existing bed.

The volume fill option was difficult to set up as it either overflowed over the top of the box, or under-filled depending on which volume was set to be filled. This problem was overcome by placing a solid lid over the top of the box, and the volume fill set to overflow. Any particles outside of the lid were then not counted in the calculations, and any particles that were initially overlapping with the box lid were removed from the system at the simulation start, as shown in Figure 7.2

The continuous option was much easier to set up as the inlet for the particles to be added to the system from could be placed at the top of the box, acting similarly to how my model functions.

The packing fractions of the systems were examined as a simple point of comparison between the different types of particle addition in Rocky DEM, and as a comparison to my model.

As shown in Table 7.1, across each of the systems investigated, the continuous addition method had a higher packing fraction than the volume fill method. This will be due to both the particles' ability to pack whilst they are being added in the continuous method, as in the volume fill method, all the particles are added in one go and then allowed to settle, which can cause gaps to occur which would allow additional particles to fit, if all particles for that system had not already been added.

Table 7.1: Packing fractions of systems created with different particle addition methods and radii

Radii Present	Particle Addition Method	Rocky Packing Fraction	My Packing Fractions
10	Volume Fill	0.439 ± 0	0.460 ± 0.007
	Continuous Fill	0.461 ± 0.005	
10, 20	Volume Fill	0.460 ± 0.004	0.497 ± 0.006
	Continuous Fill	0.475 ± 0.007	
10, 50	Volume Fill	0.436 ± 0.004	0.468 ± 0.006
	Continuous Fill	0.466 ± 0.017	

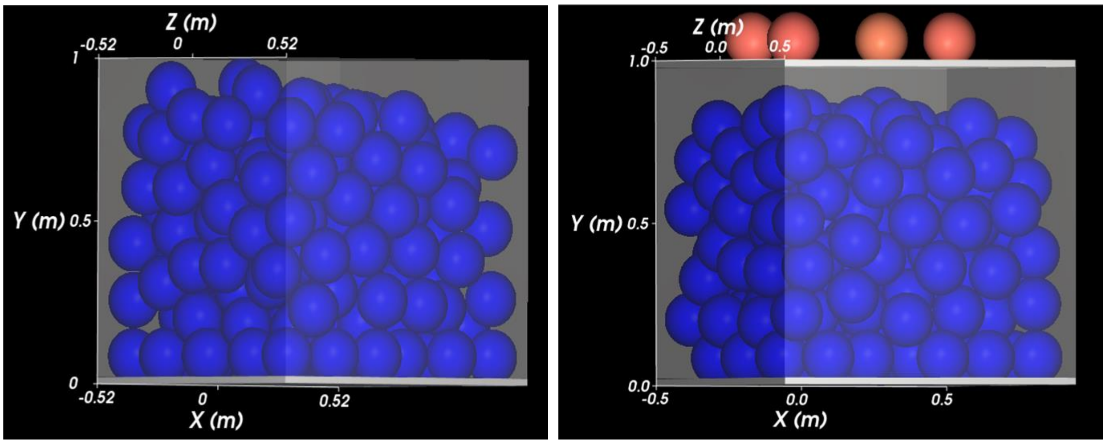


Figure 7.2: Continuous addition method (left) and Volume Fill addition method (right)

The packing fractions across the volume fill methods are also very consistent, with there being no differentiation between the repeats run in the single sized particle system. This is due to the small size of the box limiting the number of particles that can be placed around the initial centre particle, causing systems which form to be largely the same, and in the case of the only r_p = 10 system, identical.

Figure 7.2 shows the difference between the different types of particle addition, with continuous addition filling closer to the top of the box, having a higher packing fraction, and also not leaving additional particles outside it.

7.2.2 Model Packing Fraction Comparisons

As the continuous fill method is closer to how my model acts, and gives a better representation of a system being filled by particles from above than the volume fill method, it was the method used going forwards when comparing the two models. Some additional runs were conducted looking at how the different ways the particles were added in the continuous addition methods can affect the speed of the simulation runs, in order to reduce time spent waiting for them to be completed. Initially, particles were added in a consistent speed from the start to end of the simulation. However, nearer the end of the simulation, this resulted in the model attempting to add particles to the system when there was no room, wasting processing time and causing the simulation to take longer. Therefore, it was investigated what happened if particles were not added nearer the end of the simulation when the box got too full. This resulted in boxes that were not completely full as particles near the top were still able to pack down once addition had been halted. The solution used to correct this issue was to stagger the particle injections, to give each one some time to settle so that they are not immediately in the way of the next group of particles to be added.

Table 7.2: Comparison of packing fractions calculated for systems created by Rocky DEM and our model

Radii Ratio Present	Packing Fraction	
	Rocky DEM	My Model
1	0.460 ± 0.005	0.460 ± 0.007
1:2	0.475 ± 0.007	0.497 ± 0.006
1:5	0.466 ± 0.017	0.468 ± 0.006

In the Rocky DEM experiments, the ratios between the different radii, and between the radii and box size, within the investigated systems were kept consistent with the ones used within my model, and each radius having an equal number of particles added to the system. The larger radius in each case was also kept the same, so that the box size would remain consistent across each of the simulation runs.

The values obtained by the Rocky DEM simulations and my model are very similar, with the main difference coming in the r_p = 10, 20 systems. The r_p = 10 and r_p = 10, 50 systems gave almost identical results, giving more confidence in the reliability of my model, as it is consistent with this industry standard software. The small differences will come from the randomness of particle placement inherent in both systems, however the consistency shows that my model is able to match up with the industry standard.

7.3 Edge Effects Investigation

Because the boxes sizes used in my model are rather small, Rocky DEM was used to investigate the edge effects present on the packing fraction at different sizes of box using one size of spherical particle. The packing fraction was determined as a whole, and then individually for different sections of the system, as shown in Figure 7.3. Note that some areas overlap, so some particles are counted twice and areas are larger than their colours show. The roof packing fraction is denoted by the area covered by red particles (and some yellow particles), the wall packing fraction is denoted by the area covered by yellow particles (and some pink particles), the floor packing fraction is denoted by the area covered by pink particles, and the whole packing fraction uses the area of the whole system. There is an additional section that cannot be seen inside accounting for the centre packing fraction (which would be coloured green), which starts where each of the floor, wall and roof areas end.

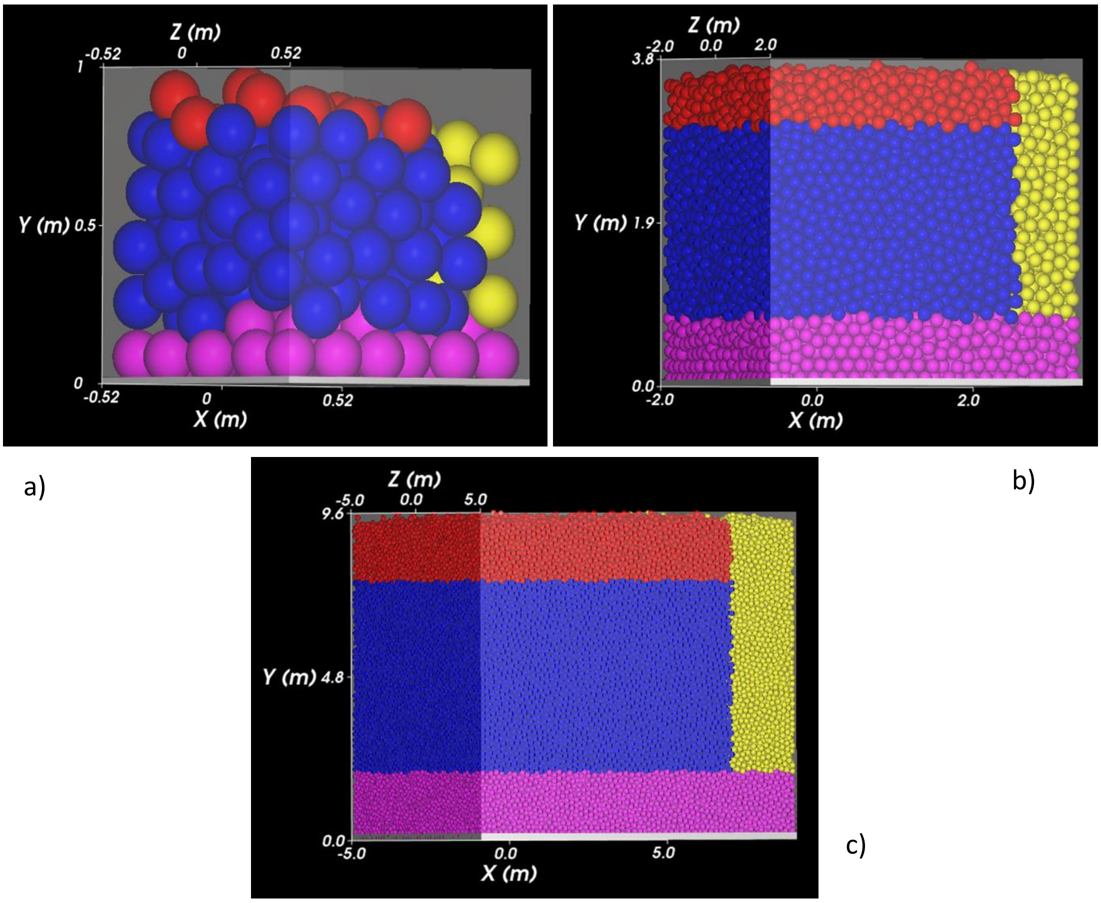


Figure 7.3: Rocky DEM systems created with different particle to box size ratios. (a) 1:6 (b) 1:24 (c) 1:60

The order of packing fractions from highest to lowest is expected to be: the centre, the floor, the walls and then the roof, as the centre of the box has no edge effects to be affected by. The floor and walls of the box are then the edges where the effect will be less due to not being the point of entry, resulting in the floor being the next most packed and the walls coming after that, as they are partly affected by the roof area. Finally, the roof section would be expected to have the lowest packing fraction as its largest surface area is affected by the entry point of the particles, where the edge effects of the box will be at their greatest. The overall packing fraction of the system would then be an appropriate weighted average of these sections, as there is some overlap between the areas and multiple walls areas to account for. This pattern is confirmed by the data shown in Tables 7.3 and 7.4.

Table 7.3: Comparison of packing fractions of different sections of particle systems with different particle:box size ratios

Particle Diameter to Box Width Ratio	Box Section				
	All	Wall	Centre	Roof	Floor
1:6	0.461 ± 0.005	0.393 ± 0.029	0.611 ± 0.035	0.180 ± 0.017	0.464 ± 0.008
1:24	0.561 ± 0.002	0.548 ± 0.003	0.612 ± 0.002	0.419 ± 0.010	0.600 ± 0.002
1:60	0.588 ± 0.001	0.582 ± 0.001	0.609 ± 0.001	0.531 ± 0.005	0.604 ± 0.001

The systems used to test the edge effects contained same sized particles equivalent to the small particles from our model systems. As the box size increases, the majority of the packing fractions increase, with the exception of the centre section, as it is not affected by the box size. The ordering of the sections packing fractions still remains the same, though the values become closer, and the overall packing fraction of the system starts to become closer to the value of the centre of the box. The increase in wall and overall packing fraction also comes from the decreased effect of the roof, as it is the area with the lowest packing fraction due to it not getting completely filled. Therefore, as the box gets taller, more area can be filled below the roof area increasing the overall packing fraction of that section.

Using this data, a set of systems were created to investigate the effect of particle size distribution, with a Particle:Box diameter ratio of 1:33 $^{1}/_{3}$, using the larger particle diameter, as it is in between two higher values tested, however it would not have as long a runtime as the 1:60 ratio systems, taking roughly two hours each instead of four, as the time on my placement was a factor. Once again, the number of particles of each size is 1:1.

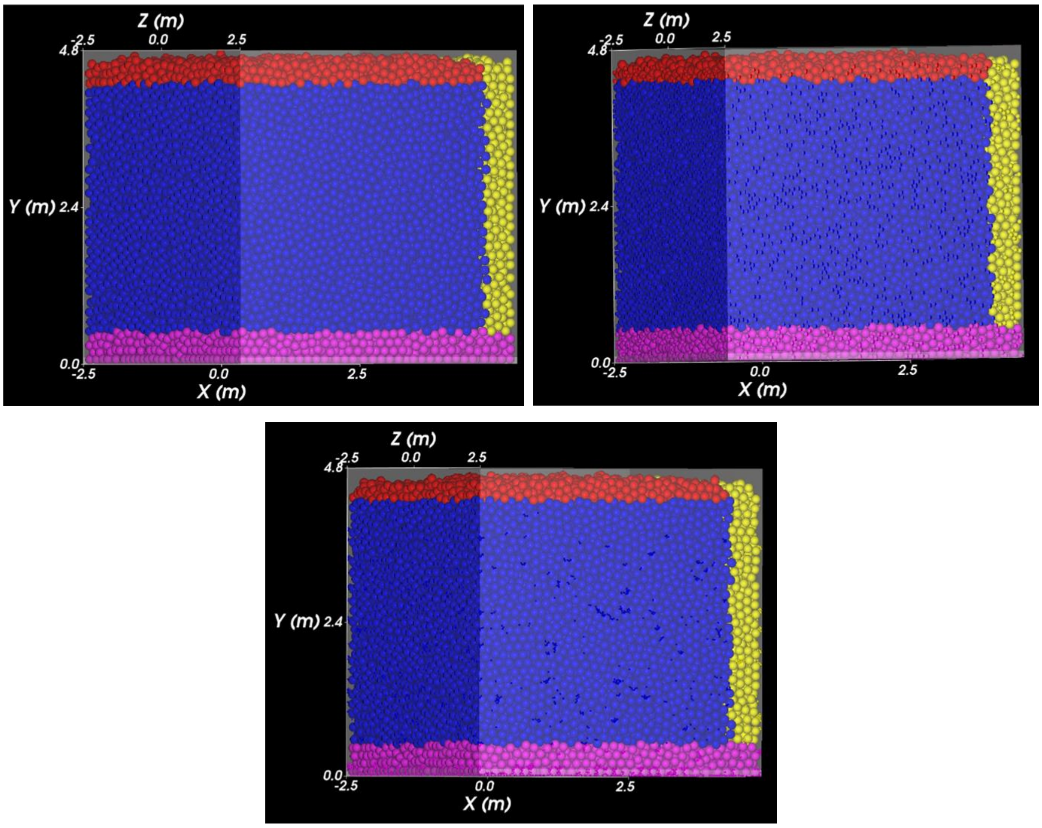


Figure 7.4: Three systems created with different particle sizes present.
a) r_p = 10, b) r_p = 10, 20, c) r_p = 10, 50

Table 7.4: Comparison of packing fractions of different sections of particle systems with different particle size distributions

Radii Ratio Present	Box Section				
	All	Wall	Centre	Roof	Floor
1	0.576 ± 0.002	0.553 ± 0.003	0.610 ± 0.0004	0.371 ± 0.021	0.594 ± 0.001
1:2	0.592 ± 0.001	0.572 ± 0.002	0.626 ± 0.0004	0.379 ± 0.009	0.612 ± 0.002
1:5	0.576 ± 0.001	0.551 ± 0.002	0.613 ± 0.0003	0.342 ± 0.015	0.603 ± 0.001

When looking at the different sections of the box across the different systems, they maintain the same pattern as previously shown. This is expected as the size of the particles will not have any real interaction with the edge effects, instead only affecting the packing fraction of the system as a whole. Across the different systems, the packing fractions also showed the same pattern as above, with an increase between the r_p = 10 to r_p = 10, 20 systems, and then a decrease to the r_p = 10, 50 system. However, in these cases, the r_p = 10 and r_p = 10, 50 systems are much closer together compared to the smaller box systems, with some of the sections of the r_p = 10, 50 system having a smaller packing fraction than the respective area in the r_p = 10 system.

7.4 Design of Experiments using Rocky DEM

The second part of my placement involved using Rocky DEM to determine various properties of an AstraZeneca compound, referred to here as Compound A. A Design of Experiments (DoE) approach was used to ensure that the experiments performed gave useful outputs, instead of using a trial-and-error approach. Within these simulations, we are modelling Compound A as a spherical particle, as modelling the exact shape and size of it would take longer than the placement would allow. For these purposes, MODDE DoE software was used to take the inputted parameters and a starting value, shown in Table 7.5, to produce a series of experiments, shown in Table 7.6, that would give a good understanding of each parameters effect on the output value.

7.4.1 Parameter Setup

Five parameters were varied during our tests: which were the Rolling Resistance; Coefficient of Restitution; Static and Dynamic Friction Coefficients; and Young’s Modulus, using the values shown in Table 7.5. The same values were used in each experiment for both the Static and Dynamic Friction Coefficients. In future experiments, we would want to use a wider variety of parameters, and use different values for each of them, however the first investigations were limited to these parameters, and had two of them equal, due to time constraints.

Table 7.5: Range of parameter values used in the Design of Experiments method

Parameter	Rolling Resistance	Coefficient of Restitution	Static and Dynamic Friction Coefficients	Young's Modulus (Pa)
Values	0.125, 0.15, 0.175	0.1, 0.2, 0.3	0.175, 0.2, 0.225	$9e^6$, $1e^7$, $1.1e^7$

The simulations were done modelling a Freeman Rheometer (FT4)[8], which is a small-scale powder rheometer that has been widely adopted in the pharmaceutical sector, shown in Figure 7.5, to calculate the BFE of Compound A, which can also be determined through laboratory experiments. Therefore, we can compare the model's output to the physically measured value to determine if the values for the tested parameters are possibly valid. The simulation involves filling the cylinder with the testing particles, and then the mixing blade pressing downwards whilst rotating.

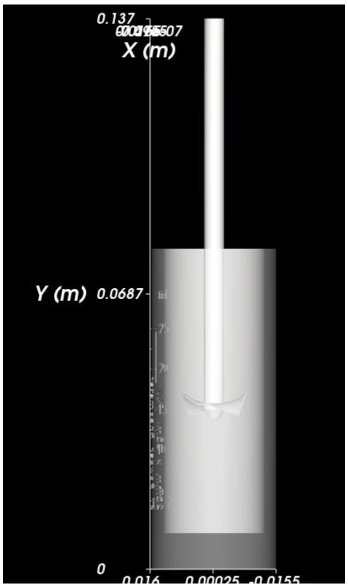


Figure 7.5: The FT4 system geometries used within Rocky DEM

7.4.2 BFE Results

Table 7.6 shows each of the experiments run and the BFE calculated from them. The BFE is a measure of the flowability of the powder whilst it is being forced to move, in this case by the energy required for the mixing blade to move downwards. Only one run was completed per set of parameters due to the time constraint on my placement, however more simulations per set would be run usually to get a better understanding of each parameter’s influence on the BFE output.

Table 7.6: Simulation runs completed and parameter values used

Run Number	Rolling Resistance	Coefficient of Restitution	Static and Dynamic Friction Coefficients	Young’s Modulus (Pa)	BFE (mJ)
1	0.125	0.1	0.175	9.00E+06	365
2	0.175	0.1	0.175	9.00E+06	432
3	0.125	0.3	0.175	9.00E+06	357
4	0.175	0.3	0.175	9.00E+06	427
5	0.125	0.1	0.225	9.00E+06	466
6	0.175	0.1	0.225	9.00E+06	634
7	0.125	0.3	0.225	9.00E+06	446
8	0.175	0.3	0.225	9.00E+06	612
9	0.125	0.1	0.175	1.10E+07	425
10	0.175	0.1	0.175	1.10E+07	534
11	0.125	0.3	0.175	1.10E+07	416
12	0.175	0.3	0.175	1.10E+07	505
13	0.125	0.1	0.225	1.10E+07	532
14	0.175	0.1	0.225	1.10E+07	716
15	0.125	0.3	0.225	1.10E+07	532
16	0.175	0.3	0.225	1.10E+07	739
17	0.15	0.2	0.2	1.00E+07	503
18	0.15	0.2	0.2	1.00E+07	527
19	0.15	0.2	0.2	1.00E+07	501
20	0.15	0.2	0.2	1.00E+07	512

By looking at how the output BFE value was affected across each experiment, the MODDE DoE software can determine the influence each parameter has on the BFE value, as well as any influence the interaction between parameters may have. Parameters that do not have an influence on the BFE output are removed from the calculations, refining all the parameters relationships to the BFE output to Equation 7.6. It can then use this to calculate a point within the ranges tested that satisfy our desired BFE for Compound A of 542.23 mJ, shown as value set 1 in Table 7.7. The value of 542.23 mJ was previously determined in laboratory experiments, so we can therefore use this value to determine if the simulation is accurately modelling Compound A. An equation to calculate the BFE can also be determined, including coefficients for each of the parameters tested based on their influence on the final BFE, determined by how big a change in the BFE value each parameter caused.

$$BFE = -5174.26r_r - 2825.76f + 4.13899e^{-5}y + 39127.7(rr * f) + 0c_r + 262.533 \quad (7.6)$$

$$BFE = -5174.26r_r - 2825.76f + 4.13899e^{-5}y + 39127.7(rr * f) + 262.533 \quad (7.7)$$

where *BFE* is the Basic Flowability Energy, r_r is the Rolling Resistance, f is the Static and Dynamic Friction Coefficients, c_r is the Coefficient of Restitution and y is the Young's Modulus.

Equation 7.6 shows that the Rolling Resistance and Static and Dynamic Friction coefficients had a large influence on the BFE, with the Young's Modulus having a small influence and the Coefficient of Restitution having no influence. The interaction between the rolling resistance and the friction coefficients was also determined to have an influence. This is shown by the coefficients in the equation for the parameters that have a high influence being in the order of 10^4 and 10^5, whereas the parameters with a small influence had a coefficient in the order of 10^{-5} and 0. For each of these influential parameters, a higher value results in a higher BFE. This fitted equation has an R2 value of 0.99 and a Q2 value of 0.97, showing that there is a high measure of both the model's fit and predictability. R2, also known as R-squared, is a measure of the variance of a dataset, with a value ranging from 0 to 1. A value closer to 1 denotes that there is less variance within the dataset, and a value closer to 0 denotes that there is a large degree of variance. Our value of R2 being 0.99 in this dataset shows that the variance is minimal, and changes to the outputs are based upon our changing of the parameters and not random chance. Q2, also known as Q-squared, is a measure of the predictive relevance of a model, with a positive value denoting good predictive relevance. Our Q2 value of 0.97 shows that this model has predictive relevance.[9] This equation is only valid within the specific parameter ranges tested as part of the DoE runs, stated in Table 7.5, however as the target value is within this range it does not matter. Equation 7.7 shows a simplified form of equation 7.6, removing the parameters that had no influence on the calculation.

Runs 17 to 20 were completed using the midpoints for the range of values used for each parameter. These repeats were performed to ensure the BFE values obtained were consistent with each other when the parameter values were kept the same, as there is inherent randomness within the simulations from the particle placements, and the calculations that stem from it. Therefore, ensuring that the data gained from the repeats are similar enough, even given these changes between simulations, means that we can trust the simulations and know that variations in the outputs are due to the effect of changing parameters, rather than just a random event in a simulation. These repeats gave an average of 511 ± 10.3, giving a variance of just under 2%.

7.4.3 Value Confirmation

Three sets of values were then calculated using Equation 7.6, as shown in Table 7.7, that should give the target BFE of 542 mJ, then FT4 model simulation runs were completed using these sets of values to see if the simulation and equation values were the same.

Table 7.7: "Solution" parameter sets and their determined BFE

Value Set	Rolling Resistance	Coefficient of Restitution	Static and Dynamic Friction Coefficients	Young's Modulus (Pa)	Equation BFE (mJ)	Simulation BFE (mJ)
1	0.156334	0.3	0.212233	9.3264e6	538.05	529.72 ± 10.4
2	0.157	0.3	0.213	9.33e6	542.92	511.14 ± 13.8
3	0.16	0.3	0.21	9.33e6	542.10	534.11 ± 8.50

Three repeats were completed for each of the value sets, and the average BFEs determined. There was some variance between each of the runs for the data parameter values due to the randomness that can occur in the system, which is consistent with the previous repeats, however Sets 1 and 3 both gave values close to the desired output. Run 2 gave consistently lower BFE values than anticipated, even with the equation stating it should have been close to 542 mJ, showing that there are likely more interactions at play than our current series of experiments account for. Given more time, a larger range of values would have been used across more variables, to be able to investigate the effect of as many parameters as possible and looking for valid values within a larger range, instead of focusing on a narrower area.

As there were many sets of values that can satisfy the expected BFE condition, we then used these values to simulate a test to calculate the Angle of Repose (AoR) of Compound A, another property which can be readily compared to an experimentally measured value. The simulations involved filling a cylinder with our particles, then slowing raising the cylinder, allowing the particles to form a pile on a platform below it, as shown in Figure 7.6. Five repeats were completed for each of the value sets identified in Table 7.7, and the angles calculated are shown in Table 7.8.

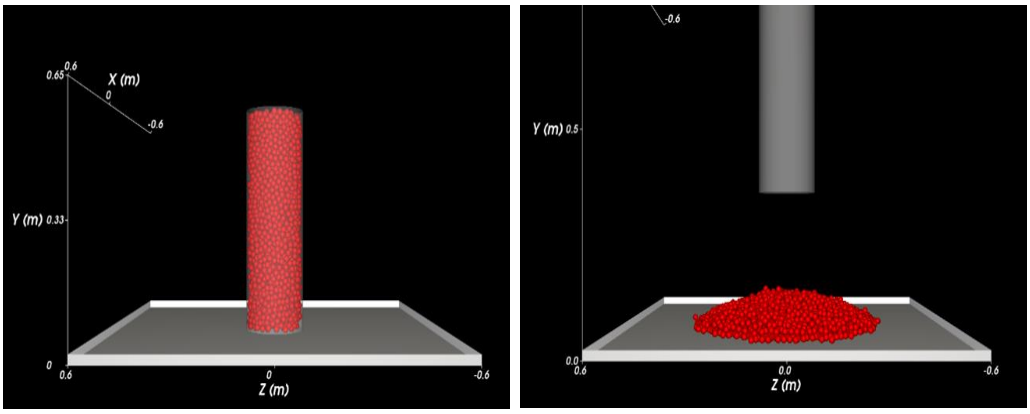


Figure 7.6: A cylinder of particles before and after it has been lifted to allow the particles to settle forming a heap.

Each simulation gave two angles, one calculated from the bottom of the pile and one calculated from the top, as illustrated in Figure 7.7. To calculate the angle, a line is drawn across the pile of particles, and the maximum height at set intervals across the line is calculated. The line then rotates 10^{o} around the y-axis (normal to the platform) and the heights collected again. This is repeated until the line has rotated a full 360^{o} and is back in its starting rotation.

The black dots in Figure 7.7 represent the average of the maximum heights at each of the intervals used. The light red area is the maximum of the maximum heights found, with the dark red area being the minimum of the

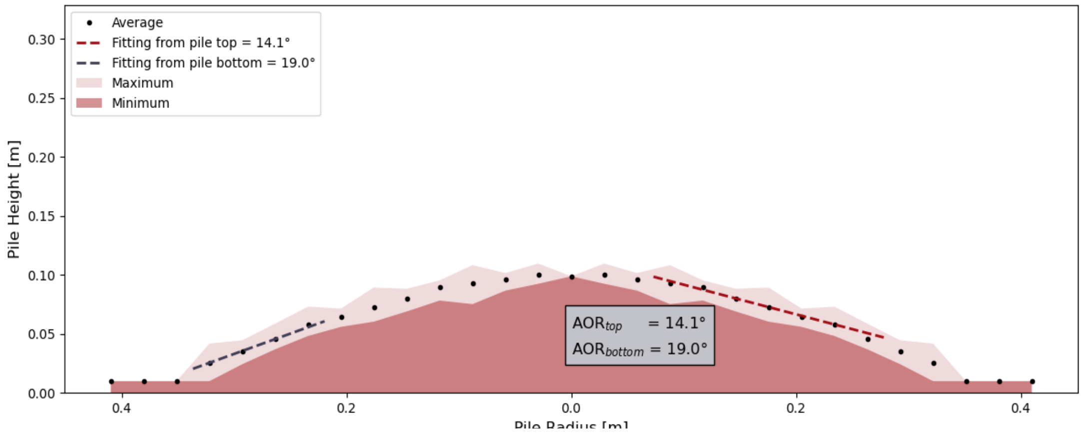


Figure 7.7: Example particle system with labelled Angles of Repose from Ansys Rocky DEM

maximum heights found. Using the average maximum heights, the angles are then calculated, as shown by the fitting lines.

Table 7.8: The Angles of Repose calculated for each of the parameter value sets tested

Value Set	Bottom AoR (°)	Top AoR (°)
1	14.9 ± 1.8	18.9 ± 0.9
2	14.7 ± 2.4	18.2 ± 1.2
3	17.2 ± 1.9	20.0 ± 2.3

The values calculated by the model can then be compared with the laboratory value to determine which parameter value sets are closest. Alongside the data from the BFE experiments, the values can be checked to see which ones satisfy both tests. The laboratory values for the AoR for Compound A were not able to be obtained before the end of my project, however these data can still be used for comparison once the data is available.

7.5 Summary and Conclusions

At this point my placement came to an end, however a discussion was had about how this work would be used by AstraZeneca going forward. More conditions would be required to confirm the parameter values as only two conditions, being the BFE and AoR, do not give enough confidence in the values generated by Ansys Rocky DEM. Other possible confirmation experiments could be Ring Shear Tests[10] or Granudrum[11] equipment testing for powder cohesion. This would be the next stage for AstraZeneca going forward to further confirm the data values for the parameters we have tested, as well as testing as many other parameters as possible, to increase the confidence in the parameter values determined. Once a full set of model parameters are obtained, simulation of Compound A would be possible.

The Design of Experiments method was useful for having a predetermined set of experiments to run instead of taking a trial-and-error approach, as it gave more structure to the workflow. It also allowed us to properly investigate and determine the effects of each parameter, instead of just finding a value set that satisfied our condition.

The comparisons between my model and Rocky DEM showed them producing very similar packing fraction data, showing the same trend across different particle size distributions. This increases my confidence in the accuracy of my model, as it compares with professionally made software.

7.6 References

(1) *AstraZeneca.* https://www.astrazeneca.co.uk/ (accessed 2023).
(2) *Ansys Rocky DEM.* https://www.ansys.com/en-gb/products/fluids/ansys-rocky (accessed 2023).
(3) *American Society for Quality: Design of Experiments*. https://asq.org/quality-resources/design-of-experiments (accessed 2023).
(4) Chiaravalle, A., G., Cotabarren, I., M. and Pina, J., DEM breakage calibration for single particle fracture of maize kernels under a particle replacement approach, *Chem. Eng. Res. Des.*, **2023**, *195*, 151-165.
(5) Freeman, F., Measuring the flow properties of consolidated, conditioned and aerated powders — A comparative study using a powder rheometer and a rotational shear cell, *Powder Technology,* **2007**, *174*, 25-33.
(6) ESSS Rocky, Release 2021 R2.2, DEM Technical Manual, ESS Rocky DEM, S. R. l., **2021**.
(7) Walton, O., R., and Braun, R., L., Viscosity, granular-temperature, and stress calculations for shearing assemblies of inelastic, frictional disks. *Journal of Rheology*, **1986,** *30*, 948–980.
(8) *Freeman Technology*, https://www.freemantech.co.uk/powder-testing/ft4-powder-rheometer-powder-flow-tester/external-variables (accessed 2023).

(9) *Research with Fawad,* https://researchwithfawad.com/index.php/lp-courses/basic-and-advance-data-analysis-using-smart-pls/understanding-r-square-f-square-and-q-square-using-smart-pls/ (accessed 2024).

(10) *American Society for Testing and Materials.* https://www.astm.org/d6467-21e01.html (accessed 2023).

(11) *Granutools: GranuDrum.* https://products.granutools.com/granudrum (accessed 2023).

8 Conclusions and Future Work

8.1 Summary and Conclusions

Over the course of the project multiple algorithms have been produced to investigate particle packing, in both 2D and 3D, and with spherical and non-spherical particles. Initial investigations were done into these systems, comparing them to systems created by other models and the expected mathematical answers to the properties, such as minimum/maximum packing fractions.

Comparisons were made between the packing of the 2D and 3D systems that were produced, containing single-sized particles, against previously calculated values of packing fractions for perfectly packed structures, and those created under different algorithms. It was found that my system's packing fraction values fell within the expected range, as they were less than the most ordered system, as my systems have a degree of randomness, and were above the packing fractions of RSA systems, where the circles are not under as many constraints as within my systems therefore pack less efficiently. Initial investigations were also made into the number of contacts each particle had compared to the expected amount, four in 2D and six in 3D, with this being mostly shown in the data however it was greatly affected by the box size therefore further investigation with larger systems sizes is needed.

An improvement was made to the model to allow chain particles to be created within the systems. The packing fractions were found to be rather consistent, especially compared to the single particle systems, and an expected increase when single void-filling particles were added into the systems. The angles of the chains were also determined, and it was found that as the chains became longer, the particles tended to rest more vertically.

The model was also compared to Industry standard software, Rocky Ansys DEM, and it was found that when similar systems were created in both models, the data found was very similar. This gives a lot of confidence in the outputs of my model as they are being corroborated by modelling software that has been used in many different scientific studies.

Once consistent note throughout each of the systems was the effect of the size of the box on the outputs. As these were only initial tests, the box sizes were often limited by the project duration, or the capabilities of the model at the time. Now the model has improved, investigations into larger systems are possible in future projects. The evidence presented provides confidence in these algorithms such that they can be used in future work to investigate these structures further and the model can be further improved to investigate a wider range of systems and properties.

The research completed through this project has given some more insight into the packing of spherical shapes when placed under gravity, as well as some investigation into non-spherical shapes. This aids other research completed into these topics and the completion of this model allows another avenue into further investigation into both these questions, as well as, once the model has had more features implemented, further areas within this topic. The creation of this model allows for more direct research into the packing of the systems it creates. Due to its simplicity, the model could be applied to a variety of different fields, beyond the initial pharmaceutical base it was created for, as with no defined scale the particle could be any size the researcher wants, e.g. larger for use in soil sciences. It is also an easy base to build on, compared to editing more complex modelling software or creating new forcefields, so can again be tailored for a variety of applications.

Some of the project objectives were achieved with the completion of the current algorithms, however preferably the 3D chains algorithm would have been completed, along with the upgraded version of the contact breakage algorithm. Details of these plans and other future work are discussed in the next section.

8.2 Future Work

As this project has created the basis of a modelling system, there are many various applications for it going forward beyond my project.

Some of the most obvious are continuations of work that has been started as part of my project but could not be investigated to the extent that would be wanted, such as the investigation into the contact forces present between particles. The 3D non-spherical particle algorithm could be finalised, so that initial property investigations, as have been done for 2D and 3D spherical particle structures, can be completed**.** In addition, it would be preferable if the spherical basis of the chain particles could be removed to be able to model smooth edged particles. One way this could be approached with the current method is to increase the amount by which the circular particles, that make up a chain, overlap, to create a smoother surface.

Further work could be done to gain a larger range of data in 2D systems, investigating the effects, if any, of percolation structures and particle size distribution on how the bed breaks apart. The algorithm could also be expanded to be able to handle 3D structures and non-spherical particle structures.

Further investigation of some of the phenomena observed during the initial tests of the model, such as looking at the path that a small particle takes through a bed formed of larger particles would be desirable. This could also help to investigate fluid paths through the bed. As part of potentially investigating fluids, additional dynamics would be added to the model, such as solvent effects, and particles being affected by friction as they are settling.

Experimental work could also be done alongside the model systems created, one of which would be to build on recent unpublished work undertaken using the Diamond Light Synchrotron to gather images of packed systems that can be compared against those that the model produces. Breakage tests can also be done to see how systems break apart under stresses to compare to our breakage algorithm.

As the scales of the particles is separate from the units used, the particles could be any size that the user desires them to be. For example, the model could be used within soil sciences to model the settling of larger particulates than would be investigated in the pharmaceutical industry.

Appendix 1: 2D System Algorithm

This appendix contains the algorithm used for creating 2D systems.

```
module allSubs ! Initialises the variables used through all functions
character, dimension (:,:), allocatable, public :: RA*4
integer, dimension (:,:), allocatable, public :: RAMolClose
integer MolNo, RadLarge, RadSmall, BoxSize, GridSize, Rads, count, SN, Full,
FullCount
integer MLr, AllocateVal, RoofCount
real MLxReal, MLyReal
integer, dimension (:,:), allocatable, public :: Ones
integer, dimension (:,:), allocatable, public :: Contacts
dimension MLr(10000), Rads(10), MLxReal(10000), MLyReal(10000)
logical FullCheck, Hit, RoofHit, StartPlace
integer x, y, Long, Tall, RadT
end module allSubs



program packedbed
use allSubs ! Loads the variables from the module

! Initialises local variables
character t, FileName*15, FileID*3
integer m, n, check, PrintNo, ProgCount, PCId, count3
dimension FileID(1000)
real Rand, Dist
integer count2, RadTnew, TotLength
logical Finished, Cont, Impact

StartPlace = .TRUE.

t = 'y'

if (t == 'y') then
	Rads(1) = 10	! Sets the radius of a particle. Additional radii would be
inputted as Rads(x) = 'Radius'
	SN = 1		! Sets the number of different radii in the system

	RadLarge = 0
	RadSmall = 0

	do count = 1, SN
		if (RadLarge < Rads(count)) then
			RadLarge = Rads(count)
		end if
		if (RadSmall > Rads(count) .or. RadSmall == 0) then
			RadSmall = Rads(count)
		end if
	end do

	RadLarge = 10
	RadSmall = 10
```

```
! Calculates the box size based on the largest radius present
BoxSize = (RadLarge*6)
GridSize = BoxSize*5
AllocateVal = ((BoxSize*3)**2)*2

! Allocates the arrays
allocate(RA(1:GridSize, 1:GridSize))
allocate(RAMolClose(1:GridSize, 1:GridSize))

allocate(Ones(1:AllocateVal,1:2))

do ProgCount = 1, 50
        write(FileID(ProgCount), '(i0)') ProgCount
end do

do ProgCount = 1, 50 ! Starts the loop for the number of systems to be created

        ! Sets variables initial values
        MLxReal = 0
        MLyReal = 0
        MLr = 0

        TotLength = 0
        MolNo = 1
        FileName = ''
        Full = 0
        check = 0
        FullCount = 0
        Finished = .FALSE.

        Ones = 0

        RA = '0'
        RAMolClose = 0

        RoofCount = 0
        RoofHit = .FALSE.

        call random_seed()

        do while (count < 10000000)

                Impact = .FALSE.

                ! Picks a random radius and x coordinate, and sets y to be on
the bottom of the box
                call random_number(RX)
                count2 = 1 + floor(SN*RX)
                RadT = Rads(count2)

                call random_number(RX)
                count2 = 1 + floor((GridSize-(2*RadT))*RX)
                x = count2+RadT

                y = RadT
```

```
if (MolNo > 1) then ! Checks there is already at least one particle in the system
    hitloop: do count3 = 1, MolNo - 1
        Dist = ((MLxReal(count3)-x)**2)+((MLyReal(count3)-y)**2)
        Dist = sqrt(Dist)
        if (Dist <= ((RadT*2)+(Mlr(count3)))) then
            count = count + 1
            Impact = .TRUE.
            exit hitloop
        end if
    end do hitloop
    if (Impact .eqv. .FALSE. .and. x <= GridSize-(RadT*2)) then ! If the particle is not overlapping with any others and is inside the grid, its location is saved
        count = 0
        MLxReal(MolNo) = x
        MLyReal(MolNo) = y
        MLr(MolNo) = RadT
        MolNo = MolNo + 1
    end if
else
    MLxReal(MolNo) = x
    MLyReal(MolNo) = y
    MLr(MolNo) = RadT
    MolNo = MolNo + 1
end if
end do

! This loops through the base line to check that there is nowhere a small particle could fall through to the bottom of the box, and if so, places a particle there
do m = RadSmall, GridSize-RadSmall
    RadT = RadSmall
    Impact = .FALSE.
    hitloop2: do count3 = 1, MolNo - 1
        Dist = ((MLxReal(count3)-m)**2)
        Dist = sqrt(Dist)
        if (Dist < ((RadSmall)+(Mlr(count3)))) then
            count = count + 1
            Impact = .TRUE.
            exit hitloop2
        end if
    end do hitloop2
    if (Impact .eqv. .FALSE.) then
        count = 0
        MLxReal(MolNo) = m
        MLyReal(MolNo) = RadSmall
        MLr(MolNo) = RadSmall
        MolNo = MolNo + 1
    end if
end do

StartPlace = .FALSE.

do n = 1, 2500 ! Loops for each particle being added to the system, using the main function. At the end of each loop, it checks if the box is full and if so, leaves the loop.
```

```
                call molpos
                if (Full == 1) then
                        exit
                elseif (RoofHit .eqv. .TRUE.) then
                        exit
                end if
        end do

        t = 'y'
        if (t == 'y' .and. RoofHit .eqv. .TRUE.) then ! Saves the particle locations to a file
                FileName = ''
                FileName = trim(adjustl(FileID(ProgCount))) // '.csv'
                open(1, file = FileName, status = 'new')
                do y = 1, MolNo-1
                        write(1,*) MLxReal(y), ',' , MLyReal(y), ',' , MLr(y)
                end do
                close(1)
        end if

        t = 'y'
        if (t == 'y' .and. RoofHit .eqv. .TRUE.) then ! Saves the particle contacts to a file
                allocate(Contacts(1:MolNo, 1:MolNo))
                Contacts = 0
                do y = 1, MolNo-1
                        n = 1
                        do x = 1, MolNo-1
                                if (x /= y) then
                                        Dist = ((MLxReal(y) - MLxReal(x))**2) + ((MLyReal(y) - MLyReal(x))**2)
                                        Dist = sqrt(Dist)
                                        if (Dist <= (MLr(y) + MLr(x) + 0.01)) then
                                                Contacts(y,n) = x
                                                n = n + 1
                                        end if
                                        if (n > TotLength) then
                                                TotLength = n
                                        end if
                                end if
                        end do
                end do

                FileName = ''
                FileName = 'contacts' // trim(adjustl(FileID(ProgCount))) // '.csv'
                open(3, file = FileName, status = 'new')
                do y = 1, MolNo-1
                        do x = 1, TotLength
                                write(3,'(I4,A1,X)', advance='no') Contacts(y,x), ','
                        end do
                        write(3, *) ''
                end do
                close(3)
                deallocate(Contacts)
```

```
                    end if
            end do
    end if

    end program


    subroutine molpos
    use allSubs ! Loads the variables from the module

    ! Sets up the local variables
    integer count2, Spot, Height, DoubRad, RowRad, a,b,c, RadIn, m, n, TempX, TempY
    real Rand, MidWay, Dist
    character t, FileName*15
    integer SavIncremX, SavIncremY, SafeLocCount, LR, SavOneX, SavOneY, SavDist,
SavOnePart, LRNo, OneCount, RealPos1, RealPos2
    integer TempRealPos1, TempRealPos2
    logical SafeLocFound, ResetCheck
    real TempXa, TempXb, TempYa, TempYb, DistAB, DistBC, DistAC, AngleA, AngleB,
AngleFin, GradFin, HelpDist
    real FDistA, FDistB, FDistC, FDistD
    real xDiff, yDiff, FinalSavX, FinalSavY
    integer FinalSavLong, FinalSavTall, checktime, Balanced
    integer Balances, Touches
    dimension Balances(10)
    real DistFac, RadScale
    integer NewPos, TRP1Swap, TRP2Swap
    integer FinalPart, SideCount
    dimension FinalPart(2,3)
    real PartCoords, stochDists, sumDist, dx, stochxnew, newCoords, stochynew
    integer ibad
    dimension PartCoords(2,2), stochDists(2), newCoords(2), dx(2), stochxnew(2)
    real RX, NewX, NewY
    logical ChainAdd
    real OverDist
    integer OverDistNo
    logical NotBal, FirstBal
    dimension BalCheckNo(10000), BalCombi(100000,2)
    integer BalCheckCount, BalCount, BalCheckNo, BalCombi, BalAttempt
    integer OverlapCount
    logical EdgeCase, FirstEdge
    integer EdgeCombi, EdgeCount, EdgeAttempt
    dimension EdgeCombi(10000)
    real Dy, intC, CheckY

40  CONTINUE

    if (FullCount == 2500000) then
            Full = 1
    end if

    ! Setting initial values of variables
    Hit = .FALSE.
    ResetCheck = .FALSE.
    checktime = 1
    Balanced = 0
    Touches = 0
```

```
Balances = 0

Dy = 0
intC = 0
CheckY = 0

EdgeAttempt = 0
EdgeCount = 0
EdgeCombi = 0
EdgeCase = .FALSE.
FirstEdge = .TRUE.

FinalPart = 999999
OverDist = 0
OverDistNo = 0

OverlapCount = 0

TRP1Swap = 0
TRP2Swap = 0

ChainAdd = .FALSE.

NotBal = .FALSE.
FirstBal = .TRUE.
BalCheckNo = 0
BalCombi = 0
BalCheckCount = 0
BalCount = 0
BalCheckNo = 0
BalCombi = 0
BalAttempt = 0

RadScale = 0
TempRealPos1 = 0
TempRealPos2 = 0
DistFac = 0
FDistA = 0
FDistB = 0
FDistC = 0
FDistD = 0
SavDist = 0
SavOneX = 0
SavOneY = 0
TempXa = 0
TempXb = 0
TempYa = 0
TempYb = 0
DistAB = 0
DistBC = 0
DistAC = 0
AngleA = 0
AngleB = 0
AngleFin = 0
GradFin = 0
xDiff = 0
yDiff = 0
```

```
        NewPos = 0

        FinalDists = 0

        PartCoords = 0
        stochDists = 0
        sumDist = 0
        dx = 0
        stochxnew = 0
        newCoords = 0
        stochynew = 0

        ! Randomly chooses which radius will be used for this particle
        call random_number(Rand)
        RadScale = RadLarge/RadSmall
        RadScale = RadScale + 1
        count2 = 1 + floor(2*Rand)
        if (count2 == 2) then
                RadT = RadLarge
        else
                RadT = RadSmall
        end if

        wloop: do while (Hit .eqv. .FALSE.)        ! The loop to place particles

        ! Randomly chooses the x value
                call random_number(Rand)
                Spot = 1 + floor((GridSize-(2*RadT))*Rand)
                x = Spot+RadT

                do y = GridSize, RadT, -1          ! Loops from the top of the box, and sends to
the function to determine impact
                        Long = (x/BoxSize)+1
                        Tall = (y/BoxSize)+1

                        call PointSafe

                        ! If the box is full or an impact has occured, the loop is exited

                        if (Full == 1) then
                                exit wloop
                        end if
                        if (FullCheck .eqv. .TRUE.) then
                                RoofHit = .TRUE.
                                GO TO 10
                        end if

                        if (Hit .eqv. .TRUE.) then
                                exit wloop
                        end if
                end do
        end do wloop

        if (Hit .eqv. .TRUE. .and. Full /= 1) then
                if (MolNo > 1) then
                        RA = '0'
                        RAMolClose = 0
```

```
                        ! Sets up the variables to be used for particle placement

                        Ones = 0
                        OneCount = 1

                        do a = 1, MolNo - 1       ! Loops through the particles for contour plot
placement
                                MLxCor = MLxReal(a)
                                MLyCor = MLyReal(a) ! Takes the radius, x and y coordinates
of the current particle in the loop
                                RadIn = Mlr(a)

                                DoubRad = (RadIn+RadT)+1
                                do Height = 0, RadIn     ! Draws the particle onto the contour
plot, "-"s marking blocked locations, "1"s being valid spots
                                        MidWay = RadIn**2 - Height**2
                                        RowRad = abs(sqrt(MidWay))
                                        if     (MLxCor+Height<=GridSize-RadT     .and.
MLyCor+RowRad<=GridSize .and. MLyCor-RowRad>=RadT .and. MLxCor-Height<=RadT)
then
                                                RA(MLyCor+RowRad, MLxCor+Height) = '-'
                                                RA(MLyCor-RowRad, MLxCor+Height) = '-'
                                                RA(MLyCor+RowRad, MLxCor-Height) = '-'
                                                RA(MLyCor-RowRad, MLxCor-Height) = '-'
                                        end if
                                        do count2 = -RowRad,RowRad
                                                if  (MLxCor+Height<=GridSize-RadT  .and.
MLyCor+count2<=GridSize .and. MLxCor-Height>=RadT .and. MLyCor+count2>=RadT) then
                                                        RA(MLyCor+count2,
MLxCor+Height) = '-'
                                                        RA(MLyCor+count2,       MLxCor-
Height) = '-'
                                                end if
                                        end do
                                end do

                                do Height = 0, DoubRad ! Draws locations around the current
particle that are too close for the new particle to be added due to overlap
                                        MidWay = DoubRad**2 - Height**2
                                        RowRad = abs(sqrt(Midway))
                                        do count2 = -rowrad+1, rowrad-1
                                                if     (MLyCor+count2<=GridSize     .and.
MLxCor+Height<=GridSize-RadT        .and.        MLyCor+count2>=RadT        .and.
MLxCor+Height>=RadT) then
                                                        RA(MLyCor+count2,
MLxCor+Height) = '-'
                                                end if
                                                if     (MLyCor+count2<=GridSize     .and.
MLxCor-Height<=GridSize-RadT    .and.    MLyCor+count2>=RadT    .and.    MLxCor-
Height>=RadT) then
                                                        RA(MLyCor+count2,       MLxCor-
Height) = '-'
                                                end if
                                        end do
                                end do
```

```
do Height = -RadIn, RadIn
	do count2 = MlyCor-1, 1, -1
		if (count2<=GridSize .and. MLxCor+Height<=GridSize-RadT .and. count2>=RadT .and. MLxCor+Height>=RadT) then
			RA(count2, MLxCor+Height) = '-'
		end if
	end do
end do

do Height = -DoubRad, 0			! Adds the valid spots for the resting particle to be placed
	MidWay = DoubRad**2 - Height**2
	RowRad = abs(sqrt(Midway))
	if (MLyCor+RowRad<=GridSize .and. MLxCor+Height<=GridSize-RadT .and. MLxCor+Height>=RadT .and. MLyCor+RowRad>=RadT) then
		if (RA(MLyCor+RowRad, MLxCor+Height) /= '-') then
			RA(MLyCor+RowRad, MLxCor+Height) = '1'
			RAMolClose(MLyCor+RowRad, MLxCor+Height) = a
		end if
	end if
	if (MLyCor-RowRad>=RadT .and. MLxCor+height<=GridSize-RadT .and. MLxCor+height>=RadT .and. MLyCor-RowRad<=GridSize) then
		if (RA(MLyCor-RowRad, MLxCor+Height) /= '-') then
			RA(MLyCor-RowRad, MLxCor+Height) = '1'
			RAMolClose(MLyCor-RowRad, MLxCor+Height) = a
		end if
	end if
	if (MLyCor+RowRad<=GridSize .and. MLxCor-Height<=GridSize-RadT .and. MLxCor-Height>=RadT .and. MLyCor+RowRad>=RadT) then
		if (RA(MLyCor+RowRad, MLxCor-Height) /= '-') then
			RA(MLyCor+RowRad, MLxCor-Height) = '1'
			RAMolClose(MLyCor+RowRad, MLxCor-Height) = a
		end if
	end if
	if (MLyCor-RowRad>=RadT .and. MLxCor-height<=GridSize-RadT .and. MLxCor-height>=RadT .and. MLyCor-RowRad<=GridSize) then
		if (RA(MLyCor-RowRad, MLxCor-Height) /= '-') then
			RA(MLyCor-RowRad, MLxCor-Height) = '1'
			RAMolClose(MLyCor-RowRad, MLxCor-Height) = a
		end if
	end if
	do count2 = -RowRad, RowRad
```

```
                        if (MLyCor+count2<=GridSize .and. MLxCor+height<=GridSize-RadT .and. MLxCor+height>=RadT .and. MLyCor+count2>=RadT) then
                            if (RA(MLyCor+count2, MLxCor+Height) /= '-')  then
                                RA(MLyCor+count2, MLxCor+Height) = '1'
                                RAMolClose(MLyCor+count2, MLxCor+Height) = a
                            end if
                        end if
                        if (MLyCor-count2>=RadT .and. MLxCor+height<=GridSize-RadT .and. MLxCor+height>=RadT .and. MLyCor-count2<=GridSize) then
                            if (RA(MLyCor-count2, MLxCor+Height) /= '-') then
                                RA(MLyCor-count2, MLxCor+Height) = '1'
                                RAMolClose(MLyCor-count2, MLxCor+Height) = a
                            end if
                        end if
                        if (MLyCor+count2>=RadT .and. MLxCor-height<=GridSize-RadT .and. MLxCor-height>=RadT .and. MLyCor+count2<=GridSize) then
                            if (RA(MLyCor+count2, MLxCor-Height) /= '-') then
                                RA(MLyCor+count2, MLxCor-Height) = '1'
                                RAMolClose(MLyCor+count2, MLxCor-Height) = a
                            end if
                        end if
                        if (MLyCor-count2<=GridSize .and. MLxCor-height<=GridSize-RadT .and. MLxCor-height>=RadT .and. MLyCor-count2>=RadT) then
                            if (RA(MLyCor-count2, MLxCor-Height) /= '-') then
                                RA(MLyCor-count2, MLxCor-Height) = '1'
                                RAMolClose(MLyCor-count2, MLxCor-Height) = a
                            end if
                        end if
                    end do
                end do
            end do

            do a = 1, GridSize ! Finds the valid points and saves them to an array
                do b = 1, GridSize
                    if (RA(b,a) == '1') then
                        Ones(OneCount, 1) = b
                        Ones(OneCount, 2) = a
                        OneCount = OneCount + 1
                    end if
                end do
            end do
```

```
do a = 1, OneCount - 1 ! Finds the closest of these points to the impact
location and moves the particle to it
        TempX = Ones(a,2)
        TempY = Ones(a,1)

        if (TempY <= y+1) then
                Dist = ((x-TempX)**2)+((y-TempY)**2)
                Dist = sqrt(Dist)
                if (Dist < SavDist .or. SavDist == 0) then
                        SavDist = Dist
                        SavOneX = Ones(a,2)
                        SavOneY = Ones(a,1)
                end if
        end if
end do

TempX = SavOneX
TempY = SavOneY

if (TempX == 0 .and. TempY == 0) then
        GO TO 10
end if

SavOnePart = RAMolClose(TempY, TempX)

SafeLocCount = 0
SavIncremX = SavOneX
SavIncremY = SavOneY
SafeLocFound = .FALSE.
FinalSavX = 0
FinalSavY = 0
LR = 0
RealPos1 = 0
RealPos2 = 0

if (MLxReal(SavOnePart) == TempX) then ! Determines which way
the particle should roll
        call random_number(Rand)
        LRNo = 1 + floor(2*Rand)
        if (LRNo == 1) then
                LR = -1
        elseif (LRNo == 2) then
                LR = 1
        end if
elseif (MLxReal(SavOnePart) < TempX) then
        LR = 1
elseif (MLxReal(SavOnePart) > TempX) then
        LR = -1
end if

RMPInter = 0
RMPPrev = SavOnePart
RealPos1 = SavOnePart

do a = 1, OneCount-1
        if (Ones(a,2) == TempX) then
                SideCount = a
```

```
            end if
        end do

        do while (SafeLocFound .eqv. .FALSE.) ! Iterates in that direction until the next position would be higher, or reaching the edge of the box
            if (Ones(SideCount+LR,1) > Ones(SideCount,1) .or. Ones(SideCount,2) == RadT .or. Ones(SideCount,2) == GridSize-RadT) then
                SafeLocFound = .TRUE.
            else
                SideCount = SideCount + LR
            end if
        end do

        TempX = Ones(SideCount,2)
        TempY = Ones(SideCount,1)

        do a = 1, MolNo-1 ! Finds the particles closest to the low point for the new particle to be resting on
            if (MLyReal(a) < TempY+RadT) then
                Dist = ((MLxReal(a)-TempX)**2)+((MLyReal(a)-TempY)**2)
                Dist = sqrt(Dist) - RadT - MLr(a)
                if (Dist <= FinalPart(1,2)) then
                    FinalPart(2,1) = FinalPart(1,1)
                    FinalPart(2,2) = FinalPart(1,2)
                    FinalPart(2,3) = FinalPart(1,3)
                    FinalPart(1,1) = a
                    FinalPart(1,2) = Dist
                    FinalPart(1,3) = Dist + RadT + MLr(a)
                elseif (Dist <= FinalPart(2,2)) then
                    FinalPart(2,1) = a
                    FinalPart(2,2) = Dist
                    FinalPart(2,3) = Dist + RadT + MLr(a)
                end if
            end if
        end do

50      CONTINUE

        PartCoords(1,1) = MLxReal(FinalPart(1,1))
        PartCoords(1,2) = MLyReal(FinalPart(1,1))

        PartCoords(2,1) = MLxReal(FinalPart(2,1))
        PartCoords(2,2) = MLyReal(FinalPart(2,1))

        if (TempX == RadT .or. TempX == GridSize-RadT) then
            EdgeCase = .TRUE.
        end if

        if (EdgeCase .eqv. .TRUE.) then ! If the particle is on an edge, it balances the new particle on the edge + one particle
            NewX = TempX

            Dist = RadT + MLr(FinalPart(1,1))

            NewY = -(MLxReal(FinalPart(1,1))**2)+(2*MLxReal(FinalPart(1,1))*NewX)
```

```
                                        NewY = NewY+(Dist**2)-(NewX**2)
                                        NewY = sqrt(NewY)

                                        if (NewY /= NewY) then

25                                              CONTINUE

                                                if (FirstEdge .eqv. .TRUE.) then
                                                        do a = 1, MolNo-1
                                                                Dist = (MLxReal(a)-NewX)
                                                                if (Dist < RadT   +       MLr(a)       +
(RadT*8)) then
                                                                        EdgeCombi(EdgeCount)   =
a
                                                                        EdgeCount = EdgeCount +
1
                                                                end if
                                                        end do
                                                end if

                                                FirstEdge = .FALSE.

                                                if (EdgeAttempt <= EdgeCount-1) then
                                                        FinalPart(1,1)    =    EdgeCombi(EdgeCount-
EdgeAttempt)
                                                        EdgeAttempt = EdgeAttempt + 1
                                                        GO TO 50
                                                end if

                                                FullCount = FullCount + 1
                                                GO TO 10
                                        end if

                                        if  (TempY  -  (MLyReal(FinalPart(1,1))  +  NewY)  <  TempY  -
(MLyReal(FinalPart(1,1)) - NewY)) then
                                                NewY = MLyReal(FinalPart(1,1)) + NewY
                                        else
                                                NewY = MLyReal(FinalPart(1,1)) - NewY
                                        end if

                                        do a = 1, MolNo-1         ! Confirming the new particle is not
overlapping with any other particles
                                                Dist      =      ((MLxReal(a)-NewX)**2)+((MlyReal(a)-
NewY)**2)
                                                Dist = sqrt(Dist)
                                                if (Dist < MLr(a)+RadT-0.25) then
                                                        OverlapCount = OverlapCount + 1
                                                        if (OverlapCount > 2500) then
                                                                GO TO 10
                                                        end if

                                                        FinalPart(1,1) = a
                                                        FullCount = FullCount + 1
                                                        GO TO 50
                                                end if
                                        end do
```

```
				if (NewX < RadT .or. NewY < RadT .or. NewX > GridSize-
RadT .or. NewY > GridSize-RadT) then
					FullCount = FullCount + 1
					GO TO 25
				end if

			else ! else if the particle is not on an edge it balances on two particles
through stochastic optimisation
				newCoords(1) = TempX
				newCoords(2) = TempY

				stochDists(1) = FinalPart(1,3)
				stochDists(2) = FinalPart(2,3)

				sumDist = stochDists(1) + stochDists(2)

				dx(1) = 10 * RadLarge
				dx(2) = 10 * RadLarge

				do a = 1, 10
					do b = 1, 2
						dx(b) = dx(b) / 10
					end do
					do c = 1, 500
						call random_number(RX)
						stochxnew(2) = newCoords(2) + dx(2) *
(2*RX-1)
						call random_number(RX)
						stochxnew(1) = newCoords(1) + dx(1) *
(2*RX-1)

						ibad = 0

						do b = 1, 2
							stochDists(b) = ((PartCoords(b,1)-
stochxnew(1))**2)+((PartCoords(b,2)-stochxnew(2))**2)
							stochDists(b) = sqrt(stochDists(b)) –
RadT – MLr(FinalPart(b,1))
							if (stochDists(b) < 0) then
								ibad = 1
							end if
						end do
						stochynew = stochDists(1) + stochDists(2)
						if (stochynew < sumDist .and. ibad == 0)
then
							do b = 1, 2
								newCoords(b) =
stochxnew(b)
							end do
							sumDist = stochynew
						end if
					end do
				end do

				NewX = newCoords(1)
				NewY = newCoords(2)
```

```
do a = 1, MolNo-1
    Dist = ((MlxReal(a)-NewX)**2)+((MlyReal(a)-NewY)**2)

    Dist = sqrt(Dist)
    if (Dist < MLr(a)+RadT) then
        OverDist = 0
        OverlapCount = OverlapCount + 1
        if (OverlapCount > 2500) then
            NotBal = .TRUE.
            GO TO 70
        end if

        do b = 1, 2
            Dist = ((MlxReal(a)-PartCoords(b,1))**2)+((MlyReal(a)-PartCoords(b,2))**2)
            Dist = sqrt(Dist)
            if (Dist < OverDist .or. OverDist == 0) then

                OverDist = Dist
                OverDistNo = b
            end if
        end do
        FinalPart(OverDistNo,1) = a
        FullCount = FullCount + 1
        if (OverDist == 0) then
            GO TO 10
        else
            GO TO 50
        end if
    end if
end do

NotBal = .FALSE.
if (NewX > PartCoords(1,1) .and. NewX > PartCoords(2,1)) then ! Confirms the new particle is resting in between the old particles
    NotBal = .TRUE.
elseif (NewX < PartCoords(1,1) .and. NewX < PartCoords(2,1)) then

    NotBal = .TRUE.
end if
if (NewY < PartCoords(1,2) .and. NewY < PartCoords(2,2)) then

    NotBal = .TRUE.
end if
do b = 1, 2
    Dist = ((NewX-PartCoords(b,1))**2)+((NewY-PartCoords(b,2))**2)

    Dist = sqrt(Dist)
    if (Dist > RadT + MLr(FinalPart(b,1))+1) then
        NotBal = .TRUE.
    end if
end do

Dy = (PartCoords(1,2) – PartCoords(2,2)) / (PartCoords(1,1) – PartCoords(2,1))

intC =  PartCoords(1,2) – (Dy * PartCoords(1,1))
CheckY = (Dy * NewX) + intC
```

```
                                        if (NewY < CheckY) then
                                                NotBal = .TRUE.
                                        end if

70                                      CONTINUE

                                        OverlapCount = 0

                                        If (NotBal .eqv. .TRUE.) then     ! If the particle is not
correctly balancing then it looks for alternate particles to be resting on and moves to that
location
                                                if (FirstBal .eqv. .TRUE.) then
                                                        if (FinalPart(1,1) > FinalPart(2,1)) then
                                                                count = FinalPart(1,1)
                                                                FinalPart(1,1) = FinalPart(2,1)
                                                                FinalPart(2,1) = count
                                                                GO TO 50
                                                        end if

                                                        do a = 1, MolNo-1
                                                                Dist = ((MLxReal(a)-TempX)**2)
                                                                Dist = sqrt(Dist)
                                                                if (Dist < RadT   +       MLr(a)       +
(RadT*8)) then

        BalCheckNo(BalCheckCount) = a
                                                                        BalCheckCount              =
BalCheckCount + 1
                                                                end if
                                                        end do
                                                        aLoop: do a = 1, BalCheckCount-1
                                                                do b = 1, BalCheckCount-1
                                                                        if (b < a) then
                                                                                if    (BalCount    <=
100000) then
                                                                                        if
(BalCheckNo(a) /= 0 .and. BalCheckNo(b) /= 0) then
                                                                                                if
(abs(MlxReal(BalCheckNo(a)) – MlxReal(BalCheckNo(b))) <= RadT*2 + MLr(BalCheckNo(a))
+ MLr(BalCheckNo(b))) then

        BalCombi(BalCount,1) = BalCheckNo(a)

        BalCombi(BalCount,2) = BalCheckNo(b)

        BalCount = BalCount + 1

        end if
                                                                                        end if
                                                                                else
                                                                                        exit aLoop
                                                                                end if
                                                                        end if
                                                                end do
                                                        end do aLoop
                                                end if
```

```
                    FirstBal = .FALSE.

                    if (BalAttempt <= BalCount-1) then
                        FinalPart(1,1)      =      BalCombi(BalCount-
BalAttempt,1)
                        FinalPart(2,1)      =      BalCombi(BalCount-
BalAttempt,2)
                        BalAttempt = BalAttempt + 1
                        if (BalAttempt <= 100000) then
                            GO TO 50
                        end if
                    end if

                    if (TempX >= GridSize-(RadT*5)) then
                        OverlapCount = 0
                        TempX = GridSize-RadT
                        EdgeCase = .TRUE.
                        GO TO 50
                    elseif (TempX <= RadT+(RadT*5)) then
                        OverlapCount = 0
                        TempX = RadT
                        EdgeCase = .TRUE.
                        GO TO 50
                    end if

                    GO TO 10

                end if

            end if

            if (NewX > GridSize-RadT .or. NewX < RadT .or. NewY > GridSize-
RadT .or. NewY < RadT) then ! Confirms the particle is within the bounds of the box
                if (TempX >= GridSize-RadT-RadT) then
                    OverlapCount = 0
                    TempX = GridSize-RadT
                    EdgeCase = .TRUE.
                    GO TO 50
                elseif (TempX <= RadT+RadT) then
                    OverlapCount = 0
                    TempX = RadT
                    EdgeCase = .TRUE.
                    GO TO 50
                end if
                GO TO 10
            end if

            MlxReal(MolNo) = NewX        ! Saves the particle location
            MlyReal(MolNo) = NewY
            MLr(MolNo) = RadT
            FullCount = 0
            MolNo = MolNo + 1

            RA = '0'
            RAMolClose = 0
        end if
```

```
10          CONTINUE

      else
            Full = 1
      end if

      end

      subroutine PointSafe ! Determines if the falling particle has impacted yet
      use allSubs ! Loads the variables from the module
      integer a, b
      real Dist
      character t

      Hit = .FALSE.
      FullCheck = .FALSE.

      ! Checks the distance between the current falling particle location and previously
placed partice to determine if it has impacted

      cloop: do a = 1, MolNo – 1
            Dist = ((MlxReal(a)-x)**2)+((MlyReal(a)-y)**2)
            Dist = sqrt(Dist)
            if (Dist <= ((RadT)+(Mlr(a)))) then
                  Hit = .TRUE.
                  exit cloop
            end if
      end do cloop

      ! If the impact is above the top of the box, a counter is incremented to show the box
may be full
      if ((Hit .eqv. .TRUE.) .and. (y >= (GridSize – RadT))) then
            Hit = .FALSE.
            FullCount = FullCount + 1
            RoofCount = RoofCount + 1
      end if

      if (RoofCount >= 2500) then
            FullCheck = .TRUE.
      end if

      if (FullCount == 2500000) then
            Full = 1
      end if

      end
```

Appendix 2: Percolation Chain Detection Algorithm

This appendix contains the algorithm used to investigate both 2D and 3D systems for percolation chains. As it only looks at the contacts between the particles, the number of dimensions is irrelevant to it, excluding when importing file data.

```
Module VarList ! Initalises variables to be used across all functions
real, dimension (😊, allocatable :: MLxReal
real, dimension (:), allocatable :: MLyReal
!real, dimension (:), allocatable :: MLzRea– - Needs to be included for 3D systems
integer, dimension (:), allocatable :: MLr
integer, dimension (:,:), allocatable :: Contacts
integer, dimension (:), allocatable :: Visited
integer, dimension (:), allocatable :: TotShap
integer, dimension (:,:), allocatable :: Edge
integer, dimension (:), allocatable :: EdgeCount
integer :: n
integer :: nlines
integer :: GridSize
end module

program VoidCalcs
use VarList ! Loads module variables

! Initialises local variables and sets their starting values
character FileName*15, t*1
integer nlinesB, Depth, ShapeCount, CurCont, y, m
integer, dimension (1:250) :: Path
integer, dimension (1:20000,1:250) :: SetPath
integer, dimension (1:20000,1:250) :: Shapes
integer, dimension (1:20000) :: ShapesPrint
logical Found
integer, dimension (:), allocatable :: LocalShapeCount
integer, dimension (:), allocatable :: TotCont
integer, dimension (1:20000) :: ShapeLength
integer :: RadLarge
integer :: count
integer :: x
integer :: LoopCount
integer :: ProgCount
integer :: PCId
character, dimension (200) :: FileID*3
logical :: fileexists
integer :: OutCount

do ProgCount = 1, 50
        write(FileID(ProgCount)‘ '(i’)') ProgCount
end do

do ProgCount = 1, 50 ! Loop for number of files to be investigated

RadLarge = 0
```

```
GridSize = 0
m = 0
x = 0
n = 0
nlines = 0
nlinesB = 0
ShapeCount = 1
CurCont = 0
y = 0

FileName ‘’"
FileName = tri177djusttl(FileID(ProgCount))) /‘ '.c’v'

INQUIRE(File=FileName, EXIST=fileexists)

if (fileexists .eqv. .FALSE.) then
        GO TO 50
end if
! Confirms the files exists and if so loads it, if not skips it and moves to the next
open(1, file = FileName, statu’='o’d')

do
        read(1, *,iostat=io)
        if (io/=0) EXIT
        nlines=nlines+1
end do
close(1)
! Determines the number of rows in that files which is equivalent to the number of
particles in the system
! Allocates the arrays using this value
allocate(MLyReal(1:nlines))
allocate(MLxReal(1:nlines))
! allocate(MLzReal(1:nlines)– - To be included for 3D system files
allocate(MLr(1:nlines))

allocate(Visited(1:nlines))
allocate(LocalShapeCount(1:nlines))
allocate(Edge(1:nlines,1:4))
allocate(EdgeCount(1:nlines))
allocate(TotCont(1:nlines))
allocate(TotShap(1:nlines))

! Sets initial value of those arrays
MLyReal = 0
MLxReal = 0
! MLzReal = – - To be included for 3D systems
MLr = 0
Shapes = 0
ShapeLength = 0
Visited = 0
SetPath = 0
Path = 0
LocalShapeCount = 0
OneCont = 0
TotCont = 1
TotShap = 0
Edge = 0
```

```
	EdgeCount = 1
	OutCount = 0

	open(1, file = FileName, statu’='o’d')
	do n = 1, nlines
		read (1,*) MLxReal(n), MLyReal(n), MLr(n) ! Include MLzReal(n), inbetween the y and radius inputs for 3D systems
	end do
	close(1)
	! Loads in the contacts file for the system
	FileName ‘”'
	FileName ‘ 'contac’s' // tri178djusttl(FileID(ProgCount))) /‘ '.c’v'
	open(2, file = FileName, statu’='o’d')

	do
		read(2, *,iostat=io)
		if (io/=0) EXIT
		nlinesB = nlinesB + 1
	end do
	close(2)
	allocate(Contacts(1:nlinesB,20))
	Contacts = 0

	open(2, file = FileName, statu’='o’d')
	do n = 1, nlinesB
		read (2,*) Contacts(n,1), Contacts(n,2), Contacts(n,3), Contacts(n,4), Contacts(n,5), Contacts(n,6), Contacts(n,7)
	end do
	close(2)

	do n = 1, nlines
		if (MLr(n) > RadLarge) then
			RadLarge = MLr(n)
		end if
	end do
	GridSize = (((RadLarge*6))*5)
	! Determines the largest radius present and the grid size

	! Calculates how many particle contacts each particle has
	do n = 1, nlinesB
		do while (Contacts(n,TotCont(n)) /= 0)
			TotCont(n) = TotCont(n) + 1
		end do

		TotCont(n) = TotCont(n– - 1
		TotShap(n) = TotCont(n)
	end do

	do n = 1, nlines ! Loops through each particle
		if (TotShap(n) > 0 .and. MLxReal(n) == MLr(n)) then ! Checks that the current particle has at least one contact and is touching the leftmost edge of the hox
			DepthLoop: do LoopCount = 1, 50
10				CONTINUE
				MaxDepth = 250		! Maximum number of recursions that can occur before automatically unwinding
				Visited = 0
				CurCont = n
```

```
                                        y = 1
                                        Depth = 1
                                        Found = .FALSE.

                                        if (LocalShapeCount(n) >= TotShap(n)) then
                                                  exit DepthLoop
                                        elseif (Mlr(n) == 10) then          ! Confirms the particle is a
large one
                                                  exit DepthLoop
                                        end if

                                        ! Enters the recursion
                                        call
Searching(n,Depth,CurCont,y,ShapeCount,Found,MaxDepth,Path,SetPath,Shapes,ShapeLe
ngth,OutCount)

                                        if (Found .eqv. .TRUE.) then        ! If a shape has been found,
increments the number of shapes
                                                  ShapeCount = ShapeCount + 1
                                        end if
                                        OutCount = 0
                              end do DepthLoop
                    end if
          end do

          ! Saves the shapes to file
          FileName ‘”'
          FileName ‘ 'shap’s' // tri179djusttl(FileID(ProgCount))) /‘ '.c’v'
          open(3, file = FileName, statu’='n’w')
          do n = 1, ShapeCount
                    do m = 1, 250
                              write(’,'(I3,A1,’)', advanc’=''o') SetPath(n,m)‘ ’,'
                    end do
                    write(3, *‘”'
          end do
          close(3)

          ! Deallocates the arrays so they can be reallocated with the correct length for the next
file
          deallocate(MLyReal)
          deallocate(MLxReal)
          ! deallocate(MLzReal– - For 3D systems
          deallocate(MLr)

          deallocate(Visited)
          deallocate(LocalShapeCount)
          deallocate(Edge)
          deallocate(EdgeCount)
          deallocate(TotCont)
          deallocate(TotShap)

          deallocate(Contacts)

50        CONTINUE

          end do
```

```
end program

RECURSIVE SUBROUTINE
Searching(n1,Depth1,CurCont1,y1,ShapeCount1,Found,MaxDepth,Path1,SetPath1,Shapes
1,ShapeLength1,OutCount1)
use VarList     ! Loads module variables and initalises local recursion variables
integer, intent(inout) :: Depth1
integer, intent(inout) :: CurCont1
integer, intent(inout) :: y1
integer, intent(inout) :: ShapeCount1
integer, intent(in) :: n1
integer, intent(in) :: MaxDepth
integer, dimension (1:250), intent(inout) :: Path1
integer, dimension (1:20000,1:250), intent(inout) :: SetPath1
integer, dimension (1:20000,1:250), intent(inout) :: Shapes1
integer, dimension (1:20000), intent(inout) :: ShapeLength1
integer :: OutCount1
integer :: SavCont
integer :: x
integer :: count
integer :: count2
integer :: count3
integer :: PathCount
integer :: NewCont
logical, intent(inout) :: Found
logical :: LT
integer :: Dupli
integer :: Insi
integer :: InsiCount
character :: t

SavCont = CurCont1
Visited(SavCont) = 1
x = 1
y1 = 1
count = 0
count2 = 0
count3 = 0
PathCount = 0
Found = .FALSE.
Dupli = 0
Insi = 0
InsiCount = 0
OutCount1 = OutCount1 + 1
if (Depth1 <= MaxDepth) then ! Confirms the recursion has not gone too deep

llop: do x = 1, TotShap(CurCont1)     ! Loops through the current particle contacts

    LT = .FALSE.
    if (x == 1) then
        ThisLoop: do count = 1, Depth1-1 ! Makes the path travelled through
ordered in ascending particle numbers
            if (CurCont1 < Path1(count)) then
                LT = .TRUE.
                exit ThisLoop
            end if
        end do ThisLoop
```

```
                if (LT .eqv. .TRUE.) then
                    do count2 = Depth1, count+1, -1
                        Path1(count2) = Path1(count2-1)
                    end do
                    Path1(count) = CurCont1
                else
                    Path1(Depth1) = CurCont1
                end if
            end if

            if (MLr(Contacts(CurCont1,x)) == 10) then        ! If the connected particle is small, then it is skipped
                GO TO 20
            else
                NewCont = Contacts(CurCont1,x)
                if (OutCount1 > 100000) then ! Looped for too many times and may be stuck so exits the outer loop
                    exit llop
                end if

                if (MLxReal(NewCont) == GridSize-MLr(NewCont)) then ! Current contact is on the right hand edge of the box, in 3D systems, all instances of MLxReal can be swapped for MLzReal to look for chains crossing in the perpendicular direction
                    if (ShapeCount1 > 1) then ! Determining if the chain has already been found
                        PathCount = 0
                        Dupli = 0
                        ShapeLoop: do count = 1, ShapeCount1-1
                            PathCount = 0
                            do count2 = 1, ShapeLength1(count)
                                do count3 = 1, Depth1
                                    if (Shapes1(count,count2) == Path1(count3)) then
                                        PathCount = PathCount + 1
                                    end if
                                end do
                            end do
                            if (PathCount == ShapeLength1(count)) then
                                Dupli = 1
                                exit ShapeLoop
                            end if
                        end do ShapeLoop
                    end if

                    if (ShapeCount1 == 1 or. Dupli == 0) then        ! If unique chain (or the first one) then it is saved as the recursion unwinds
                        LT = .FALSE.
                        ThatLoop2: do count = 1, y1-1
                            if (SavCont < Shapes1(ShapeCount1,count)) then
                                LT = .TRUE.
                                exit ThatLoop2
                            end if
                        end do ThatLoop2
```

```
                                        if (LT .eqv. .TRUE.) then
                                             do count2 = y1, count+1, -1
                                                  Shapes1(ShapeCount1,count2)    =
Shapes1(ShapeCount1,count2-1)
                                             end do
                                             Shapes1(ShapeCount1,count) = SavCont
                                        else
                                             Shapes1(ShapeCount1,y1) = CurCont1
                                        end if
                                        SetPath1(ShapeCount1,y1) = SavCont
                                        y1 = y1 + 1
                                        ShapeLength1(ShapeCount1) = Depth1
                                        Found = .TRUE.
                                        exit llop
                                   else      ! If it has already been found the recursion continues
                                        if (Visited(NewCont) /= 1) then
                                             Depth1 = Depth1 + 1
                                             call
Searching(n1,Depth1,NewCont,y1,ShapeCount1,Found,MaxDepth,Path1,SetPath1,Shapes1
,ShapeLength1,OutCount1)
                                        end if
                                   end if
                              else      ! If not at the right hand edge, the recursion continues to the
next contact
                                   if (Visited(NewCont) /= 1) then
                                        Depth1 = Depth1 + 1
                                        call
Searching(n1,Depth1,NewCont,y1,ShapeCount1,Found,MaxDepth,Path1,SetPath1,Shapes1
,ShapeLength1,OutCount1)

                                        if (Found .eqv. .TRUE.) then
                                             LT = .FALSE.
                                             ThatLoop3: do count = 1, y1-1
                                                  if            (SavCont            <
Shapes1(ShapeCount1,count)) then
                                                       LT = .TRUE.
                                                       exit ThatLoop3
                                                  end if
                                             end do ThatLoop3

                                             if (LT .eqv. .TRUE.) then
                                                  do count2 = y1, count+1, -1

     Shapes1(ShapeCount1,count2) = Shapes1(ShapeCount1,count2-1)
                                                  end do
                                                  Shapes1(ShapeCount1,count)      =
SavCont
                                             else
                                                  Shapes1(ShapeCount1,y1)         =
CurCont1
                                             end if
                                             SetPath1(ShapeCount1,y1) = SavCont
                                             y1 = y1 + 1
                                             exit llop
                                        end if
                                   end if
                              end if
```

```
                end if

20              CONTINUE
        end do llop

        end if

        Visited(SavCont) = 0    ! Unwinding the recursion, marks the particle as no longer visited, and removes the particle from the path
        do count = 1, Depth1
                if (Path1(count) == SavCont) then
                        do count2 = count, Depth1-1
                                Path1(count2) = Path1(count2+1)
                        end do
                        Path1(Depth1) = 0
                        exit
                end if
        end do

        Depth1 = Depth1 - 1

        END SUBROUTINE Searching
```

Appendix 3: 3D System Algorithm

This appendix contains the algorithm used for creating 3D systems.

```
module allSubs ! Initialises the variables used through all functions
character, dimension (:,:,:), allocatable, public :: RA*4
integer MolNo, RadLarge, RadSmall, BoxSize, GridSize, Rads, count, SN, Full,
FullCount, OneLegacyCount
integer MLr, AllocateVal, RoofCount
real MLxReal, MLyReal, MLzReal
integer, dimension (:), allocatable, public :: OneLegacyCounterCount
integer, dimension (:,:), allocatable, public :: Ones
real, dimension (:,:,:), allocatable, public :: OnesLegacy
integer, dimension (:,:), allocatable, public :: Contacts
real, dimension (:,:,:), allocatable, public :: yVal
integer, dimension (:,:), allocatable, public :: OrderyVal
integer, dimension (:,:), allocatable, public :: FinalTriCombi
dimension MLr(10000), Rads(10), MLxReal(10000), MLyReal(10000),
MLzReal(10000)
logical Hit, FullCheck, FirstCusps, RoofHit
integer x, y, z, RadT, RestartNo
integer HMAllo
end module allSubs

program packedbed
use allSubs ! Loads the variables from the module

! Initialises local variables
character t, FileName*15, FileID*3

integer m, n, o, check, PrintNo, iSeed, ProgCount, PCId
dimension FileID(200)
dimension iSeed(50)
real RX, Dist, DistCheck
real PartArea, VoidArea, VoidFrac, Pi
integer count2, TotLength, count3, RunAmo, RunRedo

logical Cont, Impact

MLxReal = 0
MLyReal = 0
MLzReal = 0
MLr = 0
RunAmo = 250
RunRedo = 0

FirstCusps = .FALSE.

Dist = 0
DistCheck = 0

RestartNo = 0

PartArea = 0
```

```
        VoidArea = 0
        VoidFrac = 0
        Pi = 3.141596535
        TotLength = 0
        MolNo = 1
        OneLegacyCount = 1
        FileName = ''
        Full = 0
        check = 0
        FullCount = 0
        Impact = .FALSE.
        count3 = 0
        count = 0
        count2 = 0

        t = 'y'

        if (t == 'y') then
                Rads(1) = 10 ! Sets the radius of a particle. Additional radii would be inputted
as Rads(x) = 'Radius'
                SN = 1 ! Sets the number of different radii in the system

                RadLarge = 0
                RadSmall = 0

                do count = 1, SN
                        if (RadLarge < Rads(count)) then
                                RadLarge = Rads(count)
                        end if
                        if (RadSmall > Rads(count) .or. RadSmall == 0) then
                                RadSmall = Rads(count)
                        end if
                end do

                RadLarge = 10
                RadSmall = 10

                ! Calculates the box size based on the largest radius present
                BoxSize = (RadLarge*6)
                GridSize = (BoxSize*2)
                AllocateVal = (((BoxSize*3)**2)*2)
                HMAllo = GridSize + (2*RadLarge)

                ! Allocates the arrays
                allocate(RA(1:HMAllo, 1:GridSize, 1:GridSize))

                allocate(Ones(1:AllocateVal,1:3))
                allocate(FinalTriCombi(1:AllocateVal,1:3))

                allocate(yVal(1:GridSize,1:GridSize,2))
                allocate(OrderyVal(1:GridSize*GridSize,2))

                do ProgCount = 1, RunAmo
                        write(FileID(ProgCount), '(i0)') ProgCount
                end do

                do ProgCount = 1, 50 ! Starts the loop for the number of systems to be created
```

```
            PCId = ProgCount

            ! Sets variables intial values
            MLxReal = 0
            MLyReal = 0
            MLzReal = 0
            MLr = 0

            FirstCusps = .FALSE.

            Dist = 0
            DistCheck = 0

            RestartNo = 0

            PartArea = 0
            VoidArea = 0
            VoidFrac = 0
            TotLength = 0
            Pi = 3.141596535
            MolNo = 1
            OneLegacyCount = 1
            FileName = ''
            Full = 0
            check = 0
            FullCount = 0
            RoofCount = 0
            RoofHit = .FALSE.
            Impact = .FALSE.
            count3 = 0
            count = 0
            count2 = 0

            Ones = 0
!           OnesLegacy = 0
!           OneLegacyCounterCount = 0
            yVal = 0

            RA = '0'

            call random_seed()

            do while (count < 1000000)

                  Impact = .FALSE.

                  ! Picks a random radius and x and z coordinates, and sets y
to be on the bottom of the box
                  call random_number(RX)
                  count2 = 1 + floor(SN*RX)
                  RadT = Rads(count2)

                  call random_number(RX)
                  count2 = 1 + floor((GridSize-(2*RadT))*RX)
                  x = count2+RadT

                  call random_number(RX)
```

```
                count2 = 1 + floor((GridSize-(2*RadT))*RX)
                z = count2+RadT

                y = RadT

                if (MolNo > 1) then        ! Checks there is already at least
one particle in the system
                    hitloop: do count3 = 1, MolNo - 1
                        Dist         =         ((MLxReal(count3)-
x)**2)+((MLyReal(count3)-y)**2)+((MLzReal(count3)-z)**2)
                        Dist = sqrt(Dist)
                        if (Dist <= ((RadT)+(Mlr(count3)))) then
                            count = count + 1
                            Impact = .TRUE.
                            exit hitloop
                        end if
                    end do hitloop
                    if (Impact .eqv. .FALSE.) then ! If the particle is not
overlapping with any others and is inside the grid, its location is saved
                        count = 0
                        MLxReal(MolNo) = x
                        MlyReal(MolNo) = y
                        MlzReal(MolNo) = z
                        MLr(MolNo) = RadT
                        MolNo = MolNo + 1
                    end if
                else
                    MlxReal(MolNo) = x
                    MlyReal(MolNo) = y
                    MlzReal(MolNo) = z
                    MLr(MolNo) = RadT
                    MolNo = MolNo + 1
                end if
            end do

            ! This loops through the base line to check that there is nowhere a
small particle could fall through to the bottom of the box, and if so, places a particle there
            do m = RadSmall, GridSize-RadSmall
                do n = RadSmall, GridSize-RadSmall
                    Impact = .FALSE.
                    hitloop2: do count3 = 1, MolNo – 1
                        Dist         =         ((MlxReal(count3)-
m)**2)+((MlzReal(count3)-n)**2)
                        Dist = sqrt(Dist)
                        if (Dist <= ((RadSmall)+(Mlr(count3)))) then
                            count = count + 1
                            Impact = .TRUE.
                            exit hitloop2
                        end if
                    end do hitloop2
                    if (Impact .eqv. .FALSE.) then
                        count = 0
                        MlxReal(MolNo) = m
                        MlyReal(MolNo) = RadSmall
                        MlzReal(MolNo) = n
                        MLr(MolNo) = RadSmall
                        MolNo = MolNo + 1
```

```
                        end if
                end do
        end do

        do n = 1, 2500 ! Loops for each particle being added to the system,
using the main function. At the end of each loop, it checks if the box is full and if so, leaves the
loop.
                Call molpos
                if (Full == 1) then
                        exit
                elseif (RoofHit .eqv. .TRUE.) then
                        exit
                end if
        end do

        t = ‘y’
        if (t == ‘y’ .and. RoofHit .eqv. .TRUE.) then ! Saves the particle
locations to a file
                FileName = ‘’
                FileName = trim(188nitia(FileID(PCId))) // ‘.csv’
                open(1, file = FileName, status = ‘new’)
                do y = 1, MolNo-1
                        write(1,*) MlxReal(y), ‘,’ , MlyReal(y), ‘,’ , MlzReal(y),
‘,’ , MLr(y)
                end do
                close(1)
        else
                RunAmo = RunAmo + 1
                RunRedo = RunRedo + 1
        end if

        t = ‘y’
        if (t == ‘y’ .and. RoofHit .eqv. .TRUE.) then ! Saves the particle
contacts to a file
                allocate(Contacts(1:MolNo, 1:MolNo))
                Contacts = 0
                do y = 1, MolNo-1
                        n = 1
                        do x = 1, MolNo-1
                                if (x /= y) then
                                        Dist = ((MlxReal(y) –
MlxReal(x))**2) + ((MlyReal(y) – MlyReal(x))**2) + ((MlzReal(y) – MlzReal(x))**2)
                                        Dist = sqrt(Dist)
                                        if (Dist <= (MLr(y) + MLr(x) + 0.01))
then
                                                Contacts(y,n) = x
                                                n = n + 1
                                        end if
                                        if (n > TotLength) then
                                                TotLength = n
                                        end if
                                end if
                        end do
                end do

                FileName = ‘’
                FileName = ‘contacts’ // trim(188nitia(FileID(PCId))) // ‘.csv’
```

```
                    open(3, file = FileName, status = 'new')
                    do y = 1, MolNo-1
                        do x = 1, TotLength
                            write(3,'(I4,A1,X)',            advance='no')
Contacts(y,x), ','
                        end do
                        write(3, *) ''
                    end do
                    close(3)
                    deallocate(Contacts)
                end if
            end do

            FileName = ''   ! Outputs any files that failed and were not saved
            FileName = 'redo.csv'
            open(5, file = FileName, status = 'new')
            write(5,'(I4,A1,X)', advance='no') RunAmo
            write(5,'(I4,A1,X)', advance='no') RunRedo
            close(5)
        end if

        end program


        subroutine molpos
        use allSubs ! Loads the variables from the module

        ! Sets up the local variables
        integer count2, Spot, Height, DoubRad, RowRad, a,b,c, TempX, TempZ, RadIn, m, n,
o, Width
        real TempY
        real RX, MidWay, MidWayZ, Dist, RowRadReal
        character t, FileName*15
        integer SavOneX, SavOneY, SavOneZ, SavDist, OneCount
        integer Zrad, ZdoubRad
        integer MlxCor, MlyCor, MlzCor
        logical SafeLocFound, CuspFound
        integer OldX, OldZ, FinCount
        real Grad1a, Grad2a, Grad1b, Grad2b, CurrentY, ZradReal, MidWayReal
        integer FinalPartNo
        real FinalPartDist
        dimension FinalPartNo(5), FinalPartDist(5,2)
        real EquA1, EquA2, EquA3, EquB1, EquB2, EquB3, EquC1, EquC2, EquC3, EquD1,
EquD2, EquD3, NewX, NewY, NewZ, NewYa, NewYb
        real EquValuesA, EquValuesB, k1, k2, Outputs
        dimension EquValuesA(3,3), EquValuesB(3), Outputs(20)
        integer EdgeSide
        logical EdgeCase, NewXVal, NewZVal
        real PartCoords, stochDists, sumDist, dx, stochxnew, newCoords, stochynew
        integer ibad
        dimension PartCoords(3,3), stochDists(3), newCoords(3), dx(3), stochxnew(3)
        real FinalDists, FinalCuspsSaved
        dimension FinalCuspsSaved(15000,4)
        dimension FinalDists(3), CuspHighList(16,3)
        real CuspHighList, FurthDist
        integer CuspLowCount, CuspHighCount, CuspLowMoveTo
        logical OuterLayer
```

```
      real TriCheckA, TriCheckB, TriCheckC
      logical TriCheckInside
      real LowYVal
      integer LowYLoc
      dimension LowYLoc(2)
      real Sempi, SetTri, TriA, TriB, TriC, SempiA, SempiB, SempiC
      real DistAN, DistBN, DistCN, DistAB, DistBC, DistCA
      real EdgeLowVal
      integer EdgeLowA, EdgeLowB, bSide1, bSide2
      logical InitialCusp
      integer TriCheckCount, TriCheckNo, TriCount, TriCombi, TriAttempt, ReTriAttempt
      logical FirstTri, FirstEdge
      real TriCoords, OverDist
      integer OverDistNo
      dimension TriCheckNo(10000), TriCombi(100001,3), TriCombi2(10000000,3),
TriCoords(3,3)
      dimension EdgeCombi(1000000)
      integer EdgeCount, EdgeCombi, EdgeAttempt
      integer SideCheckCount, SideCheckNo, SideCount, SideCombi, SideAttempt
      integer FinalTriCount
      logical FirstSide
      real SideCoords
      dimension SideCheckNo(10000), SideCombi(100001,3), SideCoords(3,3)
      integer OverlapCount, HMSize
      real DistCheck1, DistCheck2, DistCheck3, DistCheck1Val, DistCheck2Val,
DistCheck3Val

      FinalCuspsSaved = 0

40    CONTINUE

      RestartNo = RestartNo + 1

      ! Setting initial values of variables
      Hit = .FALSE.
      EdgeSide = 0

      HMSize = 0

      OverDist = 0
      OverDistNo = 0

      OverlapCount = 0


      DistCheck1 = 0
      DistCheck2 = 0
      DistCheck3 = 0
      DistCheck1Val = 0
      DistCheck2Val = 0
      DistCheck3Val = 0

      CuspLowCount = 0
      CuspHighCount = 0
      FurthDist = 0
      CuspLowMoveTo = 0
```

```
TriCheckNo = 0
TriCheckCount = 1
TriCount = 1
EdgeCount = 1
TriCombi = 0
TriCombi2 = 0
EdgeCombi = 0
TriCoords = 0
TriAttempt = 0
EdgeAttempt = 0
FirstTri = .TRUE.
FirstEdge = .TRUE.
ReTriAttempt = 0

FinalTriCombi = 0
FinalTriCount = 1


SideCheckNo = 0
SideCheckCount = 1
SideCount = 1
SideCombi = 0
SideCoords = 0
SideAttempt = 0
FirstSide = .TRUE.

Sempi = 0
SetTri = 0
TriA = 0
TriB = 0
TriC = 0
SempiA = 0
SempiB = 0
SempiC = 0
DistAN = 0
DistBN = 0
DistCN = 0
DistAB = 0
DistBC = 0
DistCA = 0

LowYLoc = 0
LowYVal = 0

FinalDists = 0

PartCoords = 0
stochDists = 0
sumDist = 0
dx = 0
stochxnew = 0
newCoords = 0
stochynew = 0

NewXVal = .FALSE.
NewZVal = .FALSE.
```

```
EquValuesA = 0
EquValuesB = 0
k1 = 0
k2 = 0
Outputs = 0

TRP1Swap = 0
TRP2Swap = 0

ZradReal = 0
MidWayReal = 0

FinCount = 0

CuspFound = .FALSE.
CurrentY = 0
OldX = 999
OldZ = 999

SavDist = 0
SavOneX = 0
SavOneY = 0

MidWay = 0
MidWayZ = 0

Grad1a = 0
Grad2a = 0
Grad1b = 0
Grad2b = 0

FinalPartNo = 999999
FinalPartDist = 999999

EquA1 = 0
EquA2 = 0
EquA3 = 0

EquB1 = 0
EquB2 = 0
EquB3 = 0

EquC1 = 0
EquC2 = 0
EquC3 = 0

EquD1 = 0
EquD2 = 0
EquD3 = 0

NewX = 0
NewY = 0
NewZ = 0
NewYa = 0
NewYb = 0

RowRadReal = 0
```

```
EdgeCase = .FALSE.

! Randomly chooses which radius will be used for this particle
call random_number(RX)
count2 = 1 + floor(2*RX)
if (count2 == 2) then
        RadT = RadLarge
else
        RadT = RadSmall
end if

wloop: do while (Hit .eqv. .FALSE.)

        ! Randomly chooses the x value
        call random_number(RX)
        Spot = 1 + floor((GridSize-(2*RadT))*RX)
        x = Spot+RadT

        call random_number(RX)
        Spot = 1 + floor((GridSize-(2*RadT))*RX)
        z = Spot+RadT

        do y = GridSize, RadT, -1          ! Loops from the top of the box, and sends to
the function to determine impact

                call PointSafe

                ! If the box is full or an impact has I, the loop is exited

                if (Full == 1) then
                        exit wloop
                end if

                if (FullCheck .eqv. .TRUE.) then
                        RoofHit = .TRUE.
                        GO TO 10
                end if

                if (Hit .eqv. .TRUE.) then
                        exit wloop
                end if
        end do
end do wloop

if (Hit .eqv. .TRUE. .and. Full /= 1) then
        if (MolNo > 1) then
‘       ’A = '0'

                ! Sets up the variables to be used for particle placement

                Ones = 0
                OneCount = 1

                do count = 1,–MolNo - 1! Loops through the particles for contour plot
placement
```

```
MLxcor = MLxReal(count)
MLycor = MLyReal(count)
MLzcor = MLzReal(count) ! Takes the radius, x, y and z
coordinates of the current particle in the loop
RadIn = Mlr(count)

DoubRad = (RadIn+RadT)+1
ZDoubRad = (RadIn+RadT)+1

do Width = -ZDoubRad, 0
	MidWayZ = ZDoub–ad**2 - Width**2
	ZRad = abs(sqrt(MidwayZ))
	ZRadReal = abs(sqrt(MidwayZ))

	do Height = -ZRad, 0     ! Draws the particle onto the
contour“p“ot, "-"s marking blocked loca“i”ns, "1"s being valid spots

		Midway = Z–ad**2 - Height**2
		RowRad = abs(sqrt(MidWay))
		MidWayReal = ZRadR–al**2 - Height**2
		RowRadReal = abs(sqrt(MidWayReal))

		do count2 = -RowRad+1,RowRad-1
			if (MLyCor+count2<=HMAllo .and.
MLyCor+count2>=RadT) then
				if (MLxCor+Height>=RadT
.and. MLzCor+Width >= RadT) then

	RA(MLyCor+count2, MLxCor+Height, MLzCor+Wi‘t‘) = '-'
				end if
				if (MLxCor-
Height<=GridSize-RadT .and. MLzCor+Width >= RadT) then

	RA(MLyCor+count2, MLxCor-Height, MLzCor+Wi‘t‘) = '-'
				end if
				if (MLxCor+Height>=RadT
.and. MLzCor-Width <=GridSize-RadT) then

	RA(MLyCor+count2, MLxCor+Height, MLzCor-Wi‘t‘) = '-'
				end if
				if (MLxCor-
Height<=GridSize-RadT .and. MLzCor-Width <=GridSize-RadT) then

	RA(MLyCor+count2, MLxCor-Height, MLzCor-Wi‘t‘) = '-'
				end if
			end if
		end do

		do count2 = -RowRad,RowRad ! Adds the
valid spots for the resting particle to be placed
			if (MLyCor+count2>=RadT .and.
MLyCor+count2<=HMAllo) then
				if (MLxCor+Height>=RadT
.and. MLzCor+Width >= RadT) then
					if
(RA(MLyCor+count2, MLxCor+Height, MLzCor+Wid‘h‘ /= '-')  then
```

```
RA(MLyCor+count2, MLxCor+Height, MLzCor+Wi‘t’) = '1'
end if
end if
if (MLxCor-Height<=GridSize-RadT .and. MLzCor+Width >= RadT) then
if (RA(MLyCor+count2, MLxCor-Height, MLzCor+Wid‘h‘ /= '-') then
RA(MLyCor+count2, MLxCor-Height, MLzCor+Wi‘t’) = '1'
end if
end if
if (MLxCor+Height>=RadT .and. MLzCor-Width <= GridSize-RadT) then
if (RA(MLyCor+count2, MLxCor+Height, MLzCor-Wid‘h‘ /= '-') then
RA(MLyCor+count2, MLxCor+Height, MLzCor-Wi‘t’) = '1'
end if
end if
if (MLxCor-Height<=GridSize-RadT .and. MLzCor-Width <= GridSize-RadT) then
if (RA(MLyCor+count2, MLxCor-Height, MLzCor-Wid‘h‘ /= '-') then
RA(MLyCor+count2, MLxCor-Height, MLzCor-Wi‘t’) = '1'
end if
end if
end if
end do

if (MLyCor+RowRadReal>=RadT .and. MLyCor+RowRadReal<=HMAllo) then
if (MLxCor+Height>=RadT .and. MLzCor+Width >= RadT) then
if (RA(MLyCor+count2, MLxCor+Height, MLzCor+Wid‘h‘ /= '-')  then
if (MLyCor+RowRadReal > yVal(MLxCor+Height, MLzCor+Width,1)) then
yVal(MLxCor+Height, MLzCor+Width,1) = MLyCor+RowRadReal
yVal(MLxCor+Height, MLzCor+Width,2) = count
end if
end if
end if
if (MLxCor-Height<=GridSize-RadT .and. MLzCor+Width >= RadT) then
if (RA(MLyCor+count2, MLxCor-Height, MLzCor+Wid‘h‘ /= '-') then
if (MLyCor+RowRadReal > yVal(MLxCor-Height, MLzCor+Width,1)) then
yVal(MLxCor-Height, MLzCor+Width,1) = MLyCor+RowRadReal
yVal(MLxCor-Height, MLzCor+Width,2) = count
end if
```

```
end if
end if
if (MLxCor+Height>=RadT .and.
MLzCor-Width <= GridSize-RadT) then
if (RA(MLyCor+count2,
MLxCor+Height, MLzCor-Wid‘h‘ /= '-') then
if
(MLyCor+RowRadReal > yVal(MLxCor+Height, MLzCor-Width,1)) then

yVal(MLxCor+Height, MLzCor-Width,1) = MLyCor+RowRadReal

yVal(MLxCor+Height, MLzCor-Width,2) = count
end if
end if
end if
if (MLxCor-Height<=GridSize-RadT
.and. MLzCor-Width <= GridSize-RadT) then
if (RA(MLyCor+count2,
MLxCor-Height, MLzCor-Wid‘h‘ /= '-') then
if
(MLyCor+RowRadReal > yVal(MLxCor-Height, MLzCor-Width,1)) then

yVal(MLxCor-Height, MLzCor-Width,1) = MLyCor+RowRadReal

yVal(MLxCor-Height, MLzCor-Width,2) = count
end if
end if
end if
end if
end do
end do
end do

do a = 1, GridSize ! Finds the valid points and saves them to an array
do b = 1, HMAllo
do c = 1, GridSize
if (RA(b,a‘c’ == '1') then
Ones(OneCount, 1) = b
Ones(OneCount, 2) = a
Ones(OneCount, 3) = c
OneCount = OneCount + 1
end if
end do
end do
end do

if (FirstCusps .eqv. .FALSE.) then
FirstCusps = .TRUE.
c = 1
do a = RadT, GridSize-RadT
do b = RadT, GridSize-RadT
if (yVal(a+1,b,1) > yVal(a,b,1) .and. yVal(a-
1,b,1) > yVal(a,b,1)) then
if (yVal(a,b+1,1) > yVal(a,b,1) .and.
yVal(a,b-1,1) > yVal(a,b,1)) then
if (yVal(a+1,b+1,1) >
yVal(a,b,1) .and. yVal(a-1,b-1,1) > yVal(a,b,1)) then
```

```
if (yVal(a-1,b+1,1) > yVal(a,b,1) .and. yVal(a+1,b-1,1) > yVal(a,b,1)) then

FinalCuspsSaved(c,1) = a

FinalCuspsSaved(c,2) = b

FinalCuspsSaved(c,3) = yVal(a,b,1)

FinalCuspsSaved(c,4) = yVal(a,b,2)
                    c = c + 1
                end if
            end if
        end if
    end if
    end do
  end do
end if

do a = 1, On–Count - 1 ! Finds the closest of these points to the impact location and moves the particle to it
    TempX = Ones(a,2)
    TempY = Ones(a,1)
    TempZ = Ones(a,3)

    if (TempY <= y+1) then
        Dist = ((x-TempX)**2)+((y-TempY)**2)+((z-TempZ)**2)
        Dist = sqrt(Dist)
        if (Dist < SavDist .or. SavDist == 0) then
            SavDist = Dist
            SavOneX = Ones(a,2)
            SavOneY = Ones(a,1)
            SavOneZ = Ones(a,3)
        end if
    end if
end do

TempX = SavOneX
TempY = SavOneY
TempZ = SavOneZ

if (TempX == 0 .and. TempY == 0 .and. TempZ == 0) then
    GO TO 10
end if

SafeLocFound = .FALSE.

FinCount = 0
LR = 0
step = 1

FindLoop: do while (SafeLocFound .eqv. .FALSE.) ! Rolling algorithm, moves in the direction with the deepest slope until it is fully surrounded by higher points
    CurrentY = yVal(TempX,TempZ,1)
    if (TempX < RadT .or. TempZ < RadT .or. TempX > GridSize-RadT .or. TempZ > GridSize-RadT) then
```

```
                        FullCount = FullCount + 1
                        GO TO 40
                    end if
                    if (TempX-1 < RadT .or. TempZ-1 < RadT .or. TempX+1 > GridSize-RadT .or. TempZ+1 > GridSize-RadT) then
                        EdgeLoop: do while (CuspFound .eqv. .FALSE.)
                            CurrentY = yVal(TempX,TempZ,1)
                            if (TempX-1 < RadT .and. TempZ-1 < RadT) then
                                EdgeSide = 1
                                CuspFound = .TRUE.
                                exit EdgeLoop
                            elseif (TempX+1 > GridSize-RadT .and. TempZ+1 > GridSize-RadT) then
                                EdgeSide = 4
                                CuspFound = .TRUE.
                                exit EdgeLoop
                            elseif (TempX+1 > GridSize-RadT .and. TempZ-1 < RadT) then
                                EdgeSide = 2
                                CuspFound = .TRUE.
                                exit EdgeLoop
                            elseif (TempX-1 < RadT .and. TempZ+1 > GridSize-RadT) then
                                EdgeSide = 3
                                CuspFound = .TRUE.
                                exit EdgeLoop
                            end if

                            EdgeLowVal = 0
                            EdgeLowA = 0
                            EdgeLowB = 0
                            InitialCusp = .FALSE.

                            if –TempX - 1 < RadT .or. TempX + 1 > GridSize-RadT) then
                                if (yVal(TempX,TempZ+1,1) >= CurrentY .and. yVal(TempX,TempZ-1,1) >= CurrentY) then
                                    if –TempX - 1 < RadT) then
                                        if (yVal(TempX+1,TempZ+1,1)>=CurrentY.and.yVal(TempX+1,TempZ-1,1)>=CurrentY.and.yVal(TempX+1,TempZ,1)>=CurrentY) then
                                            InitialCusp = .TRUE.
                                        end if
                                    elseif (TempX + 1 > GridSize-RadT) then
                                        if (yVal(TempX-1,TempZ+1,1)>=CurrentY.and.yVal(TempX-1,TempZ-1,1)>=CurrentY.and.yVal(TempX-1,TempZ,1)>=CurrentY) then
                                            InitialCusp = .TRUE.
                                        end if
                                    end if
                                end if

                                if (InitialCusp .eqv. .TRUE.) then
```

```
                                        if –TempX - 1 < RadT) then
                                                EdgeSide = 5
                                        elseif (TempX + 1 >
GridSize-RadT) then
                                                EdgeSide = 8
                                        end if
                                        CuspFound = .TRUE.
                                else
                                        if –TempX - 1 < RadT) then
                                                bSide1 = 0
                                                bSide2 = 1
                                        elseif (TempX + 1 >
GridSize-RadT) then
                                                bSide1 = -1
                                                bSide2 = 0
                                        end if
                                        do a = -1, 1
                                                do b = bSide1,
bSide2
                                                        if
(yVal(TempX+b,TempZ+a,1) < CurrentY) then
                                                                if
(yVal(TempX+b,TempZ+a,1) < EdgeLowVal .or. EdgeLowVal == 0) then
        EdgeLowVal = yVal(TempX+b,TempZ+a,1)
        EdgeLowA = a
        EdgeLowB = b
        end if
                                                        end if
                                                end do
                                        end do
                                end if

                                TempX = TempX + EdgeLowB
                                TempZ = TempZ + EdgeLowA

                        elseif –TempZ - 1 < RadT .or. TempZ + 1 >
GridSize-RadT) then
                                if (yVal(TempX+1,TempZ,1) >=
CurrentY .and. yVal(TempX-1,TempZ,1) >= CurrentY) then
                                        if –TempZ - 1 < RadT) then
                                                if
(yVal(TempX+1,TempZ+1,1)>=CurrentY.and.yVal(TempX-
1,TempZ+1,1)>=CurrentY.and.yVal(TempX,TempZ+1,1)>=CurrentY) then
                                                        InitialCusp
= .TRUE.
                                                end if
                                        elseif (TempZ + 1 >
GridSize-RadT) then
                                                if
(yVal(TempX+1,TempZ-1,1)>=CurrentY.and.yVal(TempX-1,TempZ-
1,1)>=CurrentY.and.yVal(TempX,TempZ-1,1)>=CurrentY) then
                                                        InitialCusp
= .TRUE.
```

```
end if
end if
end if

if (InitialCusp .eqv. .TRUE.) then
if –TempZ - 1 < RadT) then
EdgeSide = 6
elseif (TempZ + 1 >
GridSize-RadT) then

EdgeSide = 7
end if
CuspFound = .TRUE.
else
if –TempZ - 1 < RadT) then
bSide1 = 0
bSide2 = 1
elseif (TempZ + 1 >
GridSize-RadT) then

bSide1 = -1
bSide2 = 0
end if
do a = -1, 1
do b = bSide1,
bSide2

if
(yVal(TempX+a,TempZ+b,1) < CurrentY) then
if
(yVal(TempX+a,TempZ+b,1) < EdgeLowVal .or. EdgeLowVal == 0) then

EdgeLowVal = yVal(TempX+a,TempZ+b,1)

EdgeLowA = a

EdgeLowB = b

end if
end if
end do
end do
end if

TempX = TempX + EdgeLowA
TempZ = TempZ + EdgeLowB
end if

FinCount = FinCount + 1
if (FinCount > 10000) then
CuspFound = .TRUE.
SafeLocFound = .TRUE.
end if

end do EdgeLoop

SafeLocFound = .TRUE.

elseif (yVal(TempX+1,TempZ,1) > CurrentY .and.
yVal(TempX-1,TempZ,1) > CurrentY) then
```

```
if (yVal(TempX,TempZ+1,1) > CurrentY .and.
yVal(TempX,TempZ-1,1) > CurrentY) then
if (yVal(TempX+1,TempZ+1,1) > CurrentY
.and. yVal(TempX-1,TempZ-1,1) > CurrentY) then
if (yVal(TempX-1,TempZ+1,1) >
CurrentY .and. yVal(TempX+1,TempZ-1,1) > CurrentY) then
CuspHighCount = 1
CuspLowCount = 1

do a = -3, 3
do b = -3, 3
OuterLayer
= .FALSE.
if
(abs(a)+abs(b) >= 3) then

OuterLayer = .TRUE.
elseif
(abs(a)+abs(b) == 2) then
if
(a == 0 .or. b == 0) then

OuterLayer = .TRUE.

end if
end if
if
(OuterLayer .eqv. .TRUE.) then
if
(yVal(TempX+a,TempZ+b,1) >= CurrentY .or. yVal(TempX+a,TempZ+b,1) == 0) then

CuspHighCount = CuspHighCount + 1

elseif (TempX+a>GridSize-RadT.or.TempX+a<RadT.or.TempZ+b>GridSize-
RadT.or.TempZ+b<RadT) then

CuspHighCount = CuspHighCount + 1

elseif (yVal(TempX+a,TempZ+b,1) == 0) then

CuspHighCount = CuspHighCount + 1

elseif (yVal(TempX+a,TempZ+b,1) < CurrentY) then

if (TempX+a<=GridSize-RadT.and.TempX+a>=RadT.and.TempZ+b<=GridSize-
RadT.and.TempZ+b>=RadT) then

CuspHighList(CuspLowCount,1) = TempX+a

CuspHighList(CuspLowCount,2) = TempZ+b

CuspHighList(CuspLowCount,3) = yVal(TempX+a,TempZ+b,1)

CuspLowCount = CuspLowCount + 1

end if
```

```
        end if
                                        end if
                                    end do
                                end do

                                if (CuspHighCount == 41) then
                                    CuspFound = .TRUE.
                                    SafeLocFound = .TRUE.
                                else
                                    do a = 1, CuspLowCount-1
                                        Dist = ((CuspHighList(a,1)-OldX)**2)+((CuspHighList(a,2)-OldZ)**2)
                                        Dist = sqrt(Dist)

                                        if (FurthDist < Dist) then
                                            FurthDist = Dist
                                            CuspLowMoveTo = a
                                        end if
                                    end do

                                    OldX = TempX
                                    OldZ = TempZ
                                    TempX = CuspHighList(CuspLowMoveTo,1)
                                    TempZ = CuspHighList(CuspLowMoveTo,2)

                                    GO TO 20

                                end if
                            end if
                        end if
                    end if
                end if

                if (CuspFound .eqv. .TRUE.) then
                    SafeLocFound = .TRUE.
                else
                    LowYLoc = 0
                    LowYVal = 0

                    do a = -1, 1
                        do b = -1, 1
                            if (yVal(TempX+a,TempZ+b,1) <= CurrentY) then
                                if (yVal(TempX+a,TempZ+b,1) < LowYVal .or. LowYVal == 0) then
```

```
                                                LowYVal         =
yVal(TempX+a,TempZ+b,1)
                                                LowYLoc(1)      =
TempX+a
                                                LowYLoc(2)      =
TempZ+b
                                            end if
                                        end if
                                    end do
                                end do

                                if (OldX == LowYLoc(1) .and. OldZ == LowYLoc(2))
then
                                    TempX = OldX
                                    TempZ = OldZ
                                    CuspFound = .TRUE.
                                elseif (OldX == TempX .and. OldZ == TempZ) then
                                    CuspFound = .TRUE.
                                else
                                    OldX = TempX
                                    OldZ = TempZ
                                    TempX = LowYLoc(1)
                                    TempZ = LowYLoc(2)
                                end if
                            end if
20                          CONTINUE
                            FinCount = FinCount + 1
                            if (FinCount > 10000) then
                                CuspFound = .TRUE.
                                SafeLocFound = .TRUE.
                            end if
                        end do FindLoop

                        TempY = yVal(TempX,TempZ,1)

                        if (TempX == RadT .or. TempZ == RadT .or. TempX == GridSize-
RadT .or. TempZ == GridSize-RadT) then
                            EdgeCase = .TRUE.
                        end if

25                      CONTINUe

                        FPcount = 0
                        do a = 1, MolNo-1 ! Finds the particles closest to the low point for the
new particle to be resting on
                            if (MLyReal(a) < TempY+(RadT/2)) then
                                Dist = ((MLxReal(a)-TempX)**2)+((MLyReal(a)-
TempY)**2)+((MLzReal(a)-TempZ)**2)
                                Dist = sqrt–Dist) – RadT - MLr(a)
                                if (Dist <= FinalPartDist(1,1)) then
                                    FinalPartNo(5) = FinalPartNo(4)
                                    FinalPartDist(5,1) = FinalPartDist(4,1)
                                    FinalPartDist(5,2) = FinalPartDist(4,2)
                                    FinalPartNo(4) = FinalPartNo(3)
                                    FinalPartDist(4,1) = FinalPartDist(3,1)
                                    FinalPartDist(4,2) = FinalPartDist(3,2)
                                    FinalPartNo(3) = FinalPartNo(2)
```

```
                                FinalPartDist(3,1) = FinalPartDist(2,1)
                                FinalPartDist(3,2) = FinalPartDist(2,2)
                                FinalPartNo(2) = FinalPartNo(1)
                                FinalPartDist(2,1) = FinalPartDist(1,1)
                                FinalPartDist(2,2) = FinalPartDist(1,2)
                                FinalPartNo(1) = a
                                FinalPartDist(1,1) = Dist
                                FinalPartDist(1,2) = Dist + RadT + MLr(a)
                        elseif (Dist <= FinalPartDist(2,1)) then
                                FinalPartNo(5) = FinalPartNo(4)
                                FinalPartDist(5,1) = FinalPartDist(4,1)
                                FinalPartDist(5,2) = FinalPartDist(4,2)
                                FinalPartNo(4) = FinalPartNo(3)
                                FinalPartDist(4,1) = FinalPartDist(3,1)
                                FinalPartDist(4,2) = FinalPartDist(3,2)
                                FinalPartNo(3) = FinalPartNo(2)
                                FinalPartDist(3,1) = FinalPartDist(2,1)
                                FinalPartDist(3,2) = FinalPartDist(2,2)
                                FinalPartNo(2) = a
                                FinalPartDist(2,1) = Dist
                                FinalPartDist(2,2) = Dist + RadT + MLr(a)
                        elseif (Dist <= FinalPartDist(3,1)) then
                                FinalPartNo(5) = FinalPartNo(4)
                                FinalPartDist(5,1) = FinalPartDist(4,1)
                                FinalPartDist(5,2) = FinalPartDist(4,2)
                                FinalPartNo(4) = FinalPartNo(3)
                                FinalPartDist(4,1) = FinalPartDist(3,1)
                                FinalPartDist(4,2) = FinalPartDist(3,2)
                                FinalPartNo(3) = a
                                FinalPartDist(3,1) = Dist
                                FinalPartDist(3,2) = Dist + RadT + MLr(a)
                        elseif (Dist <= FinalPartDist(4,1)) then
                                FinalPartNo(5) = FinalPartNo(4)
                                FinalPartDist(5,1) = FinalPartDist(4,1)
                                FinalPartDist(5,2) = FinalPartDist(4,2)
                                FinalPartNo(4) = a
                                FinalPartDist(4,1) = Dist
                                FinalPartDist(4,2) = Dist + RadT + MLr(a)
                        elseif (Dist <= FinalPartDist(5,1)) then
                                FinalPartNo(5) = a
                                FinalPartDist(5,1) = Dist
                                FinalPartDist(5,2) = Dist + RadT + MLr(a)
                        end if
                end if
        end do

60      CONTINUE

        if
(FinalPartNo(1)>MolNo.or.FinalPartNo(2)>MolNo.or.FinalPartNo(3)>MolNo.or.FinalPartNo(4)
>MolNo.or.FinalPartNo(5)>MolNo) then
                FullCount = FullCount + 1
                GO TO 40
        end if

        PartCoords(1,1) = MLxReal(FinalPartNo(1))
        PartCoords(1,2) = MLyReal(FinalPartNo(1))
```

```
PartCoords(1,3) = MLzReal(FinalPartNo(1))

PartCoords(2,1) = MLxReal(FinalPartNo(2))
PartCoords(2,2) = MLyReal(FinalPartNo(2))
PartCoords(2,3) = MLzReal(FinalPartNo(2))

PartCoords(3,1) = MLxReal(FinalPartNo(3))
PartCoords(3,2) = MLyReal(FinalPartNo(3))
PartCoords(3,3) = MLzReal(FinalPartNo(3))

EquA1 = 2*MLxReal(FinalPart–o(1)) - 2*MLxReal(FinalPartNo(2))
EquA2 = 2*MLxReal(FinalPart–o(2)) - 2*MLxReal(FinalPartNo(3))
EquA3 = 2*MLxReal(FinalPart–o(3)) - 2*MLxReal(FinalPartNo(1))

EquB1 = 2*MLyReal(FinalPart–o(1)) - 2*MLyReal(FinalPartNo(2))
EquB2 = 2*MLyReal(FinalPart–o(2)) - 2*MLyReal(FinalPartNo(3))
EquB3 = 2*MLyReal(FinalPart–o(3)) - 2*MLyReal(FinalPartNo(1))

EquC1 = 2*MLzReal(FinalPart–o(1)) - 2*MLzReal(FinalPartNo(2))
EquC2 = 2*MLzReal(FinalPart–o(2)) - 2*MLzReal(FinalPartNo(3))
EquC3 = 2*MLzReal(FinalPart–o(3)) - 2*MLzReal(FinalPartNo(1))

Equd1 = (MLzReal(FinalPartNo(1–)**2) - (MLzReal(FinalPartNo(2))**2) + (MLyReal(FinalPartNo(1))**2)
EquD1 =–Equd1 - (MLyReal(FinalPartNo(2))**2) + (MLxReal(FinalPartNo(1–)**2) - (MLxReal(FinalPartNo(2))**2)
EquD1 =–EquD1 - ((MLr(FinalPartNo(1))+RadT)**2) + ((MLr(FinalPartNo(2))+RadT)**2)

Equd2 = (MLzReal(FinalPartNo(2–)**2) - (MLzReal(FinalPartNo(3))**2) + (MLyReal(FinalPartNo(2))**2)
EquD2 =–Equd2 - (MLyReal(FinalPartNo(3))**2) + (MLxReal(FinalPartNo(2–)**2) - (MLxReal(FinalPartNo(3))**2)
EquD2 =–EquD2 - ((MLr(FinalPartNo(2))+RadT)**2) + ((MLr(FinalPartNo(3))+RadT)**2)

Equd3 = (MLzReal(FinalPartNo(3–)**2) - (MLzReal(FinalPartNo(1))**2) + (MLyReal(FinalPartNo(3))**2)
EquD3 =–Equd3 - (MLyReal(FinalPartNo(1))**2) + (MLxReal(FinalPartNo(3–)**2) - (MLxReal(FinalPartNo(1))**2)
EquD3 =–EquD3 - ((MLr(FinalPartNo(3))+RadT)**2) + ((MLr(FinalPartNo(1))+RadT)**2)

if (CuspFound .eqv. .TRUE. .and. EdgeCase .eqv. .FALSE.) then ! If in a cusp and not on an edge

    EquValuesA(1,1) = EquA1
    EquValuesA(1,2) = EquB1
    EquValuesA(1,3) = EquC1
    EquValuesB(1) = EquD1

    EquValuesA(2,1) = EquA2
    EquValuesA(2,2) = EquB2
    EquValuesA(2,3) = EquC2
    EquValuesB(2) = EquD2

    EquValuesA(3,1) = EquA3
```

```
EquValuesA(3,2) = EquB3
EquValuesA(3,3) = EquC3
EquValuesB(3) = EquD3

stochDists(1) = FinalPartDist(1,2)
stochDists(2) = FinalPartDist(2,2)
stochDists(3) = FinalPartDist(3,2)

sumDist = stochDists(1) + stochDists(2) + stochDists(3) ! Starts stochastic optimisation to find the resting position

newCoords(1) = TempX
newCoords(2) = TempY
newCoords(3) = TempZ

dx(1) = 10 * RadLarge
dx(2) = 10 * RadLarge
dx(3) = 10 * RadLarge

do a = 1, 10
	do b = 1, 3
		dx(b) = dx(b) / 10
	end do
	do c = 1, 500
		do b = 1, 3
			call random_number(RX)
			stochxnew(b) = newCoords(b) + dx(b) * (2*RX-1)
		end do
		ibad = 0
		do b = 1, 3
			stochDists(b) = ((PartCoords(b,1)-stochxnew(1))**2)+((PartCoords(b,2)-stochxnew(2))**2)+((PartCoords(b,3)-stochxnew(3))**2)
			stochDists(b) = sqrt(stochDis–s(b)) – RadT - MLr(FinalPartNo(b))
			if (stochDists(b) < 0) then
				ibad = 1
			end if
		end do
		stochynew = stochDists(1) + stochDists(2) + stochDists(3)
		if (stochynew < sumDist .and. ibad == 0) then
			do b = 1, 3
				newCoords(b) = stochxnew(b)
			end do
			sumDist = stochynew
		end if
	end do
end do


NewX = newCoords(1)
NewY = newCoords(2)
NewZ = newCoords(3)
```

```
do a = 1, MolNo-1 ! Confirms that the particle is not overlapping with any others in its new position
    Dist = ((MLxReal(a)-NewX)**2)+((MLyReal(a)-NewY)**2)+((MLzReal(a)-NewZ)**2)
    Dist = sqrt(Dist)
    if (Dist < MLr(a)+RadT) then
        OverDist = 0
        OverlapCount = OverlapCount + 1
        if (OverlapCount > 2500) then
            GO TO 50
        end if
        do b = 1, 3
            Dist = ((MLxReal(a)-PartCoords(b,1))**2)+((MLyReal(a)-PartCoords(b,2))**2)+((MLzReal(a)-PartCoords(b,3))**2)
            Dist = sqrt(Dist)
            if (Dist < OverDist .or. OverDist == 0) then
                OverDist = Dist
                OverDistNo = b
            end if
        end do
        FinalPartNo(OverDistNo) = a
        FullCount = FullCount + 1
        if (OverDist == 0) then
            GO TO 50
        else
            GO TO 60
        end if
    end if
end do

DistAB = ((PartCoords(1,1)-PartCoords(2,1))**2)+((PartCoords(1,3)-PartCoords(2,3))**2)
DistAB = sqrt(DistAB)
DistBC = ((PartCoords(2,1)-PartCoords(3,1))**2)+((PartCoords(2,3)-PartCoords(3,3))**2)
DistBC = sqrt(DistBC)
DistCA = ((PartCoords(3,1)-PartCoords(1,1))**2)+((PartCoords(3,3)-PartCoords(1,3))**2)
DistCA = sqrt(DistCA)

DistAN = ((PartCoords(1,1)-NewX)**2)+((PartCoords(1,3)-NewZ)**2)
DistAN = sqrt(DistAN)
DistBN = ((PartCoords(2,1)-NewX)**2)+((PartCoords(2,3)-NewZ)**2)
DistBN = sqrt(DistBN)
DistCN = ((PartCoords(3,1)-NewX)**2)+((PartCoords(3,3)-NewZ)**2)
DistCN = sqrt(DistCN)

Sempi = (DistAB + DistBC + DistCA) / 2
SetTri = Sempi * –Sempi - DistAB) * –Sempi - DistBC) * –Sempi - DistCA)
SetTri = sqrt(SetTri)
```

```
                    SempiA = (DistAB + DistAN + DistBN) / 2
                    TriA = SempiA * (–empiA - DistAB) * (–empiA - DistAN) * (–
empiA - DistBN)

                    TriA = sqrt(TriA)

                    SempiB = (DistBC + DistBN + DistCN) / 2
                    TriB = SempiB * (–empiB - DistBC) * (–empiB - DistBN) * (–
empiB - DistCN)

                    TriB = sqrt(TriB)

                    SempiC = (DistCA + DistCN + DistAN) / 2
                    TriC = SempiC * (–empiC - DistCA) * (–empiC - DistCN) * (–
empiC - DistAN)

                    TriC = sqrt(TriC)

                    if (TriA + TriB + TriC > SetTri + 2.5) then ! Confirms the new
particle location is correctly resting on the three particles below it

50                       CONTINUE

                         OverlapCount = 0

                         if (FirstTri .eqv. .TRUE.) then
                              do a = 1, MolNo-1
                                   Dist = ((MLxReal(a)-
NewX)**2)+((MLzReal(a)-NewZ)**2)

                                   Dist = sqrt(Dist)
                                   if (Dist < RadT + MLr(a) +
(RadT*8) .and. NewY-MLyReal(a) < RadT + MLr(a) + (RadT*6)) then

     TriCheckNo(TriCheckCount) = a
                                        TriCheckCount =
TriCheckCount + 1

                                   end if
                              end do
                              aLoop: do a = TriCheckCount-1, 1, -1
                                   do b = TriCheckCount-1, 1, -1
                                        if (b < a) then
                                             do c =
TriCheckCount-1, 1, -1

                                                  if (c < b)
then

                                                       if
(TriCount <= 100000) then

     TriCombi(TriCount,1) = TriCheckNo(a)

     TriCombi(TriCount,2) = TriCheckNo(b)

     TriCombi(TriCount,3) = TriIckNo(c)

     TriCount = TriCount + 1

     else

     exit aLoop
```

```
          end if
                                                            end if
                                                        end do
                                                    end if
                                                end do
                                            end do aLoop
                                        end if

                                        FirstTri = .FALSE.

                                        if (TriAttempt <= TriCount-1) then
                                            FinalPartNo(1) = TriCombi(TriCount-
TriAttempt,1)
                                            FinalPartNo(2) = TriCombi(TriCount-
TriAttempt,2)
                                            FinalPartNo(3) = TriCombi(TriCount-
TriAttempt,3)
                                            TriAttempt = TriAttempt + 1
                                            if (TriAttempt <= 100000) then
                                                GO TO 60
                                            end if
                                        end if

                                        if (TempX >= GridSize-RadT-RadT .or. TempX <=
RadT+RadT .or. TempZ >= GridSize-RadT-RadT .or. TempZ <= RadT+RadT) then
                                            if (TempX <= RadT+RadT .and. TempZ <=
RadT+RadT) then
                                                EdgeSide = 1
                                            elseif (TempX >= GridSize-RadT-RadT
.and. TempZ >= GridSize-RadT-RadT) then
                                                EdgeSide = 4
                                            elseif (TempX >= GridSize-RadT-RadT
.and. TempZ <= RadT+RadT) then
                                                EdgeSide = 2
                                            elseif (TempX <= RadT+RadT .and. TempZ
>= GridSize-RadT-RadT) then
                                                EdgeSide = 3
                                            elseif (TempX <= RadT+RadT) then
                                                EdgeSide = 5
                                            elseif (TempX >= GridSize-RadT-RadT)
then
                                                EdgeSide = 8
                                            elseif (TempZ <= RadT+RadT) then
                                                EdgeSide = 6
                                            elseif (TempZ >= GridSize-RadT-RadT)
then
                                                EdgeSide = 7
                                            end if
                                            EdgeCase = .TRUE.
                                            GO TO 25
                                        end if
                                        FullCount = FullCount + 1
                                        GO TO 40
                                    end if

                                    do a = 1, 3
```

```
                    FinalDists(a) = ((PartCoords(a,1)-NewX)**2)+((PartCoords(a,2)-NewY)**2)+((PartCoords(a,3)-NewZ)**2)
                    FinalDists(a) = sqrt(FinalDists(a))
                    if (FinalDists(a) > RadT+MLr(FinalPartNo(a))+1) then
                        FullCount = FullCount + 1
                        GO TO 50
                    end if
                end do

                if (NewX < RadT .or. NewY < RadT .or. NewZ < RadT) then
                    FullCount = FullCount + 1
                    if (TempX >= GridSize-RadT-RadT .or. TempX <= RadT+RadT .or. TempZ >= GridSize-RadT-RadT .or. TempZ <= RadT+RadT) then
                        if (TempX <= RadT+RadT .and. TempZ <= RadT+RadT) then
                            EdgeSide = 1
                        elseif (TempX >= GridSize-RadT-RadT .and. TempZ >= GridSize-RadT-RadT) then
                            EdgeSide = 4
                        elseif (TempX >= GridSize-RadT-RadT .and. TempZ <= RadT+RadT) then
                            EdgeSide = 2
                        elseif (TempX <= RadT+RadT .and. TempZ >= GridSize-RadT-RadT) then
                            EdgeSide = 3
                        elseif (TempX <= RadT+RadT) then
                            EdgeSide = 5
                        elseif (TempX >= GridSize-RadT-RadT) then
                            EdgeSide = 8
                        elseif (TempZ <= RadT+RadT) then
                            EdgeSide = 6
                        elseif (TempZ >= GridSize-RadT-RadT) then
                            EdgeSide = 7
                        end if
                        EdgeCase = .TRUE.
                        GO TO 25
                    end if
                    GO TO 50
                elseif (NewX > GridSize-RadT .or. NewY > GridSize-RadT .or. NewZ > GridSize-RadT) then
                    FullCount = FullCount + 1
                    if (TempX >= GridSize-RadT-RadT .or. TempX <= RadT+RadT .or. TempZ >= GridSize-RadT-RadT .or. TempZ <= RadT+RadT) then
                        if (TempX <= RadT+RadT .and. TempZ <= RadT+RadT) then
                            EdgeSide = 1
                        elseif (TempX >= GridSize-RadT-RadT .and. TempZ >= GridSize-RadT-RadT) then
                            EdgeSide = 4
                        elseif (TempX >= GridSize-RadT-RadT .and. TempZ <= RadT+RadT) then
                            EdgeSide = 2
                        elseif (TempX <= RadT+RadT .and. TempZ >= GridSize-RadT-RadT) then
```

```
                                                EdgeSide = 3
                                        elseif (TempX <= RadT+RadT) then
                                                EdgeSide = 5
                                        elseif (TempX >= GridSize-RadT-RadT)
then
                                                EdgeSide = 8
                                        elseif (TempZ <= RadT+RadT) then
                                                EdgeSide = 6
                                        elseif (TempZ >= GridSize-RadT-RadT)
then
                                                EdgeSide = 7
                                        end if
                                        EdgeCase = .TRUE.
                                        GO TO 25
                                end if
                                GO TO 50
                        end if

                        MLxReal(MolNo) = newX
                        MLyReal(MolNo) = newY
                        MLzReal(MolNo) = NewZ

                elseif (EdgeCase .eqv. .TRUE.) then ! Else if the particle has come to
rest upon an edge

                        if (EdgeSide < 5) then ! If the particle is resting in a corner, so
needs a single particle contact
                                if (EdgeSide == 1) then
                                        NewX = RadT
                                        NewZ = RadT
                                elseif  (EdgeSide == 2) then
                                        NewX = GridSize-RadT
                                        NewZ = RadT
                                elseif  (EdgeSide == 3) then
                                        NewX = RadT
                                        NewZ = GridSize-RadT
                                elseif  (EdgeSide == 4) then
                                        NewX = GridSize-RadT
                                        NewZ = GridSize-RadT
                                end if

                                Dist = RadT + MLr(FinalPartNo(1))

                                newY                    =                       -
(MLxReal(FinalPartNo(1))**2)+(2*MLxReal(FinalPartNo(1))*NewX)-
(MLzReal(FinalPartNo(1))**2)
                                NewY                                            =
NewY+(2*MLzReal(FinalPartNo(1))*NewZ)+(Dist**2)-(NewX**2)-(NewZ**2)
                                NewY = sqrt(NewY)

                                if (NewY /= NewY) then
                                        if (FirstEdge .eqv. .TRUE.) then
                                                do a = 1, MolNo-1
                                                        dist    =       ((MLxReal(a)-
Newx)**2)+((MLzReal(a)-NewZ)**2)
                                                        Dist = sqrt(Dist)
```

```
                                        if (Dist < RadT  +  MLr(a)  +
(RadT*8)) then
        EdgeCombi(EdgeCount) = a
                                                EdgeCount       =
EdgeCount + 1
                                        end if
                                end do
                        end if

                        FirstEdge = .FALSE.

                        if (EdgeAttempt <= EdgeCount-1) then
                                FinalPartNo(1)                  =
EdgeCombi(EdgeCount-EdgeAttempt)
                                EdgeAttempt = EdgeAttempt + 1
                                GO TO 60
                        end if

                        FullCount = FullCount + 1
                        GO TO 40
                end if

                –f  (tempY  -  (MLyReal(FinalPartNo(1))  +  NewY–  <
tempY - (MLyReal(FinalP–rtNo(1)) - NewY)) then
                        NewY = MLyReal(FinalPartNo(1)) + NewY
                else
                        NewY = MLyReal(FinalP–rtNo(1)) - NewY
                end if

                do a = 1, MolNo-1
                        dist            =               ((MLxReal(a)-
Newx)**2)+((MLyReal(a)-Newy)**2)+((MLzReal(a)-NewZ)**2)
                        Dist = sqrt(Dist)
                        if (Dist < MLr(a)+RadT-1) then
                                OverlapCount = OverlapCount + 1
                                if (OverlapCount > 2500) then
                                        GO TO 40
                                end if
                                FinalPartNo(1) = a
                                FullCount = FullCount + 1
                                GO TO 60
                        end if
                end do

        else ! On a regular edge so resting on two particles
                NewXVal = .FALSE.
                NewZVal = .FALSE.

                if (EdgeSide == 5) then
                        stochxnew(1) = RadT
                        newCoords(3) = TempZ
                        NewXVal = .TRUE.
                elseif (EdgeSide == 6) then
                        newCoords(1) = TempX
                        stochxnew(3) = RadT
                        NewZVal = .TRUE.
```

```
elseif (EdgeSide == 7) then
    newCoords(1) = TempX
    stochxnew(3) =–GridSize - RadT
    NewZVal = .TRUE.
elseif (EdgeSide == 8) then
    stochxnew(1) =–GridSize - RadT
    newCoords(3) = TempZ
    NewXVal = .TRUE.
end if

stochDists(1) = FinalPartDist(1,2)
stochDists(2) = FinalPartDist(2,2)

sumDist = stochDists(1) + stochDists(2)

newCoords(2) = TempY

dx(1) = 10 * RadLarge
dx(2) = 10 * RadLarge
dx(3) = 10 * RadLarge

do a = 1, 10
    do b = 1, 3
        dx(b) = dx(b) / 10
    end do
    do c = 1, 500
        call random_number(RX)
        stochxnew(2) = newCoords(2) + dx(2) * (2*RX-1)

        if (NewXVal .eqv. .TRUE.) then
            call random_number(RX)
            stochxnew(3) = newCoords(3) + dx(3) * (2*RX-1)
        elseif (NewZVal .eqv. .TRUE.) then
            call random_number(RX)
            stochxnew(1) = newCoords(1) + dx(1) * (2*RX-1)
        end if

        ibad = 0

        do b = 1, 2
            stochDists(b) = ((PartCoords(b,1)-stochxnew(1))**2)+((PartCoords(b,2)-stochxnew(2))**2)+((PartCoords(b,3)-stochxnew(3))**2)

            stochDists(b) = sqrt(stoch–ists(b–) - RadT - MLr(FinalPartNo(b))

            if (stochDists(b) < 0) then
                ibad = 1
            end if
        end do
        stochynew = stochDists(1) + stochDists(2)

        if (stochynew < sumDist .and. ibad == 0) then

            do b = 1, 3
```

```
                newCoords(b) = stochxnew(b)
            end do
            sumDist = stochynew
        end if
    end do
end do

NewX = newCoords(1)
NewY = newCoords(2)
NewZ = newCoords(3)

do a = 1, MolNo-1
    dist = ((MLxReal(a)-Newx)**2)+((MLyReal(a)-Newy)**2)+((MLzReal(a)-NewZ)**2)
    Dist = sqrt(Dist)
    if (Dist < MLr(a)+RadT-1) then
        OverDist = 0
        OverlapCount = OverlapCount + 1
        if (OverlapCount > 2500) then
            GO TO 30
        end if
        do b = 1, 2
            dist = ((MLxReal(a)-PartCoords(b,1))**2)+((MLyReal(a)-PartCoords(b,2))**2)+((MLzReal(a)-PartCoords(b,3))**2)
            Dist = sqrt(Dist)
            if (Dist < OverDist .or. OverDist == 0) then
                OverDist = Dist
                OverDistNo = b
            end if
        end do
        FinalPartNo(OverDistNo) = a
        FullCount = FullCount + 1
        if (OverDist == 0) then
            GO TO 30
        else
            GO TO 60
        end if
    end if
end do

do a = 1, 2
    FinalDists(a) = ((PartCoords(a,1)-NewX)**2)+((PartCoords(a,2)-NewY)**2)+((PartCoords(a,3)-NewZ)**2)
    FinalDists(a) = sqrt(FinalDists(a))
    if (FinalDists(a) > RadT+MLr(FinalPartNo(a))+1) then
        FullCount = FullCount + 1

30      CONTINUE

        if (FirstSide .eqv. .TRUE.) then
            do b = 1, MolNo-1
                dist = ((MLxReal(b)-Newx)**2)+((MLzReal(b)-NewZ)**2)

                Dist = sqrt(Dist)
```

```
                              if (Dist < RadT   +
MLr(b) + (RadT*8)) then
          SideCheckNo(SideCheckCount) = b
          SideCheckCount = SideCheckCount + 1
                              end if
                         end do

                         bLoop:   do   b   =
SideCheckCount-1, 1, -1
                              do   c   =
SideCheckCount-1, 1, -1
                                   if  (c  <  b)
then
                                        if
(SideCount <= 100000) then
          SideCombi(SideCount,1) = SideCheckNo(b)
          SideCombi(SideCount,2) = SICheckNo(c)
          SideCount = SideCount + 1
          else
          exit bLoop
          end if
                                   end if
                              end do
                         end do bLoop
                    end if

                    SideAttempt = SideAttempt + 1
                    FirstSide = .FALSE.

                    do b = SideAttempt, SideCount-1
                         FinalPartNo(1)        =
SideCombi(b,1)
                         FinalPartNo(2)        =
SideCombi(b,2)
                         GO TO 60
                    end do
                    FullCount = FullCount + 1
                    GO TO 40
               end if
          end do

          if (NewX < RadT .or. NewY < RadT .or. NewZ <
RadT) then
               FullCount = FullCount + 1
               GO TO 30
          elseif (NewX > GridSize-RadT .or. NewY > GridSize-
RadT .or. NewZ > GridSize-RadT) then
               FullCount = FullCount + 1
               GO TO 30
```

```
                                        end if
                                end if

                                if (NewX < RadT .or. NewY < RadT .or. NewZ < RadT) then
                                        FullCount = FullCount + 1
                                        GO TO 40
                                elseif (NewX > GridSize-RadT .or. NewY > GridSize-RadT
.or. NewZ > GridSize-RadT) then
                                        FullCount = FullCount + 1
                                        GO TO 40
                                end if

                                MLxReal(MolNo) = NewX
                                MLyReal(MolNo) = NewY
                                MLzReal(MolNo) = NewZ

                        else
                                MLxReal(MolNo) = TempX          ! Saves the particle location
                                MLyReal(MolNo) = TempY
                                MLzReal(MolNo) = TempZ
                        end if

                        MLr(MolNo) = RadT
                        FullCount = 0
                        OverlapCount = 0
                        Full = 0
                        MolNo = MolNo + 1

                else
                        MLxReal(MolNo) = x
                        MLyReal(MolNo) = y
                        MLzReal(MolNo) = z
                        MLr(MolNo) = RadT
                        FullCount = 0
                        OverlapCount = 0
                        Full = 0
                        MolNo = MolNo + 1
                end if

10              CONTINUE

        else
                Full = 1
90              CONTINUE
        end if

        end


        subroutine PointSafe ! Determines if the falling particle has impacted yet
        use allSubs ! Loads the variables from the module
        integer a, b, c
        real Dist
        character t

        Hit = .FALSE.
        FullCheck = .FALSE.
```

```
        ! Checks the distance between the current falling particle location and previously
placed partice to determine if it has impacted

        cloop: do – = 1, MolNo - 1
                Dist = ((MLxReal(a)-x)**2)+((MLyReal(a)-y)**2)+((MLzReal(a)-z)**2)
                Dist = sqrt(Dist)
                if (Dist <= ((RadT)+(Mlr(a)))) then
                        Hit = .TRUE.
                        exit cloop
                end if
        end do cloop

        ! If the impact is above the top of the box, a counter is incremented to show the box
may be full
        if ((Hit .eqv. .TRUE.) .and. (y –= (GridSize - RadT))) then
                Hit = .FALSE.
                FullCount = FullCount + 1
                RoofCount = RoofCount + 1
        end if

        if (RoofCount >= 500) then
                FullCheck = .TRUE.
        end if

        end
```

Appendix 4: 2D Chain System Algorithm

This appendix contains the algorithm used to create the 2D chain systems.

```
	module allSubs ! Initialises the variables used through all functions
	character, dimension (:,:), allocatable, public :: RA*4
	integer, dimension (:,:), allocatable, public :: RAMolClose
	integer MolNo, RadLarge, RadSmall, BoxSize, GridSize, Rads, count, SN, Full,
FullCount, OneLegacyCount
	integer MLx, MLy, MLr, Quad, QuadC, AllocateVal, Roofcount
	real MLxReal, MLyReal
	integer, dimension (:), allocatable, public :: OneLegacyCounterCount
	integer, dimension (:,:), allocatable, public :: Ones
	integer, dimension (:,:), allocatable, public :: ChainOnes
	integer, dimension (:,:,:), allocatable, public :: OnesLegacy
	integer, dimension (:,:), allocatable, public :: Contacts
	dimension MLr(10000), Rads(10), MLxReal(10000), MLyReal(10000)
	logical FullCheck, Hit, RoofHit, StartPlace
	integer x, y, Long, Tall, RadT, ChainLength, OverFallCount, MinFallCount, LoopNo
	end module allSubs

	program packedbed
	use allSubs ! Loads the variables from the module

	! Initialises local variables
	character t, FileName*15, FileID*3
	integer m, n, check, PrintNo, ProgCount, PCId, iSeed, count3
	dimension FileID(1000)
	real RX, ScaleFac, ScaleVal
	real Dist
	real PartArea, VoidArea, VoidFrac, Pi
	integer count2, RadTnew, TotLength
	logical Finished, Cont, Impact
	dimension iSeed(50)

!	Variables initial values set
	StartPlace = .TRUE.
	OverFallCount = 0
	MinFallCount = 0
	LoopNo = ‘
’	if (t == 'y') then
		Rads(1) = 10	! Sets the radius of a particle. Additional radii would be
inputted ‘s Rads’x) = 'Radius'
		SN = 1		! Sets the number of different radii in the system

		RadLarge = 0
		RadSmall = 0
		ChainLength = 5 ! Sets the number of particles per chain

		do count = 1, SN
			if (RadLarge < Rads(count)) then
				RadLarge = Rads(count)
			end if
			if (RadSmall > Rads(count) .or. RadSmall == 0) then
```

```
                    RadSmall = Rads(count)
               end if
          end do

          RadLarge = 10
          RadSmall = 10

          ! Calculates the box size based on the largest radius present
          BoxSize = (RadLarge*6) * (ChainLength/2)
          GridSize = BoxSize*5
          AllocateVal = ((BoxSize*3)**2)*2

          ! Allocates the arrays
          allocate(RA(1:GridSize, 1:GridSize))
          allocate(RAMolClose(1:GridSize, 1:GridSize))

          allocate(Ones(1:AllocateVal,1:2))
          allocate(ChainOnes(1:AllocateVal,1:2))
          allocate(OnesLegacy(1:270,1:AllocateVal,1:2))
          allocate(OneLegacyCounterCount(1:270))

          do ProgCount = 1, 50
               write(FileID‘Prog’ount), '(i0)') ProgCount
          end do

          do ProgCount = 1, 50 ! Starts the loop for the number of systems to be created

               ! Se219nitialiables intial values
               MLxreal = 0
               MLyReal = 0
               MLr = 0

               PartArea = 0
               VoidArea = 0
               VoidFrac = 0
               Pi = 3.141596535
               TotLength = 0
               MolNo = 1
               OneLegacyCount = 1
               OverFallCount = 0
     ‘’FileName = ''
               Full = 0
               check = 0
               FullCount = 0
               Finished = .FALSE.

               Ones = 0
               OnesLegacy = 0
               OneLegacyCounterCount ‘ ’

               RA = '0'
               RAMolClose = 0

               RoofCount = 0
               RoofHit = .FALSE.

               call random_seed()
```

```
LayerLoop: do while (count < 10000000)

    Impact = .FALSE.

    ! Picks a random radius and x coordinate, and sets y to be on
the bottom of the box
    call random_number(RX)
    count2 = 1 + floor(SN*RX)
    RadT = Rads(count2)

    call random_number(RX)
    count2 = 1 + floor((GridSize-(2*RadT))*RX)
    x = count2+RadT

    y = RadT

    if (MolNo > 1) then        ! Checks there is already at least
one particle in the system
        hitloop: do count– = 1, MolNo - 1
            Dist = ((MLxReal(count3)-
x)**2)+((MLyReal(count3)-y)**2)

            Dist = sqrt(Dist)
            if (Dist <= ((RadT*2)+(Mlr(count3)))) then
                count = count + 1
                Impact = .TRUE.
                exit hitloop
            end if
        end do hitloop
        if ((Impact .eqv. .FALSE.) .and. (x <= GridSize-
(RadT*ChainLength))) then ! If the particle is not overlapping with any others and is inside the
grid, its location is saved
            MLxReal(MolNo) = x
            MLyReal(MolNo) = y
            MLr(MolNo) = RadT
            MolNo = MolNo + 1

            do m = 1, ChainLength - 1
                x = x + RadT

                if (x < GridSize-RadT) then
                    Impact = .FALSE.
                    hitloop3: do count3 = 1,
MolNo - (m+1)
                        Dist =
((MLxReal(count3)-x)**2)+((MLyReal(count3)-y)**2)

                        Dist = sqrt(Dist)
                        if (Dist <=
((RadT*2)+(Mlr(count3)))) then

                            count =
count + 1

                            Impact =
.TRUE.

                            exit
hitloop3

                        end if
                    end do hitloop3
```

```
                                                if (Impact .eqv. .FALSE.) then
                                                        MLxReal(MolNo) = x
                                                        MLyReal(MolNo) = y
                                                        MLr(MolNo) = RadT
                                                        MolNo = MolNo + 1
                                                else
                                                        MolNo = MolNo - (m)
                                                        Cycle LayerLoop
                                                end if
                                        else
                                                MolNo = MolNo - (m)
                                                Cycle LayerLoop
                                        end if
                                end do

                                count = 0
                        end if
                else
                        MLxReal(MolNo) = x
                        MLyReal(MolNo) = y
                        MLr(MolNo) = RadT
                        MolNo = MolNo + 1

                        do m = 1, ChainLength - 1
                                x = x + RadT

                                if (x < GridSize - RadT) then
                                        MLxReal(MolNo) = x
                                        MLyReal(MolNo) = y
                                        MLr(MolNo) = RadT
                                        MolNo = MolNo + 1
                                else
                                        MolNo = MolNo - (m)
                                        Cycle LayerLoop
                                end if
                        end do
                end if
        end do LayerLoop

        ! This loops through the base line to check that there is nowhere a small particle could fall through to the bottom of the box, and if so, places a particle there
        do m = RadSmall, GridSize-RadSmall
                RadT = RadSmall
                Impact = .FALSE.
                hitloop2: do count3 = 1, MolNo - 1
                        Dist = ((MLxReal(count3)-m)**2)
                        Dist = sqrt(Dist)
                        if (Dist < ((RadSmall)+(Mlr(count3)))) then
                                count = count + 1
                                Impact = .TRUE.
                                exit hitloop2
                        end if
```

```
					end do hitloop2
					if (Impact .eqv. .FALSE.) then
						count = 0
						MLxReal(MolNo) = m
						MLyReal(MolNo) = RadSmall
						MLr(MolNo) = RadSmall
						MolNo = MolNo + 1

						call AddChain
					end if
				end do

				StartPlace = .FALSE.

				do n = 1, 500	! Loops for each particle being added to the system, using the main function. At the end of each loop, it checks if the box is full and if so, leaves the loop.
					LoopNo = n
					call molpos
					if (Full == 1) then
						exit
					elseif (RoofHit .eqv. .TRUE.) then
						exit
					end if
				end do

				t = 'y'
				if (t == 'y' .and. RoofHit .eqv. .TRUE.) then ! Saves the particle locations to a file
					FileName = ''
					FileName = trim(adjustl(FileID(ProgCount))) // '.csv'
					open(1, file = FileName, status = 'new')
					do y = 1, MolNo-1
						write(1,*) MLxReal(y), ',' , MLyReal(y), ',' , MLr(y)
					end do
					close(1)
				end if
			end do
		end if

		end program


		subroutine molpos
		use allSubs ! Loads the variables from the module

		! Sets up the local variables
		integer count2, Spot, Height, DoubRad, RowRad, a,b,c, RadIn, m, n, TempX, TempY
		real MidWay, Dist
		character t, FileName*15
		integer SavIncremX, SavIncremY, SafeLocCount, LR, SavOneX, SavOneY, SavOnePart, LRNo, OneCount, RealPos1, RealPos2
		integer TempRealPos1, TempRealPos2, ChainOneCount
		logical SafeLocFound, Go, ResetCheck
		real TempXa, TempXb, TempYa, TempYb, DistAB, DistBC, DistAC, AngleA, AngleB, AngleFin, GradFin, HelpDist
		real FDistA, FDistB, FDistC, FDistD
```

```
      real xDiff, yDiff, Pi, FinalSavX, FinalSavY, SavDist
      integer FinalSavLong, FinalSavTall, checktime, Balanced
      integer Balances, Touches
      dimension Balances(10)
      real DistFac, RadScale
      integer NewPos, TRP1Swap, TRP2Swap
      integer FinalPart, SideCount
      dimension FinalPart(2,3)
      real PartCoords, stochDists, sumDist, dx, stochxnew, newCoords, stochynew
      integer ibad
      dimension PartCoords(2,2), stochDists(2), newCoords(2), dx(2), stochxnew(2)
      real RX, NewX, NewY
      logical ChainAdd
      real OverDist
      integer OverDistNo
      logical NotBal, FirstBal
      dimension BalCheckNo(10000), BalCombi(100000,2)
      integer BalCheckCount, BalCount, BalCheckNo, BalCombi, BalAttempt
      integer OverlapCount
      logical EdgeCase, FirstEdge
      integer EdgeCombi, EdgeCount, EdgeAttempt
      dimension EdgeCombi(10000)
      integer ChainStartNo, ChainStartCont, d
      real ChainGrad, EquC
      logical ChainHitCheck, ChainEdgeHitCheck, ChainTopHitCheck
      logical TopCase
      integer OverTopCount, PartHit, MiddlePart, MiddlePartA, MiddlePartB, FallPoint,
ContPoint, ContPointB
      real Dy, intC, CheckY, MiddlePoint, ContDist, ContSpot
      logical Fell, Tilt, HMAdju
      integer FallCount, AdjCount
      real ContSpotTemp, ContSpotTempUy, ContSpotL, ContSpotR, ContSpotUy,
ContSpotUx
      integer ContSpotUn, NLCount
      logical RoundTwo
      real Valdx, ValDist, ValAng1,ValDist2, ValDist3, ValAng2a, ValAng2b, ValAng2,
ValAng3
      logical SideBal, Upwards, Downwards


      Fell = .FALSE.
      Tilt = .FALSE.
      FallPoint = 0
      FallCount = 0
      AdjCount = 0

40    CONTINUE

      if (FullCount == 2500000) then
            Full = 1
      end if
      if (FallCount > 20) then
            OverFallCount = OverFallCount + 1
            if (OverFallCount >= 100) then
                  Full = 1
            end if
            GO TO 10
```

```
end if

! Setting initial values of variables
Hit = .FALSE.
ResetCheck = .FALSE.
TopCase = .FALSE.
ChainTopHitCheck = .FALSE.
checktime = 1
Balanced = 0
Touches = 0
Balances = 0

NLCount = 0

HMAdju = .FALSE.
ContSpotTemp = 0
ContSpotTempUy = 0
ContSpotL = 0
ContSpotR = 0
ContSpotUy = 0
ContSpotUx = 0
ContSpotUn = 0

SideBal = .FALSE.
Upwards = .FALSE.
Downwards = .FALSE.

RoundTwo = .FALSE.

MiddlePart = 0
MiddlePartA = 0
MiddlePartB = 0
MiddlePoint = 0
ContPoint = 0
ContPointB = 0
ContSpot = 0
ContDist = 99999

Valdx = 0
ValDist = 0
ValAng1 = 0
ValDist2 = 0
ValDist3 = 0
ValAng2a = 0
ValAng2b = 0
ValAng2 = 0
ValAng3 = 0

Dy = 0
intC = 0
CheckY = 0

ChainStartNo = 0
ChainStartCont = 1
ChainGrad = 0
EquC = 0
```

```
EdgeAttempt = 0
EdgeCount = 0
EdgeCombi = 0
EdgeCase = .FALSE.
FirstEdge = .TRUE.

FinalPart = 999999
OverDist = 0
OverDistNo = 0

OverlapCount = 0
OverTopCount = 0

TRP1Swap = 0
TRP2Swap = 0

ChainAdd = .FALSE.

NotBal = .FALSE.
FirstBal = .TRUE.
BalCheckNo = 0
BalCombi = 0
BalCheckCount = 0
BalCount = 0
BalCheckNo = 0
BalCombi = 0
BalAttempt = 0

RadScale = 0
TempRealPos1 = 0
TempRealPos2 = 0
DistFac = 0
FDistA = 0
FDistB = 0
FDistC = 0
FDistD = 0
SavDist = 0
SavOneX = 0
SavOneY = 0
TempXa = 0
TempXb = 0
TempYa = 0
TempYb = 0
DistAB = 0
DistBC = 0
DistAC = 0
AngleA = 0
AngleB = 0
AngleFin = 0
GradFin = 0
xDiff = 0
yDiff = 0
Pi = 3.141596535
NewPos = 0

FinalDists = 0
```

```
        PartCoords = 0
        stochDists = 0
        sumDist = 0
        dx = 0
        stochxnew = 0
        newCoords = 0
        stochynew = 0

        if (Fell .eqv. .TRUE.) then
                Fell = .FALSE.
                GO TO 66
        end if

        ! Randomly chooses which radius will be used for this particle
        call random_number(RX)
        RadScale = RadLarge/RadSmall
        RadScale = RadScale + 1
        count2 = 1 + floor(2*RX)
        if (count2 == 2) then
                RadT = RadLarge
        else
                RadT = RadSmall
        end if

        wloop: do while (Hit .eqv. .FALSE.)
                FallPoint = GridSize

                ! Randomly chooses the x value
                call random_number(RX)
                Spot = 1 + floor((GridSize-(2*RadT))*RX)
                x = Spot+RadT

66              CONTINUE

                do y = FallPoint, RadT, -1 ! Loops from the top of the box, and sends to the
function to determine impact
                        Long = (x/BoxSize)+1
                        Tall = (y/BoxSize)+1

                        call PointSafe

                        ! If the box is full or an impact has occured, the loop is exited

                        if (Full == 1) then
                                exit wloop
                        end if
                        if (FullCheck .eqv. .TRUE.) then
                                RoofHit = .TRUE.
                                GO TO 10
                        end if

                        if (Hit .eqv. .TRUE.) then
                                exit wloop
                        end if
                end do
        end do wloop
```

```
if ((Hit .eqv. .TRUE.) .and. (Full /= 1)) then
        if (MolNo > 1) then
                RA = '0'
                RAMolClose = 0

                ! Sets up the variables to be used for particle placement

                Ones = 0
                OneCount = 1
                ChainOnes = 0
                ChainOneCount = 1

                do a = 1, MolNo - 1        ! Loops through the particles for contour plot
placement
                        MLxCor = MLxReal(a)
                        MLyCor = MLyReal(a) ! Takes the radius, x and y coordinates
of the current particle in the loop
                        RadIn = Mlr(a)

                        DoubRad = (RadIn+RadT)+1
                        do Height = 0, RadIn      ! Draws the particle onto the contour
plot, "-"s marking blocked locations, "1"s being valid spots
                                MidWay = RadIn**2 - Height**2
                                RowRad = abs(sqrt(MidWay))
                                if((MLxCor+Height<=GridSize-
RadT).and.(MLyCor+RowRad<=GridSize-RadT).and.(MLyCor-
RowRad>=RadT).and.(MLxCor-Height>=RadT))then
                                        RA(MLyCor+RowRad, MLxCor+Height) = '-'
                                        RA(MLyCor-RowRad, MLxCor+Height) = '-'
                                        RA(MLyCor+RowRad, MLxCor-Height) = '-'
                                        RA(MLyCor-RowRad, MLxCor-Height) = '-'
                                end if
                                do count2 = -RowRad,RowRad
                                        if((MLxCor+Height<=GridSize-
RadT).and.(MLyCor+count2<=GridSize).and.(MLxCor-
Height>=RadT).and.(MLyCor+count2>=RadT))then
                                                RA(MLyCor+count2,
MLxCor+Height) = '-'
                                                RA(MLyCor+count2,          MLxCor-
Height) = '-'
                                        end if
                                end do
                        end do

                        do Height = 0, DoubRad ! Draws locations around the current
particle that are too close for the new particle to be added due to overlap
                                MidWay = DoubRad**2 - Height**2
                                RowRad = abs(sqrt(Midway))
                                do count2 = -rowrad+1, rowrad-1
                                        if
((MLyCor+count2<=GridSize).and.(MLxCor+Height<=GridSize-
RadT).and.(MLyCor+count2>=RadT).and.(MLxCor+Height>=RadT))then
                                                RA(MLyCor+count2,
MLxCor+Height) = '-'
                                        end if
```

```
                                        if
((MLyCor+count2<=GridSize).and.(MLxCor-Height<=GridSize-
RadT).and.(MLyCor+count2>=RadT).and.(MLxCor-Height>=RadT))then
                                                RA(MLyCor+count2,          MLxCor-
Height) = '-'
                                        end if
                                end do
                        end do

                        do Height = -RadIn, RadIn
                                do count2 = MlyCor-1, 1, -1
                                        if
((count2<=GridSize).and.(MLxCor+Height<=GridSize-
RadT).and.(count2>=RadT).and.(MLxCor+Height>=RadT))then
                                                RA(count2, MLxCor+Height) = '-'
                                        end if
                                end do
                        end do

                        do Height = -DoubRad, 0            ! Adds the valid spots for the
resting particle to be placed
                                MidWay = DoubRad**2 - Height**2
                                RowRad = abs(sqrt(Midway))

                                if
((MLyCor+RowRad<=GridSize).and.(MLxCor+height<=GridSize-
RadT).and.(MLxCor+height>=RadT).and.(MLyCor+RowRad>=RadT))then
                                        if  (RA(MLyCor+RowRad,  MLxCor+Height)
/= '-') then
                                                RA(MLyCor+RowRad,
MLxCor+Height) = '1'
                                                RAMolClose(MLyCor+RowRad,
MLxCor+Height) = a
                                        end if
                                end if
                                if                                      ((MLyCor-
RowRad>=RadT).and.(MLxCor+height<=GridSize-
RadT).and.(MLxCor+height>=RadT).and.(MLyCor-RowRad<=GridSize)) then
                                        if (RA(MLyCor-RowRad, MLxCor+Height) /=
'-') then
                                                RA(MLyCor-RowRad,
MLxCor+Height) = '1'
                                                RAMolClose(MLyCor-RowRad,
MLxCor+Height) = a
                                        end if
                                end if
                                if      ((MLyCor+RowRad<=GridSize).and.(MLxCor-
Height<=GridSize-RadT).and.(MLxCor-Height>=RadT).and.(MLyCor+RowRad>=RadT))then
                                        if (RA(MLyCor+RowRad, MLxCor-Height) /=
'-') then
                                                RA(MLyCor+RowRad,      MLxCor-
Height) = '1'
                                                RAMolClose(MLyCor+RowRad,
MLxCor-Height) = a
                                        end if
                                end if
```

```
if ((MLyCor-RowRad>=RadT).and.(MLxCor-height<=GridSize-RadT).and.(MLxCor-height>=RadT).and.(MLyCor-RowRad<=GridSize))then
    if (RA(MLyCor-RowRad, MLxCor-Height) /= '-') then
        RA(MLyCor-RowRad, MLxCor-Height) = '1'
        RAMolClose(MLyCor-RowRad, MLxCor-Height) = a
    end if
end if
do count2 = -RowRad, RowRad
    if ((MLyCor+count2<=GridSize).and.(MLxCor+height<=GridSize-RadT).and.(MLxCor+height>=RadT).and.(MLyCor+count2>=RadT))then
        if (RA(MLyCor+count2, MLxCor+Height) /= '-')  then
            RA(MLyCor+count2, MLxCor+Height) = '1'
            RAMolClose(MLyCor+count2, MLxCor+Height) = a
        end if
    end if
    if ((MLyCor-count2>=RadT).and.(MLxCor+height<=GridSize-RadT).and.(MLxCor+height>=RadT).and.(MLyCor-count2<=GridSize))then
        if (RA(MLyCor-count2, MLxCor+Height) /= '-') then
            RA(MLyCor-count2, MLxCor+Height) = '1'
            RAMolClose(MLyCor-count2, MLxCor+Height) = a
        end if
    end if
    if ((MLyCor+count2>=RadT).and.(MLxCor-height<=GridSize-RadT).and.(MLxCor-height>=RadT).and.(MLyCor+count2<=GridSize))then
        if (RA(MLyCor+count2, MLxCor-Height) /= '-') then
            RA(MLyCor+count2, MLxCor-Height) = '1'
            RAMolClose(MLyCor+count2, MLxCor-Height) = a
        end if
    end if
    if ((MLyCor-count2<=GridSize).and.(MLxCor-height<=GridSize-RadT).and.(MLxCor-height>=RadT).and.(MLyCor-count2>=RadT))then
        if (RA(MLyCor-count2, MLxCor-Height) /= '-') then
            RA(MLyCor-count2, MLxCor-Height) = '1'
            RAMolClose(MLyCor-count2, MLxCor-Height) = a
        end if
    end if
end do
```

```
    end do
end do

do a = 1, GridSize ! Finds the valid points and saves them to an array
    do b = 1, GridSize
        if (RA(b,a) == '1') then
            ReWLoop: do c = 1, OneCount
                if ((Ones(c,2) == a) .and. (Ones(c,1) < b)) then
                    Ones(c,1) = b
                    exit ReWLoop
                end if
            end do ReWLoop
            if ((c-1 == OneCount) .or. (OneCount == 1)) then
                Ones(OneCount, 1) = b
                Ones(OneCount, 2) = a
                OneCount = OneCount + 1
            end if
        end if
    end do
end do

do a = 1, OneCount - 1 ! Finds the closest of these points to the impact location and moves the particle to it
    TempX = Ones(a,2)
    TempY = Ones(a,1)

    if (TempY <= y+1) then
        Dist = ((x-TempX)**2)+((y-TempY)**2)
        Dist = sqrt(Dist)
        if ((Dist < SavDist) .or. (SavDist == 0)) then
            SavDist = Dist
            SavOneX = Ones(a,2)
            SavOneY = Ones(a,1)
        end if
    end if
end do

TempX = SavOneX
TempY = SavOneY

if ((TempX == 0) .and. (TempY == 0)) then
    GO TO 10
end if

SavOnePart = RAMolClose(TempY, TempX)

SafeLocCount = 0
SavIncremX = SavOneX
SavIncremY = SavOneY
SafeLocFound = .FALSE.
FinalSavX = 0
FinalSavY = 0
LR = 0
RealPos1 = 0
RealPos2 = 0
```

```
if (MLxReal(SavOnePart) == TempX) then! Determines which way the particle should roll
        call random_number(RX)
        LRNo = 1 + floor(2*RX)
        if (LRNo == 1) then
                LR = -1
        elseif (LRNo == 2) then
                LR = 1
        end if
elseif (MLxReal(SavOnePart) < TempX) then
        LR = 1
        Upwards = .TRUE.
elseif (MLxReal(SavOnePart) > TempX) then
        LR = -1
        Downwards = .TRUE.
end if

RMPInter = 0
RMPPrev = SavOnePart
RealPos1 = SavOnePart

do a = 1, OneCount-1
        if (Ones(a,2) == TempX) then
                SideCount = a
        end if
end do

do while (SafeLocFound .eqv. .FALSE.) ! Iterates in that direction until the next position would be higher, or reaching the edge of the box
        if ((Ones(SideCount+LR,1) > Ones(SideCount,1)) .or. (Ones(SideCount,2) == RadT) .or. (Ones(SideCount,2) == GridSize-RadT)) then
                SafeLocFound = .TRUE.
        else
                SideCount = SideCount + LR
        end if
end do

19      CONTINUE

TempX = Ones(SideCount+AdjCount,2)
TempY = Ones(SideCount+AdjCount,1)

if (TempX > GridSize-RadT) then
        TempX = GridSize-RadT
        TempY = Ones(SideCount-AdjCount,1)
elseif (TempX < RadT) then
        TempX = RadT
        TempY = Ones(SideCount-AdjCount,1)
end if

do a = 1, MolNo-1 ! Finds the particles closest to the low point for the new particle to be resting on
        if (MLyReal(a) < TempY+RadT) then
                Dist = ((MLxReal(a)-TempX)**2)+((MLyReal(a)-TempY)**2)

                Dist = sqrt(Dist) - RadT - MLr(a)
```

```
                    if (Dist <= FinalPart(1,2)) then
                        FinalPart(2,1) = FinalPart(1,1)
                        FinalPart(2,2) = FinalPart(1,2)
                        FinalPart(2,3) = FinalPart(1,3)
                        FinalPart(1,1) = a
                        FinalPart(1,2) = Dist
                        FinalPart(1,3) = Dist + RadT + MLr(a)
                    elseif (Dist <= FinalPart(2,2)) then
                        FinalPart(2,1) = a
                        FinalPart(2,2) = Dist
                        FinalPart(2,3) = Dist + RadT + MLr(a)
                    end if
                end if
            end do

50          CONTINUE

            PartCoords(1,1) = MLxReal(FinalPart(1,1))
            PartCoords(1,2) = MLyReal(FinalPart(1,1))

            PartCoords(2,1) = MLxReal(FinalPart(2,1))
            PartCoords(2,2) = MLyReal(FinalPart(2,1))

            if ((TempX == RadT) .or. (TempX == GridSize-RadT)) then
                EdgeCase = .TRUE.
            end if

            if (EdgeCase .eqv. .TRUE.) then ! If the particle is on an edge, it
balances the new particle on the edge + one particle
                NewX = TempX

                Dist = RadT + MLr(FinalPart(1,1))

                NewY = -
(MLxReal(FinalPart(1,1))**2)+(2*MLxReal(FinalPart(1,1))*NewX)
                NewY = NewY+(Dist**2)-(NewX**2)
                NewY = sqrt(NewY)

                if (NewY /= NewY) then
71                  CONTINUE
                    if (FirstEdge .eqv. .TRUE.) then
                        do a = 1, MolNo-1
                            Dist = (MLxReal(a)-NewX)
                            if (Dist < RadT + MLr(a) +
(RadT*8)) then
                                EdgeCombi(EdgeCount) =
a
                                EdgeCount = EdgeCount +
1
                            end if
                        end do
                    end if

                    FirstEdge = .FALSE.

                    if (EdgeAttempt <= EdgeCount-1) then
```

```
                    FinalPart(1,1) = EdgeCombi(EdgeCount-
EdgeAttempt-1)
                    EdgeAttempt = EdgeAttempt + 1
                    GO TO 50
                end if

                FullCount = FullCount + 1
                GO TO 10
            end if

            if ((TempY - (MLyReal(FinalPart(1,1)) + NewY)) < (TempY -
(MLyReal(FinalPart(1,1)) - NewY))) then
                NewY = MLyReal(FinalPart(1,1)) + NewY
            else
                NewY = MLyReal(FinalPart(1,1)) - NewY
            end if

            do a = 1, MolNo-1       ! Confirming the new particle is not
overlapping with any other particles
                Dist = ((MLxReal(a)-NewX)**2)+((MLyReal(a)-
NewY)**2)
                Dist = sqrt(Dist)
                if (Dist < MLr(a)+RadT-0.1) then
                    OverlapCount = OverlapCount + 1
                    if (OverlapCount > 2500) then
                        GO TO 71
                    end if

                    FinalPart(1,1) = a
                    FullCount = FullCount + 1
                    GO TO 50
                end if
            end do

            if ((NewX < RadT) .or. (NewY < RadT) .or. (NewX > GridSize-
RadT) .or. (NewY > GridSize-RadT)) then
                FullCount = FullCount + 1
                GO TO 10
            end if

        else ! else if the particle is not on an edge it balances on two particles
through stochastic optimisation

            newCoords = 0
            stochDists = 0
            sumDist = 0
            dx = 0
            stochxnew = 0
            ibad = 0

            newCoords(1) = TempX
            newCoords(2) = TempY

            stochDists(1) = FinalPart(1,3)
            stochDists(2) = FinalPart(2,3)

            sumDist = stochDists(1) + stochDists(2)
```

```
dx(1) = 10 * RadLarge
dx(2) = 10 * RadLarge

do a = 1, 10
	do b = 1, 2
		dx(b) = dx(b) / 10
	end do

	do c = 1, 500
		call random_number(RX)
		stochxnew(2) = newCoords(2) + (dx(2) *
((2*RX)-1))
		call random_number(RX)
		stochxnew(1) = newCoords(1) + (dx(1) *
((2*RX)-1))

		ibad = 0

		do b = 1, 2
			stochDists(b) = ((PartCoords(b,1) -
stochxnew(1))**2) + ((PartCoords(b,2)-stochxnew(2))**2)
			stochDists(b) = sqrt(stochDists(b)) -
RadT - MLr(FinalPart(b,1))
			if (stochDists(b) < 0) then
				ibad = 1
			end if
		end do

		stochynew = stochDists(1) + stochDists(2)
		if ((stochynew < sumDist) .and. (ibad == 0))
then
			do b = 1, 2
				newCoords(b)           =
stochxnew(b)
			end do
			sumDist = stochynew
		end if
	end do
end do

NewX = newCoords(1)
NewY = newCoords(2)

do a = 1, MolNo-1
	Dist    =    ((MLxReal(a)-NewX)**2)+((MLyReal(a)-
NewY)**2)
	Dist = sqrt(Dist)
	if (Dist < MLr(a)+RadT) then
		OverDist = 0
		OverlapCount = OverlapCount + 1
		if (OverlapCount > 2500) then
			NotBal = .TRUE.
			GO TO 70
		end if

		do b = 1, 2
```

```
                                                Dist            =            ((MLxReal(a)-
PartCoords(b,1))**2)+((MLyReal(a)-PartCoords(b,2))**2)
                                                Dist = sqrt(Dist)
                                                if ((Dist < OverDist) .or. (OverDist ==
0)) then
                                                        OverDist = Dist
                                                        OverDistNo = b
                                                end if
                                        end do
                                        FinalPart(OverDistNo,1) = a
                                        if (FinalPart(1,1) == FinalPart(2,1)) then
                                                NotBal = .TRUE.
                                                GO TO 70
                                        end if
                                        FullCount = FullCount + 1
                                        if (OverDist == 0) then
                                                NotBal = .TRUE.
                                                GO TO 70
                                        else
                                                GO TO 50
                                        end if
                                end if
                        end do

                        NotBal = .FALSE.
                        if (HMAdju .eqv. .FALSE.) then
                                if  ((NewX  >  PartCoords(1,1)+0.25)  .and.  (NewX  >
PartCoords(2,1)+0.25)) then ! Confirms the new particle is resting in between the old particles
                                        NotBal = .TRUE.
                                elseif ((NewX < PartCoords(1,1)-0.25) .and. (NewX
< PartCoords(2,1)-0.25)) then
                                        NotBal = .TRUE.
                                end if
                        end if

                        do b = 1, 2
                                Dist      =      ((NewX-PartCoords(b,1))**2)+((NewY-
PartCoords(b,2))**2)

                                Dist = sqrt(Dist)
                                if (Dist > RadT + MLr(FinalPart(b,1))+0.25) then
                                        NotBal = .TRUE.
                                end if
                        end do

                        Dy = (PartCoords(1,2) - PartCoords(2,2)) / (PartCoords(1,1)
- PartCoords(2,1))

                        intC =  PartCoords(1,2) - (Dy * PartCoords(1,1))
                        CheckY = (Dy * NewX) + intC
                        if (NewY < CheckY) then
                                NotBal = .TRUE.
                        end if

70                      CONTINUE

                        OverlapCount = 0
```

```
if (NotBal .eqv. .TRUE.) then    ! If the particle is not correctly balancing then it looks for alternate particles to be resting on and moves to that location
    if (FirstBal .eqv. .TRUE.) then
        do a = 1, MolNo-1
            Dist = ((MLxReal(a)-NewX)**2)
            Dist = sqrt(Dist)
            if ((Dist < RadT + MLr(a) + (RadT*8)) .and. (NewY-MlyReal(a) < RadT + MLr(a) + (RadT*6))) then

                BalCheckNo(BalCheckCount) = a
                BalCheckCount = BalCheckCount + 1
            end if
        end do
        aLoop: do a = BalCheckCount-1, 1, -1
            do b = BalCheckCount-1, 1, -1
                if (BalCheckNo(b) < BalCheckNo(a)) then
                    if (BalCount <= 100000) then
                        if (abs(MlxReal(BalCheckNo(a)) – MlxReal(BalCheckNo(b))) <= RadT*2 + MLr(BalCheckNo(a)) + MLr(BalCheckNo(b))) then

                            BalCombi(BalCount,1) = BalCheckNo(a)

                            BalCombi(BalCount,2) = BalCheckNo(b)

                            BalCount = BalCount + 1
                        end if
                    else
                        exit aLoop
                    end if
                end if
            end do
        end do aLoop
    end if

    FirstBal = .FALSE.

    if (BalAttempt <= BalCount-1) then
        do while (BalCombi(BalCount-BalAttempt,1) == 0 .or. BalCombi(BalCount-BalAttempt,2) == 0)
            BalAttempt = BalAttempt + 1
        end do
        FinalPart(1,1) = BalCombi(BalCount-BalAttempt,1)

        FinalPart(2,1) = BalCombi(BalCount-BalAttempt,2)

        BalAttempt = BalAttempt + 1

        if (BalAttempt <= 100000) then
            GO TO 50
        end if
    end if
```

```
                                        if (TempX >= GridSize-(RadT*5)) then
                                                TempX = GridSize-RadT
                                                EdgeCase = .TRUE.
                                                GO TO 50
                                        elseif (TempX <= RadT+(RadT*5)) then
                                                TempX = RadT
                                                EdgeCase = .TRUE.
                                                GO TO 50
                                        end if

                                        if ((AdjCount < 75) .and. (AdjCount > -75)) then
                                                AdjCount = AdjCount – LR
                                                HMAdju = .TRUE.
                                                GO TO 19
                                        elseif (AdjCount < -75 .and. Downwards .eqv.
.TRUE.) then
                                                Downwards = .FALSE.
                                                AdjCount = 0
                                                LR = 1
                                                GO TO 19
                                        elseif (AdjCount > 75 .and. Upwards .eqv. .TRUE.)
then
                                                Downwards = .TRUE.
                                                AdjCount = 0
                                                LR = -1
                                                GO TO 19
                                        else
                                                GO TO 10
                                        end if

                                        GO TO 10

                                end if

                        end if

                        if ((NewX > GridSize-RadT) .or. (NewX < RadT) .or. (NewY >
GridSize-RadT) .or. (NewY < RadT)) then
                                if (TempX >= GridSize-RadT-RadT) then
                                        TempX = GridSize-RadT
                                        EdgeCase = .TRUE.
                                        GO TO 50
                                elseif (TempX <= RadT+RadT) then
                                        TempX = RadT
                                        EdgeCase = .TRUE.
                                        GO TO 50
                                end if
                                GO TO 10
                        end if

                        MlxReal(MolNo) = NewX           ! Saves the first particle location
                        MlyReal(MolNo) = NewY
                        MLr(MolNo) = RadT
                        FullCount = 0
                        MolNo = MolNo + 1

                        RA = ‘0’
```

```
			RAMolClose = 0
		else
			MlxReal(MolNo) = x
			MlyReal(MolNo) = y
			MLr(MolNo) = RadT
			FullCount = 0
			MolNo = MolNo + 1
		end if

		! The first particle in the chain placed, now moving onto placing the rest of the chain

		RA = ‘’
		OverlapCount = 0

		MlxCor = MlxReal(MolNo-1)
		MlyCor = MlyReal(MolNo-1)
		RadIn = MLr(MolNo-1)
		DoubRad = (RadIn+RadT)+1

		do Height = -RadIn, 0 ! The contour plot is remade similar to the first time, however valid points are only placed attached to the particle just added to the system
			MidWay = RadIn**2 – Height**2
			RowRad = abs(sqrt(Midway))
			if ((MlyCor+RowRad<=GridSize).and.(MlxCor+Height<=GridSize-RadT).and.(MlxCor+Height>=RadT).and.(MlyCor+RowRad>=RadT))then
				if (MlyCor+RowRad>= MlyCor) then
				do a = 1, MolNo-2
					Dist = ((MlxReal(a)-(MlxCor+Height))**2) + ((MlyReal(a)-(MlyCor+RowRad))**2)
					Dist = sqrt(Dist)
					if (Dist < RadIn + MLr(a)) then
						GO TO 15
					end if
				end do
				RA(MlyCor+RowRad, MlxCor+Height) = ‘1’
				end if
			end if
15					CONTINUE
			if ((MlyCor-RowRad>=RadT).and.(MlxCor+Height<=GridSize-RadT).and.(MlxCor+height>=RadT).and.(MlyCor-RowRad<=GridSize))then
				if (MlyCor-RowRad>= MlyCor) then
				do a = 1, MolNo-2
					Dist = ((MlxReal(a)-(MlxCor+Height))**2) + ((MlyReal(a)-(MlyCor-RowRad))**2)
					Dist = sqrt(Dist)
					if (Dist < RadIn + MLr(a)) then
						GO TO 16
					end if
				end do
				RA(MlyCor-RowRad, MlxCor+Height) = ‘1’
				end if
			end if
16					CONTINUE
			if ((MlyCor+RowRad<=GridSize).and.(MlxCor-Height<=GridSize-RadT).and.(MlxCor-Height>=RadT).and.(MlyCor+RowRad>=RadT))then
				if (MlyCor+RowRad>= MlyCor) then
```

```
                        do a = 1, MolNo-2
                                Dist = ((MlxReal(a)-(MlxCor-Height))**2) + ((MlyReal(a)-(MlyCor+RowRad))**2)
                                Dist = sqrt(Dist)
                                if (Dist < RadIn + MLr(a)) then
                                        GO TO 17
                                end if
                        end do
                        RA(MlyCor+RowRad, MlxCor-Height) = '1'
                        end if
                end if
17                              CONTINUE
                if ((MlyCor-RowRad>=RadT).and.(MlxCor-Height<=GridSize-RadT).and.(MlxCor-height>=RadT).and.(MlyCor-RowRad<=GridSize))then
                        if (MlyCor-RowRad>= MlyCor) then
                        do a = 1, MolNo-2
                                Dist = ((MlxReal(a)-(MlxCor-Height))**2) + ((MlyReal(a)-(MlyCor-RowRad))**2)
                                Dist = sqrt(Dist)
                                if (Dist < RadIn + MLr(a)) then
                                        GO TO 18
                                end if
                        end do
                        RA(MlyCor-RowRad, MlxCor-Height) = '1'
                        end if
                end if
18                              CONTINUE
        end do

        ChainOneCount = 1

        do a = 1, GridSize
                do b = 1, GridSize
                        if (RA(b,a) == '1') then
                                ChainOnes(ChainOneCount, 1) = b
                                ChainOnes(ChainOneCount, 2) = a
                                ChainOneCount = ChainOneCount + 1
                        end if
                end do
        end do

        call random_number(RX)
        LRNo = 1 + floor(2*RX)
14      CONTINUE

        if (NLCount >= 250) then
                OverFallCount = OverFallCount + 1
                MolNo = MolNo – 1
                if (OverFallCount >= 100) then
                        Full = 1
                end if
                GO TO 10
        end if

        if (LRNo == 1) then ! Picks whether the chain should fall left or right
                TempX = ChainOnes(1,2)
                TempY = ChainOnes(1,1)
```

```
            elseif (LRNo == 2) then
                TempX = ChainOnes(ChainOneCount-1,2)
                TempY = ChainOnes(ChainOneCount-1,1)
            end if

            if  (MlxReal(MolNo-1) == GridSize-RadT) then
                TempX = ChainOnes(1,2)
                TempY = ChainOnes(1,1)
            elseif (MlxReal(MolNo-1) == RadT) then
                TempX = ChainOnes(ChainOneCount-1,2)
                TempY = ChainOnes(ChainOneCount-1,1)
            end if

            if (TempX == 0 .or. TempY == 0) then
                TempX = MlxReal(MolNo-1)
                TempY = MlyReal(MolNo-1)+MLr(MolNo-1)
            end if

            FinalPart = 99999
            FinalPart(1,1) = MolNo-1
            Dist = ((MlxReal(a)-TempX)**2)+((MlyReal(a)-TempY)**2)
            Dist = sqrt(Dist) – MLr(a)
            FinalPart(1,2) = Dist
            FinalPart(1,3) = Dist + MLr(a)

            do a = 1, MolNo-2
                if (MlyReal(a) < TempY+1) then
                    Dist = ((MlxReal(a)-TempX)**2)+((MlyReal(a)-TempY)**2)
                    Dist = sqrt(Dist) – RadT – MLr(a)
                    if (Dist <= FinalPart(2,2)) then
                        FinalPart(2,1) = a
                        FinalPart(2,2) = Dist
                        FinalPart(2,3) = Dist + RadT + MLr(a)
                    end if
                end if
            end do

60          CONTINUE

            PartCoords(1,1) = MlxReal(FinalPart(1,1))
            PartCoords(1,2) = MlyReal(FinalPart(1,1))
            PartCoords(2,1) = MlxReal(FinalPart(2,1))
            PartCoords(2,2) = MlyReal(FinalPart(2,1))

            EdgeCase = .FALSE.
            TopCase = .FALSE.

            if ((TempX <= RadT) .or. (TempX >= GridSize-RadT)) then
                EdgeCase = .TRUE.
            elseif (TempY >= GridSize-RadT) then
                TopCase = .TRUE.
            end if

            if (EdgeCase .eqv. .TRUE.) then ! Does the same as above but resting the
particle against the edge attached to the first particle
                NewX = TempX
                Dist = MLr(FinalPart(1,1))
```

```
                    NewY = -(MlxReal(FinalPart(1,1))**2)+(2*MlxReal(FinalPart(1,1))*NewX)
                    NewY = NewY+(Dist**2)-(NewX**2)
                    NewY = sqrt(NewY)

                    if (NewY /= NewY) then
                          MolNo = MolNo – 1
                          GO TO 10
                    end if

                    if (TempY – (MlyReal(FinalPart(1,1)) + NewY) < TempY – (MlyReal(FinalPart(1,1)) – NewY)) then
                          NewY = MlyReal(FinalPart(1,1)) + NewY
                    else
                          NewY = MlyReal(FinalPart(1,1)) – NewY
                    end if
              elseif (TopCase .eqv. .TRUE.) then
                    NewY = TempY
                    Dist = MLr(FinalPart(1,1)) ! Does the same as above but resting the particle against the roof of the box attached to the first particle

                    NewX = -(MlyReal(FinalPart(1,1))**2)+(2*MlyReal(FinalPart(1,1))*NewY)
                    NewX = NewX+(Dist**2)-(NewY**2)
                    NewX = sqrt(NewX)

                    if (NewX /= NewX) then
                          MolNo = MolNo – 1
                          GO TO 10
                    end if

                    if (TempX – (MlxReal(FinalPart(1,1)) + NewX) < TempX – (MlxReal(FinalPart(1,1)) – NewX)) then
                          NewX = MlxReal(FinalPart(1,1)) + NewX
                    else
                          NewX = MlxReal(FinalPart(1,1)) – NewX
                    end if

              else ! Does the same as above resting the particle against a particle while still being attached to the first particle

                    newCoords = 0
                    stochDists = 0
                    sumDist = 0
                    dx = 0
                    stochxnew = 0
                    ibad = 0

                    newCoords(1) = TempX
                    newCoords(2) = TempY

                    stochDists(1) = FinalPart(1,3)
                    stochDists(2) = FinalPart(2,3)

                    sumDist = stochDists(1) + stochDists(2)
```

```
				dx(1) = 10 * RadLarge
				dx(2) = 10 * RadLarge

				do a = 1, 10
					do b = 1, 2
						dx(b) = dx(b) / 10
					end do
					do c = 1, 500
						call random_number(RX)
						stochxnew(2) = newCoords(2) + dx(2) * ((2*RX)-1)
						call random_number(RX)
						stochxnew(1) = newCoords(1) + dx(1) * ((2*RX)-1)

						ibad = 0

						stochDists(1) = ((PartCoords(1,1)-stochxnew(1))**2)+((PartCoords(1,2)-stochxnew(2))**2)
						stochDists(1) = sqrt(stochDists(1)) – MLr(FinalPart(1,1))
						if (stochDists(1) < 0) then
							ibad = 1
						end if

						stochDists(2) = ((PartCoords(2,1)-stochxnew(1))**2)+((PartCoords(2,2)-stochxnew(2))**2)
						stochDists(2) = sqrt(stochDists(2)) – RadT – MLr(FinalPart(2,1))
						if (stochDists(2) < 0) then
							ibad = 1
						end if

						stochynew = stochDists(1) + stochDists(2)
						if ((stochynew < sumDist) .and. (ibad == 0)) then
							do b = 1, 2
								newCoords(b) = stochxnew(b)
							end do
							sumDist = stochynew
						end if
					end do
				end do

				ContPoint = FinalPart(2,1)
				NewX = newCoords(1)
				NewY = newCoords(2)
			end if

			if ((NewX > GridSize-RadT) .or. (NewX < RadT) .or. (NewY > GridSize-RadT) .or. (NewY < RadT)) then
				if (EdgeCase .eqv. .FALSE.) then
					EdgeCase = .TRUE.
				end if
				MolNo = MolNo – 1
				GO TO 10
			end if

			do a = 1, MolNo-2
				Dist = ((MlxReal(a)-NewX)**2)+((MlyReal(a)-NewY)**2)
```

```
Dist = sqrt(Dist)
if (Dist < MLr(a)+RadT) then
    OverlapCount = OverlapCount + 1
    if (OverlapCount > 2500) then
        if (Tilt .eqv. .FALSE.) then
            OverlapCount = 0
            if (LRNo == 1) then
                LRNo = 2
            else
                LRNo = 1
            end if
            Tilt = .TRUE.
            GO TO 14
        end if
        MolNo = MolNo – 1
        if ((AdjCount < 75) .and. (AdjCount > -75)) then
            AdjCount = AdjCount – LR
            HMAdju = .TRUE.
            GO TO 19
        elseif (AdjCount < -75 .and. Downwards .eqv.
.TRUE.) then
            Downwards = .FALSE.
            AdjCount = 0
            LR = 1
            GO TO 19
        elseif (AdjCount > 75 .and. Upwards .eqv. .TRUE.)
then
            Downwards = .TRUE.
            AdjCount = 0
            LR = -1
            GO TO 19
        else
            GO TO 10
        end if
        GO TO 10
    end if

    FinalPart(2,1) = a
    GO TO 60

    FullCount = FullCount + 1
    MolNo = MolNo – 1
    GO TO 10
end if
end do

MlxReal(MolNo) = NewX ! Saves the second particle in the chain
MlyReal(MolNo) = NewY
MLr(MolNo) = RadT
FullCount = 0
MolNo = MolNo + 1

ChainStartNo = MolNo-2 ! If the requested chain length is longer than 2 then
the following code is done
ChainStartCont = MolNo-1
CSCy = MlyReal(ChainStartCont)
CSCx = NewX
```

```
        do a = 1, ChainLength – 2 ! Loops for the chain particles beyond the first two

            OverlapCount = 0

            ChainGrad = (MlyReal(ChainStartCont) – MlyReal(ChainStartNo)) / (MlxReal(ChainStartCont) – MlxReal(ChainStartNo))
            EquC = MlyReal(ChainStartCont) – (ChainGrad * MlxReal(ChainStartCont))

            if (MlxReal(ChainStartCont) – MlxReal(ChainStartNo) == 0) then ! If particle is vertical then places the new one on top
                NewX = MlxReal(ChainStartNo)
                NewY = MlyReal(ChainStartNo+a) + RadT
            else ! Otherwise works out the gradient and then angle of the 2 particle chain to add the following particles onto
                NewX = RadT / (sqrt((ChainGrad**2) + 1))
                NewY = (ChainGrad * NewX)

                if (MlxReal(ChainStartCont) < MlxReal(ChainStartNo)) then
                    if (MlyReal(ChainStartCont) > MlyReal(ChainStartNo)) then
                        NewX = MlxReal(MolNo-1) – NewX
                        NewY = MlyReal(MolNo-1) – NewY
                    else
                        NewX = MlxReal(MolNo-1) – NewX
                        NewY = MlyReal(MolNo-1) – NewY
                    end if
                elseif (MlxReal(ChainStartCont) > MlxReal(ChainStartNo)) then
                    if (MlyReal(ChainStartCont) > MlyReal(ChainStartNo)) then
                        NewX = MlxReal(MolNo-1) + NewX
                        NewY = MlyReal(MolNo-1) + NewY
                    else
                        NewX = MlxReal(MolNo-1) + NewX
                        NewY = MlyReal(MolNo-1) + NewY
                    end if
                end if
            end if

98          CONTINUE

            ChainHitCheck = .FALSE.
            ChainEdgeHitCheck = .FALSE.
            ChainTopHitCheck = .FALSE.

            ChainHitLoop: do b = 1, MolNo – a-2     ! Checks if the new part of the chain is overlapping with anything
                Dist = ((MlxReal(b)-NewX)**2)+((MlyReal(b)-NewY)**2)
                Dist = sqrt(Dist)
                if (Dist < ((RadT)+(Mlr(b)))) then
                    ChainHitCheck = .TRUE.
                    PartHit = b
                end if
            end do ChainHitLoop
```

```
99                      CONTINUE

                        if ((NewX > GridSize-RadT) .or. (NewX < RadT)) then ! Checks if the
new part of the chain is outside of the box
                                ChainEdgeHitCheck = .TRUE.
                        elseif (NewY > GridSize-RadT) then
                                ChainTopHitCheck = .TRUE.
                        end if

                        if (ChainHitCheck .eqv. .TRUE.) then ! If overlapping then the particle
moves to be resting upon the particle it is overlapping

                                if (OverlapCount > 2500) then
                                        MolNo = MolNo – a
                                        if (Tilt .eqv. .FALSE.) then
                                                OverlapCount = 0
                                                if (LRNo == 1) then
                                                        LRNo = 2
                                                else
                                                        LRNo = 1
                                                end if
                                                Tilt = .TRUE.
                                                GO TO 14
                                        end if
                                        MolNo = MolNo – 1
                                        if ((AdjCount < 75) .and. (AdjCount > -75)) then
                                                AdjCount = AdjCount – LR
                                                HMAdju = .TRUE.
                                                GO TO 19
                                        elseif (AdjCount < -75 .and. Downwards .eqv.
.TRUE.) then
                                                Downwards = .FALSE.
                                                AdjCount = 0
                                                LR = 1
                                                GO TO 19
                                        elseif (AdjCount > 75 .and. Upwards .eqv. .TRUE.)
then
                                                Downwards = .TRUE.
                                                AdjCount = 0
                                                LR = -1
                                                GO TO 19
                                        else
                                                GO TO 10
                                        end if
                                        GO TO 10
                                end if

                                newCoords = 0
                                stochDists = 0
                                sumDist = 0
                                dx = 0
                                stochxnew = 0
                                ibad = 0

                                newCoords(1) = NewX
                                newCoords(2) = NewY
```

```
stochDists(1) = Dist
stochDists(2) = ((MlxReal(MolNo-a-1)-NewX)**2)+((MlyReal(MolNo-a-1)-NewY)**2)
stochDists(2) = sqrt(stochDists(2))

sumDist = stochDists(1) + stochDists(2)

dx(1) = 10 * RadLarge
dx(2) = 10 * RadLarge

do b = 1, 10
        do c = 1, 2
                dx© =©(c) / 10
        end do
        do c = 1, 500
                call random_number(RX)
                stochxnew(2) = newCoords(2) + dx(2) * ((2*RX)-1)
                call random_number(RX)
                stochxnew(1) = newCoords(1) + dx(1) * ((2*RX)-1)

                ibad = 0

                stochDists(1) = ((MLxReal(PartHit)-stochxnew(1))**2)+((MLyReal(PartHit)-stochxnew(2))**2)
                stochDists(1) = sqrt(stochDist–(1)) - (–adT) - (MLr(PartHit))
                if (stochDists(1) < 0) then
                        ibad = 1
                end if

                stochDists(2) = ((MLxReal(MolNo-a-1)-stochxnew(1))**2)+((MLyReal(MolNo-a-1)-stochxnew(2))**2)
                stochDists(2) = sqrt(stochDist–(2)) - ((a+1) * RadT)
                if (stochDists(2) < 0) then
                        ibad = 1
                end if

                stochynew = stochDists(1) + stochDists(2)
                if ((stochynew < sumDist) .and. (ibad == 0)) then
                        do d = 1, 2
                                newCoords(d) = stochxnew(d)
                        end do
                        sumDist = stochynew
                end if
        end do
end do

ContPoint = PartHit
NewX = newCoords(1)
NewY = newCoords(2)

if (NewY < MLyReal(PartHit)) then
```

```
                AngleFix: do b = 1, MolNo-a-2
                    Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
                    Dist = sqrt(Dist)
                    if (Dist < MLr(b)+RadT) then
                        Valdx = abs(MLxReal(MolNo–a-1) - NewX)
                        ValDist = ((MLxReal(MolNo-a-1)-NewX)**2)+((MLyReal(MolNo-a-1)-NewY)**2)
                        ValDist = sqrt(ValDist)
                        ValAng1 = asin(Valdx/ValDist)
                        ValDist2 = ((MLxReal(MolNo-a-1)-MLxReal(PartHit))**2)+((MLyReal(MolNo-a-1)-MLyReal(PartHit))**2)
                        ValDist2 = sqrt(ValDist2)
                        ValDist3 = ((newX-MLxReal(PartHit))**2)+((newY-MLyReal(PartHit))**2)
                        ValDist3 = sqrt(ValDist3)
                        ValAng2a = (ValDist2**2) + (ValDis–**2) - (ValDist3**2)
                        ValAng2b = 2 * ValDist * ValDist2
                        ValAng2 = acos(ValAng2a/ValAng2b)
                        if ((ValAng2 /= ValAng2) .or. ValAng2 < 0.01) then
                            ValAng3 =–(Pi) - ValAng1
                        else
                            ValAng3 =–(Pi) - Va–Ang1 - (ValAng2*2)
                            if (MLxReal(PartHit) > MLxReal(MolNo-a-1)) then
                                NewX = MLxReal(MolNo-a-1) + (sin(ValAng3) * ValDist)
                            else
                                NewX = MLxReal(MolNo–a-1) - (sin(ValAng3) * ValDist)
                            end if
                            NewY = MLyReal(MolNo-a-1) + (cos(ValAng3) * ValDist)
                        end if
                        exit AngleFix
                    end if
                end do AngleFix
            end if

            ChainHitCheck = .FALSE.

            do b = 1, –olNo - a-2
                Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
                Dist = sqrt(Dist)
                if (Dist < ((RadT)+(Mlr(b)))) then
                    ChainHitCheck = .TRUE.
                    PartHit = b
                end if
```

```
                end do

                if (ChainHitCheck .eqv. .TRUE.) then
                        OverlapCount = OverlapCount + 1
                        GO TO 99
                end if

                if ((NewX < RadT) .or. (NewX > Gri–Size - RadT)) then
                        ChainHitCheck = .FALSE.
                        ChainEdgeHitCheck = .TRUE.
                        GO TO 99
                end if
                if (NewY > Gri–Size - RadT) then
                        ChainHitCheck = .FALSE.
                        ChainTopHitCheck = .TRUE.
                        GO TO 99
                end if

                ChainGrad = –NewY - MLyReal(ChainStartNo)) / –NewX -
MLxReal(ChainStartNo))
                EquC =–NewY - (ChainGrad * NewX)
                CSCy = NewY
                CSCx = NewX

                do d = 1, a
                        OverlapCount = 0

                        if –NewX - MLxReal(ChainStartNo) == 0) then
                                MLxReal(MolNo-d) = NewX
                                CSCy =–CSCy - RadT
                                MLyReal(MolNo-d) = CSCy
                        else
                                MLxReal(MolNo-d)       =       RadT       /
(sqrt((ChainGrad**2) + 1))
                                MLyReal(MolNo-d)      =      (ChainGrad      *
MLxReal(MolNo-d))

                                if (NewX < MLxReal(ChainStartNo)) then
                                        CSCy = CScy + MLyReal(MolNo-d)
                                        CSCx = CScx + MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
                                elseif   (NewX   >   MLxReal(ChainStartNo))
then
                                        CSCy =–CScy - MLyReal(MolNo-d)
                                        CSCx =–CScx - MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
                                end if
                        end if
                end do
        end if

        if  (ChainEdgeHitCheck  .eqv.  .TRUE.)  then  !  If  the  particle  is
overlapping with an edge then it is moved to be resting against it
                if (OverlapCount > 2500) then
                        MolNo = –olN– - a - 1
```

```
                GO TO 10
        end if
        if (NewX > GridSize-RadT) then
                NewX = GridSize-RadT
        elseif (NewX < RadT) then
                NewX = RadT
        end if
        Dist = (a+1)*RadT

        NewY = -(MLxReal(MolNo-a-1)**2)+(2*MLxReal(MolNo-a-
1)*NewX)

        NewY = NewY+(Dist**2)-(NewX**2)
        NewY = sqrt(NewY)

        if (NewY /= NewY) then
                MolNo = –olN– - a - 1
                GO TO 10
        end if

        if (–empy - (MLyReal(MolNo-a-1) + NewY) < –empy -
(MLyReal(MolNo–a-1) - NewY)) then
                NewY = MLyReal(MolNo-a-1) + NewY
        else
                NewY = MLyReal(MolNo–a-1) - NewY
        end if

        do b = 1, –olNo - a-2
                Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-
NewY)**2)

                Dist = sqrt(Dist)
                if (Dist < ((RadT)+(Mlr(b)))) then
                        ChainHitCheck = .TRUE.
                        PartHit = b
                        OverlapCount = OverlapCount + 1
                        GO TO 99
                end if
        end do

        ChainGrad = –NewY - MLyReal(ChainStartNo)) / –NewX -
MLxReal(ChainStartNo))

        EquC =–NewY - (ChainGrad * NewX)
        CSCy = NewY
        CSCx = NewX

        do d = 1, a ! The previous particle in the chain are moved to
be in line with the fixed location of the new particle
                OverlapCount = 0

                if –NewX - MLxReal(ChainStartNo) == 0) then
                        MLxReal(MolNo-d) = NewX
                        CSCy =–CSCy - RadT
                        MLyReal(MolNo-d) = CSCy
                else
                        MLxReal(MolNo-d) = RadT /
(sqrt((ChainGrad**2) + 1))
                        MLyReal(MolNo-d) = (ChainGrad *
MLxReal(MolNo-d))
```

```
                        if (NewX < MLxReal(ChainStartNo)) then
                            CSCy = CScy + MLyReal(MolNo-d)
                            CSCx = CScx + MLxReal(MolNo-d)
                            MLxReal(MolNo-d) = CSCx
                            MLyReal(MolNo-d) = CSCy
                        elseif (NewX > MLxReal(ChainStartNo))
then
                            CSCy =–CScy - MLyReal(MolNo-d)
                            CSCx =–CScx - MLxReal(MolNo-d)
                            MLxReal(MolNo-d) = CSCx
                            MLyReal(MolNo-d) = CSCy
                        end if
                    end if
                end do

            elseif (ChainTopHitCheck .eqv. .TRUE.) then ! If the particle is over
the top of the box it is moved to be resting against it
                OverTopCount = OverTopCount + 1
                if (OverlapCount > 2500) then
                    MolNo = –olN– - a - 1
                    GO TO 10
                end if
                if (OverTopCount > 2500) then
                    MolNo = –olN– - a - 1
                    GO TO 10
                end if
                NewY = GridSize-RadT
                Dist = (a+1)*RadT

                NewX = -(MLyReal(MolNo-a-1)**2)+(2*MLyReal(MolNo-a-
1)*NewY)
                NewX = NewX+(Dist**2)-(NewY**2)
                NewX = sqrt(NewX)

                if (NewX /= NewX) then
                    MolNo = –olN– - a - 1
                    GO TO 10
                end if

                if (–empx - (MLxReal(MolNo-a-1) + NewX) < –empx -
(MLxReal(MolNo–a-1) - NewX)) then
                    NewX = MLxReal(MolNo-a-1) + NewX
                else
                    NewX = MLxReal(MolNo–a-1) - NewX
                end if

                do b = 1, –olNo - a-2
                    Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-
NewY)**2)

                    Dist = sqrt(Dist)
                    if (Dist < ((RadT)+(Mlr(b)))) then
                        ChainHitCheck = .TRUE.
                        PartHit = b
                        OverlapCount = OverlapCount + 1
                        GO TO 99
                    end if
```

```
                end do

                ChainGrad = –NewY - MLyReal(ChainStartNo)) / –NewX - MLxReal(ChainStartNo))

                EquC =–NewY - (ChainGrad * NewX)
                CSCy = NewY
                CSCx = NewX

                do d = 1, a
                        OverlapCount = 0
                        OverTopCount = 0

                        if –NewX - MLxReal(ChainStartNo) == 0) then
                                MLxReal(MolNo-d) = NewX
                                CSCy =–CSCy - RadT
                                MLyReal(MolNo-d) = CSCy
                        else
                                MLxReal(MolNo-d) = RadT / (sqrt((ChainGrad**2) + 1))
                                MLyReal(MolNo-d) = (ChainGrad * MLxReal(MolNo-d))

                                if (NewX < MLxReal(ChainStartNo)) then
                                        CSCy = CScy + MLyReal(MolNo-d)
                                        CSCx = CScx + MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
                                elseif (NewX > MLxReal(ChainStartNo)) then
                                        CSCy =–CScy - MLyReal(MolNo-d)
                                        CSCx =–CScx - MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
                                end if
                        end if

                end do
                RoofCount = RoofCount + 1
        end if

        do m = –olNo - a-1, MolNo-1
                do n = 1, –olNo - a  - 2
                        Dist = ((MLxReal(n)-MLxReal(m))**2)+((MLyReal(n)-MLyReal(m))**2)
                        Dist = sqrt(Dist)
                        if (Dist+0.1 < (MLr(m)+MLr(n))) then
                                MolNo = –olN– - a - 1
                                if ((AdjCount < 75) .and. (AdjCount > -75)) then
                                        AdjCount = Adj–ount - LR
                                        HMAdju = .TRUE.
                                        GO TO 19
                                elseif (AdjCount < -75 .and. Downwards .eqv. .TRUE.) then
                                        Downwards = .FALSE.
                                        AdjCount = 0
                                        LR = 1
```

```
                                        GO TO 19
                                elseif (AdjCount > 75 .and. Upwards .eqv.
.TRUE.) then
                                        Downwards = .TRUE.
                                        AdjCount = 0
                                        LR = -1
                                        GO TO 19
                                else
                                        GO TO 10
                                end if
                        end if
                end do
        end do

        if (NewY > GridSize-RadT) then
                RoofCount = RoofCount + 1
                MolNo = –olN– - a - 1
                GO TO 10
        elseif ((NewX > GridSize-RadT) .or. (NewX < RadT)) then
                MolNo = –olN– - a - 1
                GO TO 10
        end if

        MLxReal(MolNo) = NewX ! Saves the particle location and loops to
the next particle in the chain to be added if there are any left
        MLyReal(MolNo) = NewY
        MLr(MolNo) = RadT
        FullCount = 0
        MolNo = MolNo + 1
end do

if ((MLxReal(MolNo-1) <= RadT+0.2) .or. (MLxReal(MolNo-1) >= GridSize-
RadT-0.2)) then
        GO TO 10
end if

if (MOD(ChainLength,2) == 0) then ! Determines the centre point of the chain
for determining balance
        MiddlePartA = –olNo - ((ChainLength/2)+1)
        MiddlePartB = –olNo - ((ChainLength/2))
        MiddlePoint = ((MLxReal(MiddleP–rta) - MLxReal(MiddlePartB))/2) +
MLxReal(MiddlePartB)
elseif (MOD(ChainLength,2) == 1) then
        MiddlePart = –olNo - ((ChainLength/2)+1)
        MiddlePoint = MLxReal(MiddlePart)
end if

do m = MolNo-ChainLength, MolNo-1 ! Determines furthest point of contact
along the chain
        Dist = ((MLxReal(m)-MLxReal(ContPoint))**2)+((MLyReal(m)-
MLyReal(ContPoint))**2)
        Dist = sqrt(Dist)
        if (Dist <= (MLr(m)+MLr(ContPoint)+0.1)) then
                ContDist = Dist
                ContPointB = m
        end if
end do
```

```
        ContSpot = 0
        OuterCPLoopA: do m = MolNo-1, MolNo-ChainLength, -1
            do n = 1, –olN– - a - 2
                Dist = ((MLxReal(m)-MLxReal(n))**2)+((MLyReal(m)-MLyReal(n))**2)
                Dist = sqrt(Dist)
                if (Dist < MLr(m)+MLr(n)+0.1) then
                    if (MLxReal(MolNo-1) > MLxReal(ChainStartNo)) then
                        ContSpotTemp = ((MLxRe–l(n) - MLxReal(m))/2) + MLxReal(m)
                        if ((ContSpot == 0) .or. (ContSpotTemp > ContSpot)) then
                            ContSpot = ContSpotTemp
                            ContPoint = n
                            ContPointB = m
                        end if
                    else
                        ContSpotTemp = ((MLxRe–l(n) - MLxReal(m))/2) + MLxReal(m)
                        if ((ContSpot == 0) .or. (ContSpotTemp < ContSpot)) then
                            ContSpot = ContSpotTemp
                            ContPoint = n
                            ContPointB = m
                        end if
                    end if
                end if
            end do
        end do OuterCPLoopA

        ContSpot = ((MLxReal(ContP–int) - MLxReal(ContPointB))/2) + MLxReal(ContPointB)

        if ((MLxReal(MolNo-1) > MLxReal(ChainStartNo)) .and. (MiddlePoint > ContSpot)) then ! Checks that contact points on the chain are in the correct positions for chain balance
21          CONTINUE
            x = MLxReal(MolNo-1)
            FallPoint = MLyReal(MolNo-1)
            MolNo = –olNo - ChainLength
            Fell = .TRUE.
            if (FallCount < 10) then
                FallCount = FallCount + 1
                GO TO 40
            else
                if ((AdjCount < 75) .and. (AdjCount > -75)) then
                    AdjCount = Adj–ount - LR
                    HMAdju = .TRUE.
                    GO TO 19
                elseif (AdjCount < -75 .and. Downwards .eqv. .TRUE.) then
                    Downwards = .FALSE.
                    AdjCount = 0
                    LR = 1
                    GO TO 19
                elseif (AdjCount > 75 .and. Upwards .eqv. .TRUE.) then
```

```
                              Downwards = .TRUE.
                              AdjCount = 0
                              LR = -1
                              GO TO 19
                        else
                              GO TO 10
                        end if
                  end if
            elseif ((MLxReal(MolNo-1) < MLxReal(ChainStartNo)) .and. (MiddlePoint <
ContSpot)) then
22                CONTINUE
                  x = MLxReal(MolNo-1)
                  FallPoint = MLyReal(MolNo-1)
                  MolNo = –olNo - ChainLength
                  Fell = .TRUE.
                  if (FallCount < 10) then
                        FallCount = FallCount + 1
                        GO TO 40
                  else
                        if ((AdjCount < 75) .and. (AdjCount > -75)) then
                              AdjCount = Adj–ount - LR
                              HMAdju = .TRUE.
                              GO TO 19
                        elseif (AdjCount < -75 .and. Downwards .eqv. .TRUE.) then
                              Downwards = .FALSE.
                              AdjCount = 0
                              LR = 1
                              GO TO 19
                        elseif (AdjCount > 75 .and. Upwards .eqv. .TRUE.) then
                              Downwards = .TRUE.
                              AdjCount = 0
                              LR = -1
                              GO TO 19
                        else
                              GO TO 10
                        end if
                  end if
            end if

            ContSpotL = 0
            ContSpotR = 0
            ContSpotUy = 0
            ContSpotUx = 0
            ContSpotUn = 0

            do m = MolNo-1, MolNo-ChainLength, -1
                  do n = 1, –olN– - a - 2
                        Dist = ((MLxReal(m)-MLxReal(n))**2)+((MLyReal(m)-
MLyReal(n))**2)
                        Dist = sqrt(Dist)
                        if (Dist < MLr(m)+MLr(n)+0.1) then
                              ContPoint = n
                              ContSpotTemp = ((MLxReal(ContP–int) -
MLxReal(m))/2) + MLxReal(m)
                              if ((ContSpotL == 0) .or. (ContSpotTemp <
ContSpotL)) then
                                    ContSpotL = ContSpotTemp
```

```
                                        end if
                                        if ((ContSpotR == 0) .or. (ContSpotTemp >
ContSpotR)) then
                                                ContSpotR = ContSpotTemp
                                        end if
                                        ContSpotTempUy = ((MLyReal(ContP–int) -
MLyReal(m))/2) + MLyReal(m)
                                        if (m /= –olNo - ChainLength) then
                                                if ((ContSpotUy == 0) .or. (ContSpotTempUy
> ContSpotUy)) then
                                                        ContSpotUy = ContSpotTempUy
                                                        ContSpotUx = ContSpotTemp
                                                        ContSpotUn = m
                                                end if
                                        end if
                                end if
                        end do
                end do

                do m = MolNo-1, MolNo-ChainLength+1, -1
                        do n = 1, –olN– - a - 2
                                Dist = ((MLxReal(m)-MLxReal(n))**2)+((MLyReal(m)-
MLyReal(n))**2)
                                Dist = sqrt(Dist)
                                if (Dist < MLr(m)+MLr(n)+0.1) then
                                        ContSpotTemp = ((MLxRe–l(n) - MLxReal(m))/2) +
MLxReal(m)

                                        if ((MLxReal(MolNo-1) > MLxReal(MolNo-
ChainLength)) .and. (ContSpotTemp > MLxReal(m))) then
                                                SideBal = .TRUE.
                                        elseif ((MLxReal(MolNo-1) < MLxReal(MolNo-
ChainLength)) .and. (ContSpotTemp < MLxReal(m))) then
                                                SideBal = .TRUE.
                                        end if
                                end if
                        end do
                end do


                if (MLxReal(MolNo-1) > MLxReal(MolNo-ChainLength)) then ! If the particle
is not balancing correctly, the algorithm goes back to the beginning, however the initial falling
point is moved to the end of this chain
                        if ((ContSpotL > MLxReal(MolNo-ChainLength)) .or. (SideBal .eqv.
.FALSE.)) then
                                if ((MLxReal(MolNo-ChainLength).GT.MLr(MolNo-
ChainLength)+0.2).and.(MLyReal(MolNo-ChainLength)/=MLyReal(MolNo-1))) then
                                        NLCount = NLCount + 1
                                        OverlapCount = 0
                                        if (LRNo == 1) then
                                                LRNo = 2
                                        else
                                                LRNo = 1
                                        end if
                                        Tilt = .TRUE.
                                        MolNo = MolNo - a
                                        GO TO 14
```

```
                                        end if
                                end if
                        elseif (MLxReal(MolNo-1) < MLxReal(MolNo-ChainLength)) then
                                if ((ContSpotR < MLxReal(MolNo-ChainLength)) .or. (SideBal .eqv.
.FALSE.)) then
                                        if      ((MLxReal(MolNo-ChainLength)/=GridSize-MLr(MolNo-
ChainLength)).and.(MLyReal(MolNo-ChainLength)/=MLyReal(MolNo-1))) then
                                                NLCount = NLCount + 1
                                                OverlapCount = 0
                                                if (LRNo == 1) then
                                                        LRNo = 2
                                                else
                                                        LRNo = 1
                                                end if
                                                Tilt = .TRUE.
                                                MolNo = MolNo - a
                                                GO TO 14
                                        end if
                                end if
                        end if

                        if (MLyReal(MolNo-ChainLength) > MLyReal(MolNo-1)) then
                                if (MLxReal(MolNo-1) > MLxReal(MolNo-ChainLength)) then
                                        GO TO 21
                                elseif (MLxReal(MolNo-1) < MLxReal(MolNo-ChainLength)) then
                                        GO TO 22
                                end if
                        end if

10                      CONTINUE

                        if ((MinFallCount > FallCount) .or. (MinFallCount == 0)) then
                                MinFallCount = FallCount
                        end if

                else
                        Full = 1
                end if

                end

                subroutine PointSafe ! Determines if the falling particle has impacted yet
                use allSubs ! Loads the variables from the module
                integer a, b, CoordX, CoordY
                real Dist
                character t

                Hit = .FALSE.
                FullCheck = .FALSE.

                ! Checks the distance between the current falling particle location and previously
placed partice to determine if it has impacted

                cloop: do a = 1, MolNo - 1
                        Dist = ((MLxReal(a)-x)**2)+((MLyReal(a)-y)**2)
                        Dist = sqrt(Dist)
                        if (Dist <= ((RadT)+(Mlr(a)))) then
```

```
                    Hit = .TRUE.
                    exit cloop
                end if
        end do cloop

        ! If the impact is above the top of the box, a counter is incremented to show the box
may be full
        if ((Hit .eqv. .TRUE.) .and. (y >= (GridSize - RadT))) then
                Hit = .FALSE.
                FullCount = FullCount + 1
                RoofCount = RoofCount + 1
        end if

        if (RoofCount >= 1000) then
                FullCheck = .TRUE.
        end if

        if (FullCount == 2500000) then
                Full = 1
        end if

        end



        subroutine AddChain() ! This subroutine performs the exact same function as the main
particle chain adding code, however is used for the chains placed along the base layer
        use allsubs
        integer count2, Spot, Height, DoubRad, RowRad, a,b,c, RadIn, m, n, TempX, TempY
        real MidWay, Dist
        character t, FileName*15
        integer SavIncremX, SavIncremY, SafeLocCount, LR, SavOneX, SavOneY, SavDist,
SavOnePart, LRNo, OneCount, RealPos1, RealPos2
        integer TempRealPos1, TempRealPos2, ChainOneCount
        logical SafeLocFound, Go, ResetCheck
        real TempXa, TempXb, TempYa, TempYb, DistAB, DistBC, DistAC, AngleA, AngleB,
AngleFin, GradFin, HelpDist
        real FDistA, FDistB, FDistC, FDistD
        real xDiff, yDiff, Pi, FinalSavX, FinalSavY
        integer FinalSavLong, FinalSavTall, checktime, Balanced
        integer Balances, Touches
        dimension Balances(10)
        real DistFac, RadScale
        integer NewPos, TRP1Swap, TRP2Swap
        integer FinalPart, SideCount
        dimension FinalPart(2,3)
        real PartCoords, stochDists, sumDist, dx, stochxnew, newCoords, stochynew
        integer ibad
        dimension PartCoords(2,2), stochDists(2), newCoords(2), dx(2), stochxnew(2)
        real RX, NewX, NewY
        logical ChainAdd
        real OverDist
        integer OverDistNo
        logical NotBal, FirstBal
        dimension BalCheckNo(10000), BalCombi(100000,2)
        integer BalCheckCount, BalCount, BalCheckNo, BalCombi, BalAttempt
        integer OverlapCount
```

```
integer ChainStartNo, ChainStartCont, d
real ChainGrad, EquC
logical ChainHitCheck, ChainEdgeHitCheck, ChainTopHitCheck
logical TopCase
integer OverTopCount, PartHit

Hit = .FALSE.
ResetCheck = .FALSE.
checktime = 1
Balanced = 0
Touches = 0
Balances = 0

FinalPart = 999999
OverDist = 0
OverDistNo = 0

OverlapCount = 0

TRP1Swap = 0
TRP2Swap = 0

ChainAdd = .FALSE.
ChainOneCount = 1

NotBal = .FALSE.
FirstBal = .TRUE.
BalCheckNo = 0
BalCombi = 0
BalCheckCount = 0
BalCount = 0
BalCheckNo = 0
BalCombi = 0
BalAttempt = 0

RadScale = 0
TempRealPos1 = 0
TempRealPos2 = 0
DistFac = 0
FDistA = 0
FDistB = 0
FDistC = 0
FDistD = 0
SavDist = 0
SavOneX = 0
SavOneY = 0
TempXa = 0
TempXb = 0
TempYa = 0
TempYb = 0
DistAB = 0
DistBC = 0
DistAC = 0
AngleA = 0
AngleB = 0
AngleFin = 0
GradFin = 0
```

```
        xDiff = 0
        yDiff = 0
        Pi = 3.141596535
        NewPos = 0

        FinalDists = 0

        PartCoords = 0
        stochDists = 0
        sumDist = 0
        dx = 0
        stochxnew = 0
        newCoords = 0
        stochynew = 0

        RA = ''
        OverlapCount = 0

        do a = 1, MolNo - 1
                MLxCor = MLxReal(a)
                MLyCor = MLyReal(a)
                RadIn = MLr(a)
                DoubRad = (RadIn+RadT)+1

                if (a /= MolNo-1) then
                        do Height = 0, DoubRad
                                MidWay = DoubRad**2 - Height**2
                                RowRad = abs(sqrt(Midway))
                                do count2 = -rowrad+1, rowrad-1
                                        if
((MLyCor+count2<=GridSize).and.(MLxCor+Height<=GridSize-
RadT).and.(MLyCor+count2>=RadT).and.(MLxCor+Height>=RadT))then
                                                RA(MLyCor+count2, MLxCor+Height) = '-'
                                        end if
                                        if      ((MLyCor+count2<=GridSize).and.(MLxCor-
Height<=GridSize-RadT).and.(MLyCor+count2>=RadT).and.(MLxCor-Height>=RadT))then
                                                RA(MLyCor+count2, MLxCor-Height) = '-'
                                        end if
                                end do
                        end do

                        do Height = -RadIn, RadIn
                                do count2 = MlyCor-1, 1, -1
                                        if
((count2<=GridSize).and.(MLxCor+Height<=GridSize-
RadT).and.(count2>=RadT).and.(MLxCor+Height>=RadT))then
                                                RA(count2, MLxCor+Height) = '-'
                                        end if
                                end do
                        end do
                end if

                if (a == MolNo-1) then
                        do Height = -RadIn, 0
                                MidWay = RadIn**2 - Height**2
                                RowRad = abs(sqrt(Midway))
```

```
                    if
((MLyCor+RowRad<=GridSize).and.(MLxCor+height<=GridSize-
RadT).and.(MLxCor+height>=RadT).and.(MLyCor+RowRad>=RadT))then
                        if (RA(MLyCor+RowRad, MLxCor+Height) /= '-') then
                            RA(MLyCor+RowRad, MLxCor+Height) = '1'
                        end if
                    end if
                    if                                          ((MLyCor-
RowRad>=RadT).and.(MLxCor+height<=GridSize-
RadT).and.(MLxCor+height>=RadT).and.(MLyCor-RowRad<=GridSize))then
                        if (RA(MLyCor-RowRad, MLxCor+Height) /= '-') then
                            RA(MLyCor-RowRad, MLxCor+Height) = '1'
                        end if
                    end if
                    if              ((MLyCor+RowRad<=GridSize).and.(MLxCor-
Height<=GridSize-RadT).and.(MLxCor-Height>=RadT).and.(MLyCor+RowRad>=RadT))then
                        if (RA(MLyCor+RowRad, MLxCor-Height) /= '-') then
                            RA(MLyCor+RowRad, MLxCor-Height) = '1'
                        end if
                    end if
                    if                  ((MLyCor-RowRad>=RadT).and.(MLxCor-
height<=GridSize-RadT).and.(MLxCor-height>=RadT).and.(MLyCor-
RowRad<=GridSize))then
                        if (RA(MLyCor-RowRad, MLxCor-Height) /= '-') then
                            RA(MLyCor-RowRad, MLxCor-Height) = '1'
                        end if
                    end if
                end do
            end if
        end do

        do a = 1, GridSize
            do b = 1, GridSize
                if (RA(b,a) == '1') then
                    ChainOnes(ChainOneCount, 1) = b
                    ChainOnes(ChainOneCount, 2) = a
                    ChainOneCount = ChainOneCount + 1
                end if
            end do
        end do

        call random_number(RX)
        LRNo = 1 + floor(2*RX)
        if (LRNo == 1) then
            TempX = ChainOnes(1,2)
            TempY = ChainOnes(1,1)
        elseif (LRNo == 2) then
            TempX = ChainOnes(ChainOneCount-1,2)
            TempY = ChainOnes(ChainOneCount-1,1)
        end if

        FinalPart = 99999
        FinalPart(1,1) = MolNo-1
        Dist = ((MLxReal(a)-TempX)**2)+((MLyReal(a)-TempY)**2)
        Dist = sqrt(Dist) - MLr(a)
        FinalPart(1,2) = Dist
        FinalPart(1,3) = Dist + MLr(a)
```

```
        do a = 1, MolNo-2
                if (MLyReal(a) < TempY+1) then
                        Dist = ((MLxReal(a)-TempX)**2)+((MLyReal(a)-TempY)**2)
                        Dist = sqrt(Dist) - RadT - MLr(a)
                        if (Dist <= FinalPart(2,2)) then
                                FinalPart(2,1) = a
                                FinalPart(2,2) = Dist
                                FinalPart(2,3) = Dist + RadT + MLr(a)
                        end if
                end if
        end do

60              CONTINUE

        PartCoords(1,1) = MLxReal(FinalPart(1,1))
        PartCoords(1,2) = MLyReal(FinalPart(1,1))
        PartCoords(2,1) = MLxReal(FinalPart(2,1))
        PartCoords(2,2) = MLyReal(FinalPart(2,1))

        if ((TempX == RadT) .or. (TempX == GridSize-RadT)) then
                NewX = TempX
                Dist = MLr(FinalPart(1,1))

                NewY = -(MLxReal(FinalPart(1,1))**2)+(2*MLxReal(FinalPart(1,1))*NewX)
                NewY = NewY+(Dist**2)-(NewX**2)
                NewY = sqrt(NewY)

                if (NewY /= NewY) then
                        MolNo = MolNo - 1
                        GO TO 10
                end if

                if (TempY - (MLyReal(FinalPart(1,1)) + NewY) < TempY -
(MLyReal(FinalPart(1,1)) - NewY)) then
                        NewY = MLyReal(FinalPart(1,1)) + NewY
                else
                        NewY = MLyReal(FinalPart(1,1)) - NewY
                end if

        else
                newCoords(1) = TempX
                newCoords(2) = TempY

                stochDists(1) = FinalPart(1,3)
                stochDists(2) = FinalPart(2,3)

                sumDist = stochDists(1) + stochDists(2)

                dx(1) = 10 * RadLarge
                dx(2) = 10 * RadLarge

                do a = 1, 10
                        do b = 1, 2
                                dx(b) = dx(b) / 10
                        end do
                        do c = 1, 500
```

```
                    call random_number(RX)
                    stochxnew(2) = newCoords(2) + dx(2) * (2*RX-1)
                    call random_number(RX)
                    stochxnew(1) = newCoords(1) + dx(1) * (2*RX-1)

                    ibad = 0

                    stochDists(1) = ((PartCoords(1,1)-
stochxnew(1))**2)+((PartCoords(1,2)-stochxnew(2))**2)
                    stochDists(1) = sqrt(stochDists(1)) - MLr(FinalPart(1,1))
                    if (stochDists(1) < 0) then
                         ibad = 1
                    end if

                    stochDists(2) = ((PartCoords(2,1)-
stochxnew(1))**2)+((PartCoords(2,2)-stochxnew(2))**2)
                    stochDists(2) = sqrt(stochDists(2)) - RadT -
MLr(FinalPart(2,1))
                    if (stochDists(2) < 0) then
                         ibad = 1
                    end if

                    stochynew = stochDists(1) + stochDists(2)
                    if ((stochynew < sumDist) .and. (ibad == 0)) then
                         do b = 1, 2
                              newCoords(b) = stochxnew(b)
                         end do
                         sumDist = stochynew
                    end if
               end do
          end do

          NewX = newCoords(1)
          NewY = newCoords(2)
     end if

     if ((NewX > GridSize-RadT) .or. (NewX < RadT) .or. (NewY > GridSize-RadT) .or.
(NewY < RadT)) then
          MolNo = MolNo - 1
          GO TO 10
     end if

     do a = 1, MolNo-2
          Dist = ((MLxReal(a)-NewX)**2)+((MLyReal(a)-NewY)**2)
          Dist = sqrt(Dist)
          if (Dist < MLr(a)+RadT) then
               OverlapCount = OverlapCount + 1
               if (OverlapCount > 2500) then
                    MolNo = MolNo - 1
                    GO TO 10
               end if

               FinalPart(2,1) = a
               GO TO 60

               FullCount = FullCount + 1
               MolNo = MolNo - 1
```

```
                GO TO 10
            end if
        end do

        MLxReal(MolNo) = NewX
        MLyReal(MolNo) = NewY
        MLr(MolNo) = RadT
        FullCount = 0
        MolNo = MolNo + 1

        ChainStartNo = MolNo-2
        ChainStartCont = MolNo-1
        CSCy = MLyReal(ChainStartCont)

        do a = 1, ChainLength - 2
            OverlapCount = 0

            ChainGrad = (MLyReal(ChainStartCont) - MLyReal(ChainStartNo)) /
(MLxReal(ChainStartCont) - MLxReal(ChainStartNo))
            EquC = MLyReal(ChainStartCont) - (ChainGrad * MLxReal(ChainStartCont))
            if (MLxReal(ChainStartCont) - MLxReal(ChainStartNo) == 0) then
                NewX = MLxReal(ChainStartCont)
                CSCy = CSCy + RadT
                NewY = CSCy
            else
                NewX = RadT / (sqrt((ChainGrad**2) + 1))
                NewY = (ChainGrad * NewX)

                if (MLxReal(ChainStartCont) < MLxReal(ChainStartNo)) then
                    NewX = MLxReal(MolNo-1) - NewX
                    NewY = MLyReal(MolNo-1) - NewY
                elseif (MLxReal(ChainStartCont) > MLxReal(ChainStartNo)) then
                    NewX = MLxReal(MolNo-1) + NewX
                    NewY = MLyReal(MolNo-1) + NewY
                end if
            end if

            ChainHitCheck = .FALSE.
            ChainEdgeHitCheck = .FALSE.
            ChainTopHitCheck = .FALSE.

            ChainHitLoop: do b = 1, MolNo - a-2
                Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
                Dist = sqrt(Dist)
                if (Dist < ((RadT)+(Mlr(b)))) then
                    ChainHitCheck = .TRUE.
                    PartHit = b
                end if
            end do ChainHitLoop

99              CONTINUE

            if ((NewX > GridSize-RadT) .or. (NewX < RadT)) then
                ChainEdgeHitCheck = .TRUE.
            elseif (NewY > GridSize-RadT) then
                ChainTopHitCheck = .TRUE.
            end if
```

```
if (ChainHitCheck .eqv. .TRUE.) then
	if (OverlapCount > 2500) then
		MolNo = MolNo - a - 1
		GO TO 10
	end if

	newCoords(1) = NewX
	newCoords(2) = NewY

	stochDists(1) = Dist
	stochDists(2) = ((MLxReal(MolNo-a-1)-NewX)**2)+((MLyReal(MolNo-a-1)-NewY)**2)
	stochDists(2) = sqrt(stochDists(2))

	sumDist = stochDists(1) + stochDists(2)

	dx(1) = 10 * RadLarge
	dx(2) = 10 * RadLarge

	do b = 1, 10
		do c = 1, 2
			dx(c) = dx(c) / 10
		end do
		do c = 1, 500
			call random_number(RX)
			stochxnew(2) = newCoords(2) + dx(2) * (2*RX-1)
			call random_number(RX)
			stochxnew(1) = newCoords(1) + dx(1) * (2*RX-1)

			ibad = 0

			stochDists(1) = ((MLxReal(PartHit)-stochxnew(1))**2)+((MLyReal(PartHit)-stochxnew(2))**2)
			stochDists(1) = sqrt(stochDists(1)) - RadT - MLr(PartHit)
			if (stochDists(1) < 0) then
				ibad = 1
			end if

			stochDists(2) = ((MLxReal(MolNo-a-1)-stochxnew(1))**2)+((MLyReal(MolNo-a-1)-stochxnew(2))**2)
			stochDists(2) = sqrt(stochDists(2)) - ((a+1) * RadT)
			if (stochDists(2) < 0) then
				ibad = 1
			end if

			stochynew = stochDists(1) + stochDists(2)
			if ((stochynew < sumDist) .and. (ibad == 0)) then
				do d = 1, 2
					newCoords(d) = stochxnew(d)
				end do
				sumDist = stochynew
			end if
		end do
	end do

	NewX = newCoords(1)
```

```
                NewY = newCoords(2)

                if (NewY < RadT) then
                        NewY = RadT
                        NewX = MLxReal(MolNo-1) + RadT
                end if

                do b = MolNo - a-2, 1, -1
                        Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
                        Dist = sqrt(Dist)
                        if (Dist < ((RadT)+(Mlr(b)))) then
                                ChainHitCheck = .TRUE.
                                PartHit = b
                                OverlapCount = OverlapCount + 1
                                GO TO 99
                        end if
                end do

                if ((NewX < RadT) .or. (NewX > GridSize - RadT)) then
                        ChainHitCheck = .FALSE.
                        ChainEdgeHitCheck = .TRUE.
                        GO TO 99
                end if
                if (NewY > GridSize - RadT) then
                        ChainHitCheck = .FALSE.
                        ChainTopHitCheck = .TRUE.
                        GO TO 99
                end if

                ChainGrad = (NewY - MLyReal(ChainStartNo)) / (NewX -
MLxReal(ChainStartNo))
                EquC = NewY - (ChainGrad * NewX)
                CSCy = NewY
                CSCx = NewX

                do d = 1, a
                        OverlapCount = 0

                        if (NewX - MLxReal(ChainStartNo) == 0) then
                                MLxReal(MolNo-d) = NewX
                                MLyReal(MolNo-d) = NewY-RadT
                        else
                                MLxReal(MolNo-d) = RadT / (sqrt((ChainGrad**2) +
1))
                                MLyReal(MolNo-d) = (ChainGrad * MLxReal(MolNo-
d))

                                if (NewX < MLxReal(ChainStartNo)) then
                                        CSCy = CSCy + MLyReal(MolNo-d)
                                        CSCx = CSCx + MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
                                elseif (NewX > MLxReal(ChainStartNo)) then
                                        CSCy = CSCy - MLyReal(MolNo-d)
                                        CSCx = CSCx - MLxReal(MolNo-d)
                                        MLxReal(MolNo-d) = CSCx
                                        MLyReal(MolNo-d) = CSCy
```

```
                    end if
                end if

            end do
        end if

        if (ChainEdgeHitCheck .eqv. .TRUE.) then
            if (OverlapCount > 2500) then
                MolNo = MolNo - a - 1
                GO TO 10
            end if
            if (NewX > GridSize-RadT) then
                NewX = GridSize-RadT
            elseif (NewX < RadT) then
                NewX = RadT
            end if
            Dist = (a+1)*RadT

            NewY = -(MLxReal(MolNo-a-1)**2)+(2*MLxReal(MolNo-a-1)*NewX)
            NewY = NewY+(Dist**2)-(NewX**2)
            NewY = sqrt(NewY)

            if (NewY /= NewY) then
                MolNo = MolNo - a - 1
                GO TO 10
            end if

            if (TempY - (MLyReal(MolNo-a-1) + NewY) < TempY -
(MLyReal(MolNo-a-1) - NewY)) then
                NewY = MLyReal(MolNo-a-1) + NewY
            else
                NewY = MLyReal(MolNo-a-1) - NewY
            end if

            do b = 1, MolNo - a-2
                Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
                Dist = sqrt(Dist)
                if (Dist < ((RadT)+(Mlr(b)))) then
                    ChainHitCheck = .TRUE.
                    PartHit = b
                    OverlapCount = OverlapCount + 1
                    GO TO 99
                end if
            end do

            ChainGrad = (NewY - MLyReal(ChainStartNo)) / (NewX -
MLxReal(ChainStartNo))
            EquC = NewY - (ChainGrad * NewX)
            CSCy = NewY
            CSCx = NewX

            do d = 1, a
                OverlapCount = 0

                if (NewX - MLxReal(ChainStartNo) == 0) then
                    MLxReal(MolNo-d) = NewX
                    MLyReal(MolNo-d) = NewY-RadT
```

```
				else
					MLxReal(MolNo-d) = RadT / (sqrt((ChainGrad**2) +
1))
					MLyReal(MolNo-d) = (ChainGrad * MLxReal(MolNo-
d))

					if (NewX < MLxReal(ChainStartNo)) then
						CSCy = CSCy + MLyReal(MolNo-d)
						CSCx = CSCx + MLxReal(MolNo-d)
						MLxReal(MolNo-d) = CSCx
						MLyReal(MolNo-d) = CSCy
					elseif (NewX > MLxReal(ChainStartNo)) then
						CSCy = CSCy - MLyReal(MolNo-d)
						CSCx = CSCx - MLxReal(MolNo-d)
						MLxReal(MolNo-d) = CSCx
						MLyReal(MolNo-d) = CSCy
					end if
				end if

			end do

		elseif (ChainTopHitCheck .eqv. .TRUE.) then
			OverTopCount = OverTopCount + 1
			if (OverlapCount > 2500) then
				MolNo = MolNo - a - 1
				GO TO 10
			end if
			if (OverTopCount > 2500) then
				MolNo = MolNo - 2
				GO TO 10
			end if
			NewY = GridSize-RadT
			Dist = (a+1)*RadT

			NewX = -(MLyReal(MolNo-a-1)**2)+(2*MLyReal(MolNo-a-1)*NewY)
			NewX = NewX+(Dist**2)-(NewY**2)
			NewX = sqrt(NewX)

			if (NewX /= NewX) then
				MolNo = MolNo - a - 1
				GO TO 10
			end if

			if (TempX - (MLxReal(MolNo-a-1) + NewX) < TempX -
(MLxReal(MolNo-a-1) - NewX)) then
				NewX = MLxReal(MolNo-a-1) + NewX
			else
				NewX = MLxReal(MolNo-a-1) - NewX
			end if

			do b = 1, MolNo - a-2
				Dist = ((MLxReal(b)-NewX)**2)+((MLyReal(b)-NewY)**2)
				Dist = sqrt(Dist)
				if (Dist < ((RadT)+(Mlr(b)))) then
					ChainHitCheck = .TRUE.
					PartHit = b
					OverlapCount = OverlapCount + 1
```

```
                        GO TO 99
                    end if
                end do

                ChainGrad = (NewY - MLyReal(ChainStartNo)) / (NewX -
MLxReal(ChainStartNo))
                EquC = NewY - (ChainGrad * NewX)
                CSCy = NewY

                do d = 1, a
                    OverlapCount = 0
                    OverTopCount = 0

                    if (NewX - MLxReal(ChainStartNo) == 0) then
                        MLxReal(MolNo-d) = NewX
                        CSCy = CSCy - RadT
                        MLyReal(MolNo-d) = CSCy
                    else
                        MLxReal(MolNo-d) = RadT / (sqrt((ChainGrad**2) +
1))
                        MLyReal(MolNo-d) = (ChainGrad * MLxReal(MolNo-
d))

                        if (NewX < MLxReal(ChainStartNo)) then
                            CSCy = CSCy + MLyReal(MolNo-d)
                            CSCx = CSCx + MLxReal(MolNo-d)
                            MLxReal(MolNo-d) = CSCx
                            MLyReal(MolNo-d) = CSCy
                        elseif (NewX > MLxReal(ChainStartNo)) then
                            CSCy = CSCy - MLyReal(MolNo-d)
                            CSCx = CSCx - MLxReal(MolNo-d)
                            MLxReal(MolNo-d) = CSCx
                            MLyReal(MolNo-d) = CSCy
                        end if
                    end if

                end do

            end if

            MLxReal(MolNo) = NewX
            MLyReal(MolNo) = NewY
            MLr(MolNo) = RadT
            FullCount = 0
            MolNo = MolNo + 1
        end do

10      CONTINUE

        end
```

Appendix 5: RSA Algorithm

This appendix contains the algorithm used to create the random sequential adsorption image used in Figure 4.3.

```
program packedbed
! Sets up initial variables
integer MolNo, RadLarge, RadSmall, BoxSize, GridSize, Rads, count, SN,
integer MLr
real MLxReal, MLyReal
dimension MLr(10000), Rads(10), MLxReal(10000), MLyReal(10000)
integer x, y, RadT

character t, FileName*15, FileID*3
integer ProgCount, PCId, count3
dimension FileID(1000)
real Dist
integer count2
logical Impact

t = 'y'

if (t == 'y') then
        Rads(1) = 10 ! Inputs the radius to be present in the system
        SN = 1

        RadLarge = 0
        RadSmall = 0

        do count = 1, SN
                if (RadLarge < Rads(count)) then
                        RadLarge = Rads(count)
                end if
                if (RadSmall > Rads(count) .or. RadSmall == 0) then
                        RadSmall = Rads(count)
                end if
        end do

        RadLarge = 10  ! Determines the largest and smallest radius of the entered radii
        RadSmall = 10

        BoxSize = (RadLarge*6)
        GridSize = BoxSize*5

        do ProgCount = 1, 50
                write(FileID(ProgCount), '(i0)') ProgCount
        end do

        do ProgCount = 1, 50    ! Loops for the number of systems to be created

                MLxReal = 0
                MLyReal = 0
                MLr = 0
```

```
        MolNo = 1
        FileName = ''

        call random_seed()

        do while (count < 10000000)

                Impact = .FALSE.

                ! Finds a random x and y coordinate and radius
                call random_number(RX)
                count2 = 1 + floor(SN*RX)
                RadT = Rads(count2)

                call random_number(RX)
                count2 = 1 + floor((GridSize-(2*RadT))*RX)
                x = count2+RadT

                call random_number(RX)
                count2 = 1 + floor((GridSize-(2*RadT))*RX)
                y = count2+RadT

                ! Confirms that the chosen location does not overlap with a previously placed particle
                if (MolNo > 1) then
                        hitloop: do count3 = 1, MolNo - 1
                                Dist = ((MLxReal(count3)-x)**2)+((MLyReal(count3)-y)**2)
                                Dist = sqrt(Dist)
                                if (Dist <= (RadT+Mlr(count3))) then
                                        count = count + 1
                                        Impact = .TRUE.
                                        exit hitloop
                                end if
                        end do hitloop
                        if (Impact .eqv. .FALSE.) then    ! If there is no overlap, the location is saved, if there is an overlap the location is not saved and ignored.
                                count = 0
                                MLxReal(MolNo) = x
                                MLyReal(MolNo) = y
                                MLr(MolNo) = RadT
                                MolNo = MolNo + 1
                        end if
                else
                        MLxReal(MolNo) = x
                        MLyReal(MolNo) = y
                        MLr(MolNo) = RadT
                        MolNo = MolNo + 1
                end if
        end do

        t = 'y'
        if (t == 'y') then  ! The particle locations are saved to a file, and the next system is started
                FileName = ''
                FileName = trim(adjustl(FileID(ProgCount))) // '.csv'
                open(1, file = FileName, status = 'new')
```

```
                    do y = 1, MolNo-1
                        write(1,*) MLxReal(y), ',' , MLyReal(y), ',' , MLr(y)
                    end do
                    close(1)
                end if
        end do
end if

end program
```

Appendix 6: MatLab Code Used

This appendix contains the code used, to obtain data and visualise created systems, within MatLab.

Visualising 2D System Beds

```
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
    if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
        load ([num2str(x) '.csv']) % Loading the respective file
        box on % Turns on an outline so the edges of the box are in the image output
        th = 0:pi/50:2*pi; % Gives the variable th values in the range 0 to 2π
        L = length(evalin('base',sprintf('X%d',x))) % Sets L to the number of particles in the system
        figure(x) % Creates a figure for system x
        for q = 1:L % Loops through each particle within the system
            xunit = evalin('base',sprintf('X%d(q,3)',x)) * cos(th) + evalin('base',sprintf('X%d(q,1)',x)); % Calculates the x positions on the particle circumference
            yunit = evalin('base',sprintf('X%d(q,3)',x)) * sin(th) + evalin('base',sprintf('X%d(q,2)',x)); % Calculates the y positions on the particle circumference
            h = plot(xunit, yunit, 'k'); % Plots those positions onto the figure
            hold on % Keeps the old figure when adding a new particle to it
            xlim([0 600]); % Sets the x-axis limits
            ylim([0 600]); % Sets the y-axis limits
        end
    else
        l = x % Outputs the file number if the file does not exist for error checking
    end
end
```

Determining 2D Packing Fractions

```
PartFrac = 0 % Stores the packing fractions for all the systems
PartSum = 0 % Stores the total of all of the packing fractions for calculating the average
PartAvg = 0 % Stores the average of the packing fractions across all the systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
    PartTot = 0; % Stores the total area of Particles within the currently analysed system
    if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
        load ([num2str(x) '.csv']) % Loading the respective file
        L = length(evalin('base',sprintf('X%d',x))); % Sets L to the number of particles in the system
        for q = 1:L % Loops through each particle within the system
            PartTot = ((evalin('base',sprintf('X%d(q,3)',x)) ^ 2) * pi) + PartTot; % Calculates the area of the particle and adds it to the current total area
        end
        PartFrac(x) = PartTot / (300^2); % Calculates the packing fraction of the current system
        PartSum = PartSum + PartFrac(x); % Adds the packing fraction of the current system to the total to be averaged
    end
end
PartAvg = PartSum / 50 % Calculates the packing fraction average of the systems
```

Determining 2D Number of Contacts

```
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
    if isfile(['contacts' num2str(x) '.csv']) % Confirms the file exists in the expected folder
        load (['contacts' num2str(x) '.csv']) % Loading the respective contacts file
        load ([num2str(x) '.csv']) % Loading the respective particle locations file
        L = length(evalin('base',sprintf('contacts%d',x))); % Sets L to the number of
particles in the system
        for q = 1:L
            SmolContNo = 0; % Stores the number of contacts for small
particles
            LorgContNo = 0; % Stores the number of contacts for large particles
            f = 1; % The counter for looping through the arrays
            while evalin('base',sprintf('contacts%d(q,f)',x)) ~= 0 % Checks the
particle has more than zero contacts
                if evalin('base',sprintf('X%d(q,3)',x)) == 10 % Determines if
the particle is of radius 10 (small) or not (large)
                    SmolContNo = SmolContNo + 1; % Increments the
number of contacts for a small particle
                    f = f + 1; % Increments the array counter
                else
                    LorgContNo = LorgContNo + 1; % Increments the
number of contacts for a large particle
                    f = f + 1; % Increments the array counter
                end
            end
            SmolContCount(x,q) = SmolContNo; % Stores the number of
contacts for each small particle across all systems

            LorgContCount(x,q) = LorgContNo; % Stores the number of
contacts for each large particle across all systems
        end
    end
end
```

Determining the Percentage of 2D Systems That Contain a Percolation Chain

```
PercCount = 0 % Stores the number of systems that contain a percolation chain
TotalCount = 0 % Stores the total number of systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
    if isfile(['shapes' num2str(x) '.csv']) % Confirms the file exists in the expected folder
        load (['shapes' num2str(x) '.csv']) % Loading the respective particle shapes
file
        CurShape = evalin('base',sprintf('shapes%d',x)); % Retrieves the chains from
the file
        [numRows,numCols] = size(CurShape); % Determines the size of the array
        TotalCount = TotalCount + 1; % Adds one to the total number of systems
        if numRows == 1 % Determines if the system has no percolation chains
            PercCount = PercCount + 1; % If so adds one to this counter
        end
    end
end
PercFracs = 100 - ((PercCount/TotalCount) * 100) % Determines the percentage of systems
that contain a percolation chain
```

Determining the Number of Percolation Chains in 2D Systems

```
AvgCount = 0 % Stores the average number of percolation chains across all systems
MaxCount = 0 % Stores the maximum number of percolation chain in a system
```

```
MinCount = 999 % Stores the minimum number of percolation chain in a system
RowCount = 0 % Stores the number of percolation chains in each system
TotalCount = 0 % Stores the sum of how many percolation chains across all systems
TotAmount = 0 % Stores the number of systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
	if isfile(['shapes' num2str(x) '.csv']) % Confirms the file exists in the expected folder
		load (['shapes' num2str(x) '.csv']) % Loading the respective particle shapes file
		CurShape = evalin('base',sprintf('shapes%d',x)); % Retrieves the chains from the file
		[numRows,numCols] = size(CurShape); % Determines the size of the array
		TotalCount = TotalCount + (numRows-1); % Adds the number of percolation chains in the current system to the total
		TotAmount = TotAmount + 1; % Adds one to the total number of systems
		RowCount(x) = numRows-1; % Saves number of percolation chain to the array
		if numRows-1 > MaxCount % If it is more than the current highest
			MaxCount = numRows - 1; % It is overwritten
		end
		if numRows-1 < MinCount % If it is less than the current lowest
			MinCount = numRows - 1; % It is overwritten
		end
	end
end
AvgCount = TotalCount/TotAmount % Calculates the average number of percolation chains per system
```

Determining the Shortest Percolation Chain in 2D Systems

```
SmallShapes = 0 % Stores all smallest percolation chain lengths from all systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
	SmolShap = 0; % Stores the current shortest percolation chain length
	if isfile(['shapes' num2str(x) '.csv']) % Confirms the file exists in the expected folder
		load (['shapes' num2str(x) '.csv']) % Loading the respective particle chains file
		CurShape = evalin('base',sprintf('shapes%d',x)); % Retrieves the chains from the file
		[numRows,numCols] = size(CurShape); % Determines the size of the array
		if numRows ~= 1 % Confirms there is at least one chain in the system
			for y = 1:numRows-1 % Loops through all the chains in the system
				ShapLength = 0; % Stores current chain length
				for z = 1:numCols % Loops through the columns
					if CurShape(y,z) == 0 % Checks if the chain is still going
						if ShapLength < SmolShap | SmolShap == 0 % Checks if the current chain is shorter than the previously shortest chain
							SmolShap = ShapLength; % If so it replaces it
						end
					end
					ShapLength = ShapLength + 1; % Increments the chain length counter
				end
			end
		end
	end
	SmallShapes(x) = SmolShap; % Stores the smallest chain from this system
End
```

Determining the Longest Percolation Chain in 2D Systems

```
LargeShapes = 0 % Stores all largest percolation chain lengths from all systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
	LorgShap = 0; % Stores the current longest percolation chain length
	if isfile(['shapes' num2str(x) '.csv']) % Confirms the file exists in the expected folder
		load (['shapes' num2str(x) '.csv']) % Loading the respective particle chain file
		CurShape = evalin('base',sprintf('shapes%d',x)); )); % Retrieves the chain
from the file
		[numRows,numCols] = size(CurShape); % Determines the size of the array
		if numRows ~= 1 % Confirms there is at least one chain in the system
			for y = 1:numRows-1 % Loops through all the chains in the system
				ShapLength = 0; % Stores current chain length
				for z = 1:numCols % Loops through the columns
					if CurShape(y,z) == 0 % Checks if the chain is still
going
						if ShapLength > LorgShap | LorgShap == 0
% Checks if the current chain is longer than the previously longest chain
							LorgShap = ShapLength; % If so it
replaces it
						end
					end
					ShapLength = ShapLength + 1; % Increments the
chain length counter
				end
			end
		end
	end
	LargeShapes(x) = LorgShap; % Stores the largest chain from this system
end
```

Determining the Average Percolation Chain Length in 2D Systems

```
Shapes = 0 % Stores the sum of all chain lengths across all systems
ShapeCount = 0 % Stores the number of chains
AvgLength = 0 % Stores the average length of chain
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
	if isfile(['shapes' num2str(x) '.csv']) % Confirms the file exists in the expected folder
		load (['shapes' num2str(x) '.csv']) % Loading the respective particle chain file
		CurShape = evalin('base',sprintf('shapes%d',x)); % Retrieves the chain from
the file
		[numRows,numCols] = size(CurShape); % Determines the size of the array
		if numRows ~= 1 % Confirms there is at least one chain in the system
			for y = 1:numRows-1  % Loops through all the chain in the system
				ShapLength = 0; % Stores current chain length
				for z = 1:numCols % Loops through the columns
					if CurShape(y,z) == 0 % Checks if the chain is still
going
						Shapes = Shapes + ShapLength; % Adds
the chain length to the total of all shape lengths
						ShapeCount = ShapeCount + 1; %
Increments the number of chains counter
					end
					ShapLength = ShapLength + 1; % Increments the
chain length counter
				end
```

```
                    end
               end
          end
end
AvgLength = Shapes/ShapeCount % Calculates the average percolation chain length
```

Visualising 3D System Beds

```
for a = 1:50 % This line starts the loop that will iterate for the number of systems inputted
          if isfile([num2str(a) '.csv']) % Confirms the file exists in the expected folder
               load ([num2str(a) '.csv']) % Loading the respective particle locations file
               CurPart = evalin('base',sprintf('X%d',a))';
               [x y z] = sphere; % Sets up x, y and z, as the coordinates of the sphere particle
               box on % Turns on an outline so the edges of the box are in the image output
               L = length(CurPart) % Sets L to the number of particles in the system
               figure(a) Creates a figure for system a
               for q = 1:L % Loops through each particle within the system
                    s(q)=surf(x*CurPart(4,q)+CurPart(1,q),y*CurPart(4,q)+CurPart(3,q),
               z*CurPart(4,q)+CurPart(2,q)); % Draws the sphere particle onto the figure
                    hold on % Keeps the old figure when adding a new particle to it
               end
          end
end
```

Determining 3D Packing Fractions

```
PartFrac = 0 % Stores the packing fractions for all the systems
PartFracAvg = 0 % Stores the average of the packing fractions across all the systems
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
          PartTot = 0; % Stores the total area of Particles within the currently analysed system
          if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
               load ([num2str(x) '.csv']) % Loading the respective particle locations file
               L = length(evalin('base',sprintf('X%d',x))); % Sets L to the number of particles
in the system
               for q = 1:L % Loops through each particle within the system
                    PartTot = ((evalin('base',sprintf('X%d(q,4)',x)) ^ 3) * (4/3) * pi) +
PartTot % Calculates the volume  of the particle and adds it to the current total volume
               end
               PartFrac(x) = PartTot / (240^3); % Calculates the packing fraction of the
current system
               PartFracAvg = PartFracAvg + PartFrac(x); % Adds the packing fraction of the
current system to the total to be averaged
          end
end
PartFracAvg = PartFracAvg / 50 % Calculates the packing fraction average of the systems
```

Determining 3D Number of Contacts

```
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
          if isfile(['contacts' num2str(x) '.csv']) % Confirms the file exists in the expected folder
               load (['contacts' num2str(x) '.csv']) % Loading the respective particle contacts
file
               load ([num2str(x) '.csv']) % Loading the respective particle locations file
               L = length(evalin('base',sprintf('contacts%d',x))); % Sets L to the number of
particles in the system
               for q = 1:L % Loops through each particle within the system
```

SmolContNo = 0; % Stores the number of contacts for small particles
LorgContNo = 0; % Stores the number of contacts for large particles
f = 1; % The counter for looping through the arrays
while evalin('base',sprintf('contacts%d(q,f)',x)) ~= 0 % Checks the particle has more than zero contacts
if evalin('base',sprintf('X%d(q,4)',x)) == 10 % Determines if the particle is of radius 10 (small) or not (large)
SmolContNo = SmolContNo + 1; % Increments the number of contacts for a small particle
f = f + 1; % Increments the array counter
else
LorgContNo = LorgContNo + 1; % Increments the number of contacts for a large particle
f = f + 1; % Increments the array counter
end
end
SmolContCount(x,q) = SmolContNo; % Stores the number of contacts for each small particle across all systems

LorgContCount(x,q) = LorgContNo; % Stores the number of contacts for each large particle across all systems
end
end
end

Determining Chain Angles in Single Chain Type System (Currently set at n_p = 2)

y=1 % Stores the number of chains
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
load ([num2str(x) '.csv']) % Loading the respective particle locations file
L = length(evalin('base',sprintf('X%d',x))) % Sets *L* to the number of particles in the system
for q = 1:2:L % Loops through each particle within the system skipping middle of chain particles
v1 = evalin('base',sprintf('X%d(q+1,1)',x)) - evalin('base',sprintf('X%d(q,1)',x)); % Calculates the dx between the current particle and the next particle it is attached to
v2 = evalin('base',sprintf('X%d(q+1,2)',x)) - evalin('base',sprintf('X%d(q,2)',x)); % Calculates the dy between the current particle and the next particle it is attached to
v3 = v1/v2; % Calculates the gradient of the chain
v4(x,y) = atand(v3); % Calculates the angle of the chain
y = y + 1; % Increments the number of chains counter
end
end
end

Determining Chain Angles and Packing Fractions in Chain Systems (Based on Particle Size)

ys = 1 % Stores the counter for shorter chain angle array
yc = 1 % Stores the counter for longer chain angle array
PartTotC = 0; % Stores the area covered by larger chains
PartTotS = 0; % Stores the area covered by smaller chains
PartFrac = 0; % Stores all the systems packing fractions

```
PartSum = 0; % Stores the sum of all systems packing fractions
PartAvg = 0; % Stores the average of all systems packing fractions
for x = 1:50 % This line starts the loop that will iterate for the number of systems inputted
    if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
        scount = 1; % Stores the number of smaller chains or single particles
        ccount = 1; % Stores the number of larger chains
        load ([num2str(x) '.csv']) % Loading the respective particle locations file
        L = length(evalin('base',sprintf('X%d',x))) % Sets L to the number of particles in the system
        for q = 1:L % Loops through each particle within the system
            if evalin('base',sprintf('X%d(q,3)',x)) == 5  % Determines of the radius of the particle is 5 (small chain or single) or not (long chain)
                singles(scount,1) = evalin('base',sprintf('X%d(q,1)',x)); % If small, saves the x coordinate here
                singles(scount,2) = evalin('base',sprintf('X%d(q,2)',x)); % And the y coordinate here
                scount = scount + 1; % Then adds one to the counter
            else
                chains(ccount,1) = evalin('base',sprintf('X%d(q,1)',x)); % If large, saves the x coordinate here
                chains(ccount,2) = evalin('base',sprintf('X%d(q,2)',x)); % And the y coordinate here
                ccount = ccount + 1; % Then adds one to the counter
            end
        end

        for q = 1:5:ccount-4 % Loops through the large chains (currently set for np = 5)
            v1 = chains(q+1,1) - chains(q,1); % Calculates the dx between the current particle and the next particle it is attached to
            v2 = chains(q+1,2) - chains(q,2); % Calculates the dy between the current particle and the next particle it is attached to
            v3 = v1/v2; % Calculates the gradient of the chain
            cv4(yc) = atand(v3); % Calculates the angle of the chain and saves it
            yc = yc + 1; % Increments the array counter
        end
        for q = 1:5:scount-4 % Loops through the small chains (currently set for np = 5)
            v1 = singles(q+1,1) - singles(q,1); % Calculates the dx between the current particle and the next particle it is attached to
            v2 = singles(q+1,2) - singles(q,2); % Calculates the dy between the current particle and the next particle it is attached to
            v3 = v1/v2; % Calculates the gradient of the chain
            sv4(ys) = atand(v3); % Calculates the angle of the chain and saves it
            ys = ys + 1; % Increments the array counter
        end

        PartTotC = (((10 ^ 2) * pi) - (((10^2)* (((2*pi)/3) - ((sqrt(3))/2))) * (0.8))) * (ccount); % Calculates the area covered by larger chains
        PartTotS = (((5 ^ 2) * pi) - (((5^2)* (((2*pi)/3) - ((sqrt(3))/2))) * (0.8))) * (scount); % Calculates the area covered by smaller chains
        PartFrac(x) = (PartTotS + PartTotC) / (600^2); % Calculates the systems packing fraction
        PartSum = PartSum + PartFrac(x); % Adds the packing fraction to the total for calculating the average
    else
        l = x % Outputs the file number if the file does not exist for error checking
```

```
        end
end
PartAvg = PartSum / 50 % Calculates the packing fraction average of the systems
```

Determining Chain Angles and Packing Fractions in Chain Systems (Based on Particle Overlap)

```
ys = 1 % Stores the counter for shorter chain angle array
yc = 1 % Stores the counter for longer chain angle array
PartTotC = 0; % Stores the area covered by larger chains
PartTotS = 0; % Stores the area covered by smaller chains
PartFrac = 0; % Stores all the systems packing fractions
PartSum = 0; % Stores the sum of all systems packing fractions
PartAvg = 0; % Stores the average of all systems packing fractions
for x = 1:50  % This line starts the loop that will iterate for the number of systems inputted
        if isfile([num2str(x) '.csv']) % Confirms the file exists in the expected folder
                scount = 1; % Stores the number of smaller chains or single particles
                ccount = 1; % Stores the number of larger chains
                load ([num2str(x) '.csv']) % Loading the respective particle locations file
                L = length(evalin('base',sprintf('X%d',x))) % Sets L to the number of particles in the system
                for q = 1:L % Loops through each particle within the system
                        overlap = 0; % Value for if there is an overlap (1) or not (0)

                        if q ~= L % Determining if the loop is not on the last iteration
                                m = q + 1; % Used to denote the particle placed after the current particle

                                dist = ((evalin('base',sprintf('X%d(q,1)',x)) - evalin('base',sprintf('X%d(m,1)',x)))^2) + ((evalin('base',sprintf('X%d(q,2)',x)) - evalin('base',sprintf('X%d(m,2)',x)))^2);
                                dist = sqrt(dist); % Determining the distance between particle q and m

                                if dist < ((evalin('base',sprintf('X%d(q,3)',x)) + evalin('base',sprintf('X%d(m,3)',x))) - 5) % Determining if they are overlapping
                                        overlap = 1; % If so setting variable to 'yes'
                                end
                        end

                        if q ~= 1 % Does the same as above however checks the particle placed prior
                                n = q - 1;

                                dist = ((evalin('base',sprintf('X%d(q,1)',x)) - evalin('base',sprintf('X%d(n,1)',x)))^2) + ((evalin('base',sprintf('X%d(q,2)',x)) - evalin('base',sprintf('X%d(n,2)',x)))^2);
                                dist = sqrt(dist);

                                if dist < ((evalin('base',sprintf('X%d(q,3)',x)) + evalin('base',sprintf('X%d(n,3)',x))) - 5)
                                        overlap = 1;
                                end
                        end

                        if overlap == 1 % If an overlap is found then the particle is part of a chain
```

```
                    chains(ccount,1) = evalin('base',sprintf('X%d(q,1)',x)); %
Saves the x coordinate here
                    chains(ccount,2) = evalin('base',sprintf('X%d(q,2)',x)); %
Saves the y coordinate here
                    ccount = ccount + 1; % Increments the array counter
               else % If no overlap then particle is a single
                    singles(scount,1) = evalin('base',sprintf('X%d(q,1)',x)); %
Saves the x coordinate here
                    singles(scount,2) = evalin('base',sprintf('X%d(q,2)',x)); %
Saves the y coordinate here
                    scount = scount + 1; % Increments the array counter
               end
          end

          for q = 1:5:ccount-4
               v1 = chains(q+1,1) - chains(q,1);
               v2 = chains(q+1,2) - chains(q,2);
               v3 = v1/v2;
               cv4(yc) = atand(v3);
               yc = yc + 1;
          end

          PartTotC = (((10 ^ 2) * pi) - (((10^2)* (((2*pi)/3) - ((sqrt(3))/2))) * (0.8))) *
(ccount); % Calculates the area covered by chains
          PartTotS = ((10 ^ 2) * pi) * (scount); % Calculates the area covered by singles
          PartFrac(x) = (PartTotS + PartTotC) / (600^2); % Calculates the systems
packing fraction
          PartSum = PartSum + PartFrac(x); % Adds the packing fraction to the total for
calculating the average
     else
          l = x % Outputs the file number if the file does not exist for error checking
     end
end
PartAvg = PartSum / 50 % Calculates the packing fraction average of the systems
```