ABSTRACT

ADDITION OF A CHAIN-CELL SEARCH METHOD AND A VAN DER WAALS FORCE MODEL TO A PARTICLE DYNAMICS CODE

by
Michael J. Sweetman

This project studies the implementation of a computational time saving technique and adds an additional force model into a discrete element method simulation code. Both aspects of the project are focused on increasing the versatility of an existing particle dynamics code by increasing the execution speed and available force models for simulation.

The first portion of the project consisted of adding a function to the collision detection mechanism to hash the particles into a spatial grid. This hashing function allows a search for near neighbor particles to be restricted to the space immediately adjacent to the particle of interest, thereby allowing for significant reductions in the amount of time needed to locate near neighbors. It has been found that the time needed to update the list is reduced to a function of $N^{1.2}$, from a function of N^2, where N is the number of particles in the simulation.

The second portion of the project involves the addition of a Van der Waals force model to the simulation code. This force becomes significant when considering small particles, and in some cases it will be even stronger than the gravitational force. The Van der Waals force is found by integrating the contribution from each molecule in a particle to the Van der Waals potential function over the whole particle, thereby obtaining equations that enable the force to be found by treating the particles as a continuum.

ADDITION OF A CHAIN-CELL SEARCH METHOD AND A VAN DER WAALS FORCE MODEL TO A PARTICLE DYNAMICS CODE

by
Michael J. Sweetman

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Mechanical Engineering

Department of Mechanical Engineering

May 2003

APPROVAL PAGE

ADDITION OF A CHAIN-CELL SEARCH METHOD AND A VAN DER WAALS FORCE MODEL TO A PARTICLE DYNAMICS CODE

Michael John Sweetman

Dr. Anthony D. Rosato, Thesis Advisor — Date
Associate Chairperson for Graduate Studies and Professor of Mechanical Engineering, NJIT

Dr. Rong-Yaw Chen — Date
Professor of Mechanical Engineering, NJIT

Dr. Zhiming Ji — Date
Associate Professor of Mechanical Engineering

BIOGRAPHICAL SKETCH

Author: Michael J. Sweetman

Degree: Master of Science

Date: May 2003

Undergraduate and Graduate Education:

- Master of Science in Mechanical Engineering
  New Jersey Institute of Technology, Newark, New Jersey, 2003

- Bachelor of Science in Mechanical Engineering
  New Jersey Institute of Technology, Newark, New Jersey, 2001

Major: Mechanical Engineering

... to determine, as far as a combination of mathematics and physics will permit, the influence of friction and cohesion in some problems of statics.

- Charles Augustin de Coulomb

ACKNOLEDGEMENT

I would like to express sincere thanks and appreciation to my advisor, Dr. Anthony Rosato for providing encouragement, support, guidance and friendship throughout this research.

Special thanks are also given to Dr. Rong Chen and Dr. Zhiming Ji for serving as members of my committee.

I would also like to thank my parents for their support, and for their help in the proofing of this paper.

TABLE OF CONTENTS

Chapter | Page

TABLE OF CONTENTS
(Continued)

Chapter Page

LIST OF TABLES

Table Page

LIST OF TABLES
(Continued)

LIST OF FIGURES

LIST OF FIGURES
(Continued)

CHAPTER 1

INTRODUCTION

1.1 Overview

This project investigates two computational aspects of a discrete element method (DEM) simulation code. The DEM is a method of running computational "experiments" on particle systems. These simulations often involve lengthy computational times, thereby placing a priority on computational efficiency. The first part of this project covers the implementation of an algorithm to improve the speed with which interacting particles are identified. The second part of this project investigates adding an additional force model that will allow the DEM computer code to simulate smaller particles.

The DEM computer code that is used in this project was originally developed by Drs. Otis Walton and Robert Braun at the Lawrence Livermore National Laboratory in Livermore, California. The original purpose of the code was to study uniform shearing flows. It has since been modified for the purpose of expanding its application to include, among other things, Couette flow (Kim, 1992).

In granular science studies, it is often prohibitively difficult to perform certain types of studies on systems of particles. Difficulty can be caused by high temperature, high-pressure, or simply the microscopic size of the particles. Therefore, computer experiments are often used to assist in the validation of theoretical models. They provide a very convenient method of directly studying the behavior of particles at the micro level in cases where it is nearly impossible to perform direct experimentation. If the model is accurate, the computer simulations of the DEM models will often provide a better understanding of the experimental results. Additionally, computer simulations have been used to expose weaknesses in continuum models that were not observed with macro level experimentation (Allen & Tildesley, 1987).

Computer simulations often use very large amounts of computer time and memory. Therefore, a well-optimized simulation scheme will be able to perform more intensive or larger simulations than a scheme that is not optimized. In most cases, the majority of the computational time is used in locating interacting particles, and calculating the forces between them. This project implements a method of reducing the time required to locate interacting particle pairs. The project also expands the investigational possibilities of this code by adding a force model that will allow more accurate simulations of particles on a smaller scale than was previously possible with this computer code.

1.2 History of Granular Science

The study of granular materials began when the human race began to collect and store food. In fact, it appears that Archimedes investigated the behavior of grain in a helical screw pump (Roberts, 1998). There were also other ancient attempts, such as the experiments with dry friction performed by Leonardo da Vinci on piles of sand (Duran, 2000).

Later, other researchers provided background for the first studies in particle technology. One of the most important of these was the work of Charles de Coulomb (1736-1806) with the friction of dry solids (O'Connor & Robertson, 2003). Later, Rankine (1820-1872) and Reynolds (1842-1912) also built on Coulomb's earlier work (Duran, 2000).

Particle technology research, however, was not begun in earnest until relatively modern times. The foundations for the modern studies in particle technology were laid in the late 19th century with the study of grain behavior in silos. Isaac Roberts observed, in the process of designing silos, that the internal friction and the wall friction in a silo limited the pressure on the base of the silo. It was H.A. Janssen, however, who would provide the first theory of pressure in silos. Janssen formulated an equation to predict the vertical pressure in silos, and this equation is still commonly used today. There were also several others who developed theories and performed experiments during this time period. In the early 20th century this work was extended to include the flow of bulk storage (Roberts, 1998).

Ralph A. Bagnold deserves particular credit for his founding work in the field of granular science. He was certainly quite a remarkable individual. In addition to his scientific contributions, he is also remembered as the founder and commander of the famous Long Range Desert Group, the group of commandos in the British Army who provided invaluable intelligence and performed many other daring operations during World War II. While Bagnold was serving as a soldier for the British Army, in the Sinai Desert and in the Sahara Desert, he became interested in the shapes of the sand dunes. He was intrigued by the similarity in the shapes of the dunes that he observed in the different deserts. He began to study the evolution of the sand dunes, and published his findings in 1941, in a book entitled *The Physics of Blown Sand and Desert Dunes.* This book is generally considered to be the foundation of most modern studies in granular science (Underwood, 1998).

Two men: Dr. Andrew W. Jenike and Dr. Jerry R. Johanson considerably advanced the study of cohesive bulk solid flow from storage bins during the 1950's. These men contributed many things to the field; including the establishment of flow modes, a radial stress theory, flow or no flow criteria, and apparatus for the determination of flow properties. One of the significant developments from these advancements, and the work of others at that time, is the realization that in many cases it is convenient, if not necessary, to look at particle technology problems as discrete element problems rather than as continuum problems. As computers have become more powerful, the use of discrete element modeling has become both more prevalent and more practical (Roberts, 1998).

The DEM for granular science is quite similar to the field of molecular dynamics (MD), and in fact, many of the techniques used in DEM simulations are adapted from earlier MD simulations. The earliest computer simulation was carried out at the Los Alamos National Laboratories in 1953. This study also laid the foundations for the modern "Monte Carlo" method of random number simulations. The early models were quite simple, but soon became more realistic, and began to include such models as the Lennard-Jones model for interaction potential (Allen & Tildesley, 1987).

The work has progressed rapidly from these early models. One of the first fluids to be studied was liquid argon. However, most molecular systems require more complicated models, which led to the development of the field of molecular dynamics. Molecular dynamics, in general, refers to simulations that focus on solving Newton's equations for a system of molecules. Alder and Wainwright were the first to accomplish this for a pair of perfectly elastic particles. The next step was to perform a MD simulation on systems which also included the Lennard-Jones force model (Allen & Tildesley, 1987). From this point, studies have progressed to include protein and many other biological molecules. This has facilitated the creation of new drugs, and lead to many advances in gene technology. Many modern MD simulations are now also beginning to include the quantum effects in the simulations. Certainly, the awarding of a Nobel Prize for Chemistry in 1998 for advances in quantum mechanics achieved through the use of computer modeling, showed that using computational techniques for research is now widely accepted in the scientific community (Becker, MacKerell, Roux, & Watanabe, 2001).

Most of the MD simulations use force models for particle contact that are readily adaptable to granular studies. However, the main difference between the MD and DEM is that macroscopic particles interact in non-conservative collisions. Plastic deformation and friction play a large role in determining the behavior of granular materials, while these forces play little or no role on the molecular level.

Therefore, while the fields are similar, the computational techniques and interpretation of results can be very different (Walton, 1984).

1.3 Techniques for Locating Near Neighbors

The first part of this project was to take a technique used in MD simulations and adapt it to a computer code that was developed to perform DEM simulations. The goal of this work was to enable the current code to perform similar simulations to those currently being performed, while reducing the memory and computational requirements.

The basic algorithm for each time step in a DEM simulation is as follows:

1. Find all near-neighbor pairs.
2. Calculate the forces acting on each particle.
3. Integrate the equations of motion.

In general, the largest amount of computational effort is spent on locating interacting particles and calculating the forces on them (Schinner, 1999). Any reduction in computational time in Steps 1 or 2 will often lead to a significant reduction in overall computational time.

The technique that was used in this project is adapted from earlier MD simulations (Allen & Tildesley, 1987). The fundamental idea of this technique is to reduce the amount of computational time involved in locating near-neighbor particle pairs. Instead of performing an exhaustive search for near-neighbor pairs over the entire simulation cell, one would divide this cell into many sub-cells. The search for near-neighbors would then only be conducted over the cell occupied by the particle of interest, and the adjoining sub-cells. The hashing function, which has the role of dividing the simulation area into sub-cells and assigning particles to these sub-cells, can be done very rapidly, and this adds very little to the overall computational time. Despite the additional overhead involved, a substantial amount of time is saved by focusing the search for near-neighbors only in an area where they are likely to be located (Allen & Tildesley, 1987).

1.4 Van der Waals Force Model

On very small systems, such as powder or fine powder systems, forces that can be safely neglected in DEM simulations of larger particles become significant. There are three types of forces that can become significant in simulations of powders. These are the capillary force, the electrostatic force, and the Van der Waals force.

The capillary forces in dry powders are caused by moisture condensing in the powder. As the moisture condenses, it causes the powder grains to stick to one another. In most situations, though not all, the capillary forces are small enough to be ignored. Similarly, in most cases, the electrostatic forces can be neglected. This is because the particles are usually in constant contact with other particles, and in a container that will act as a ground. This has the effect of causing all the particles to carry a similar and small charge. In this environment, generally the electrostatic forces will be repulsive, and not very large in comparison with the other forces.

The third type of force is the Van der Waals force. The Van der Waals is an intermolecular force which does not operate over a very large distance. However, when the contribution from each molecule is integrated over all the molecules in a particle, the resultant inter-particle force can be many times greater than the weight of the particles.

The Van der Waals force can be described by the Lennard-Jones model. When this potential function is integrated over the volume of two colliding molecules, the resultant force can be obtained. The resulting equations are not very computationally expensive. Therefore, this model can be added to the existing simulation code, thereby expanding the application, without causing an unacceptable computational expense (Rietema, 1991).

1.5 Outline of Thesis

The remainder of this thesis is organized as follows. Chapter 2 contains an overview of the code that is used in this project. This outline contains a description of the Verlet table and the force models used. Chapter 3 contains a description of the mechanics of the chain-cell search method that is being added to the simulation code. Chapter 4 presents the results that demonstrate the effectiveness of the chain-cell search method. Chapter 5 contains a description of the Van der Waals force model that is also added to the code, and presents some preliminary results. Chapter 6 presents the conclusions and recommendations for future study.

No literature survey for the whole project is presented. The reason for this is as follows. The project falls very clearly into two distinct parts. For the first part, the concepts presented are widely discussed, but generally implemented without any major alterations. Therefore, a wide ranging literature survey for the first part is unnecessary. The major works regarding the first part will be discussed in this section. The second portion of this project has a separate literature survey that is presented in Chapter 5.

The Verlet table, that is used to keep track of information and reduce computational time, was first introduced in a paper entitled *Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules* (Verlet, 1967). This method is expanded and illustrated in a book by J.M. Haile (Haile, 1992). The code that is used for this project was documented by H. Kim, in a dissertation entitled *Particle Dynamics Modeling of Boundary Effects in Granular Couette Flow.* Finally, the cell linked-list method, that was the focus of the project, was originally presented by Allen and Tildesley in the book *Computer Simulation of Liquids* (Allen & Tildesley, 1987).

CHAPTER 2

OVERVIEW OF THE DISCRETE ELEMENT CODE

2.1 Overview

The three-dimensional shear code used in this project can be used to simulate a number of different motions and configurations of particle systems. The code is set up to allow for either a cubic or a triangular cell, with either open or closed tops. The properties of the materials, such as friction coefficients, can be set in the input file.

The general algorithm of the code is as follows:

1. Read input file and initialize parameters.
2. Set initial particle positions by either reading from a file or using random numbers.
3. Increase particle radii until they reach the desired radius.
4. Update list of near neighbors. (If necessary.)
5. Initialize integration step.
6. Calculate inter-particle forces.
7. Integrate to obtain new particle velocities and positions.
8. Complete the simulation if finished, or repeat from Step 4.

The input file is set up to allow for either triangular or rectangular boundaries. These boundaries can be either hard particles or periodic boundaries. The hard boundaries generally will have the same restitution coefficients and friction coefficients as the other particles. The periodic boundaries will move the particle to the opposite side of the simulation cell, when it passes the boundary, preserving the velocity. The input file is also where the size and properties of the particles are specified. Two different groups of particles are allowed, with each group having different properties. This allows for simulation involving different sizes and types of particles.

The next step the code takes is to assign particle positions. This can be performed either by calling a random number function, or by reading the particle positions from a text file. When the particle positions are assigned, the position represents the position of the particle center. There will usually be overlapping particles when the radius of each particle is considered. Therefore, the radius of the particles is increased in steps. After each step, where the radius is increased, the force routine is called to calculate the inter-particle forces.. These forces are then used to move the particles apart, allowing for further increase in the radius. This process continues until the radii of the particles reach the radius specified in the input file. Once this step is completed, the general simulation loop begins.

The first step in the simulation is to create the linked-list of near neighbor particles. The code performs a loop though all particle pairs, checking to see if the distance separating the particles is less than the search radius. If they are closer than the search radius, the particle pair is added to the list of collisions. This list remains generally the same from one time step to the next, provided that the time step is small. Therefore, the list is only updated when the cumulative maximum position change from each time step is more than half of the search distance. This is illustrated in Figure 2.1. The two particles in that figure would be near neighbors (Verlet, 1967).

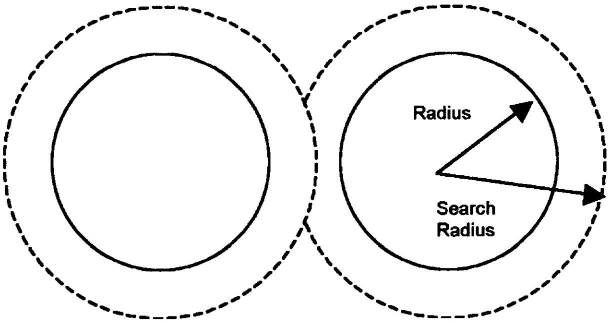


Figure 2.1 Two particles, with the radius and search radius marked.

The force routine then uses the linked-list of particles to calculate the inter-particle forces. The force model used is called a partially-latching spring model, which employs different tangential and normal friction coefficients. The force routine also deletes particle pairs that move outside of the search radius from the linked-list (Walton, 1991).

After the program completes the calculation of the forces, the next step is to integrate the particle positions and velocities. The integration is performed with the leap-frog Verlet algorithm (Becker et al., 2001). This algorithm provides a numerically stable, yet reasonably speedy routine for numerical integration. After the particle velocities and positions have been integrated, the program checks to see if a printout of results is needed, or if the run information should be saved. Then the program checks to see if the maximum time has been reached. If the maximum time has been reached the program ends, otherwise the time is incremented by one time step, and the code returns to the update step.

2.2 Description of Subroutines

The following is a description of the subroutines that make up the Three Dimensional Shear Code. The order in which they are executed is shown in Figure 2.2

- BOUND

  The bound routine assigns velocity, position, and other parameters for the boundary particles.

- DATAIN

  Reads the input data from the input file, this file is called i3ds.

- DATASAVE

  This routine takes the data desired for output, formats it, and prints it in output data files at the time intervals specified in the input file.

- DELETEM

  If the memory in use exceeds a preset threshold, then the deletem routine is called. This routine reduces the search radius for near-neighbors by ten percent, and removes near neighbor pairs from the linked-list that would now be exceeding the search radius.

- DIAGNOS2

  This routine calculates all the long and short term averages otherwise known as the diagnostic information.

- DUMPREAD

  The dumpread routine is used when restarting a previous run. This routine reads in all the information necessary for restart from a previously created dump file called d3ds.

- FINDRAD

  This routine increases the particle radius at t=0 until the radius reaches the value specified in the input file.

- FORCES

  The force routine calculates the inter-particle forces using a partially latching spring model for the normal force, and an incrementally slipping tangential friction model for the tangential force.

- INIT

  This routine initializes the general parameters for the simulation. Initial velocities and positions of the particles are also generated.

- INITCUM1

  This routine initializes the variables needed for short-term averages.

- INITCUM2

  This routine initializes the variables needed for long-term averages.

- INITSTEP

  This routine initializes the variables needed for the integration process.

- INTEG1

  This routine performs the integration for the current time step.

- INTEG2

  This routine calculates the coordinates at the end of each time step, and estimates the velocities at that time step.

- RAND

  This is the random number generator used in the routine. It was developed with emphasis on portability between different machine architectures.

- UPDATE

  The update routine loops through all pairs of particles searching for near-neighbors. When near-neighbor particle pairs are found, they are added to the linked-list, if they were not already in the list (Kim, 1992).

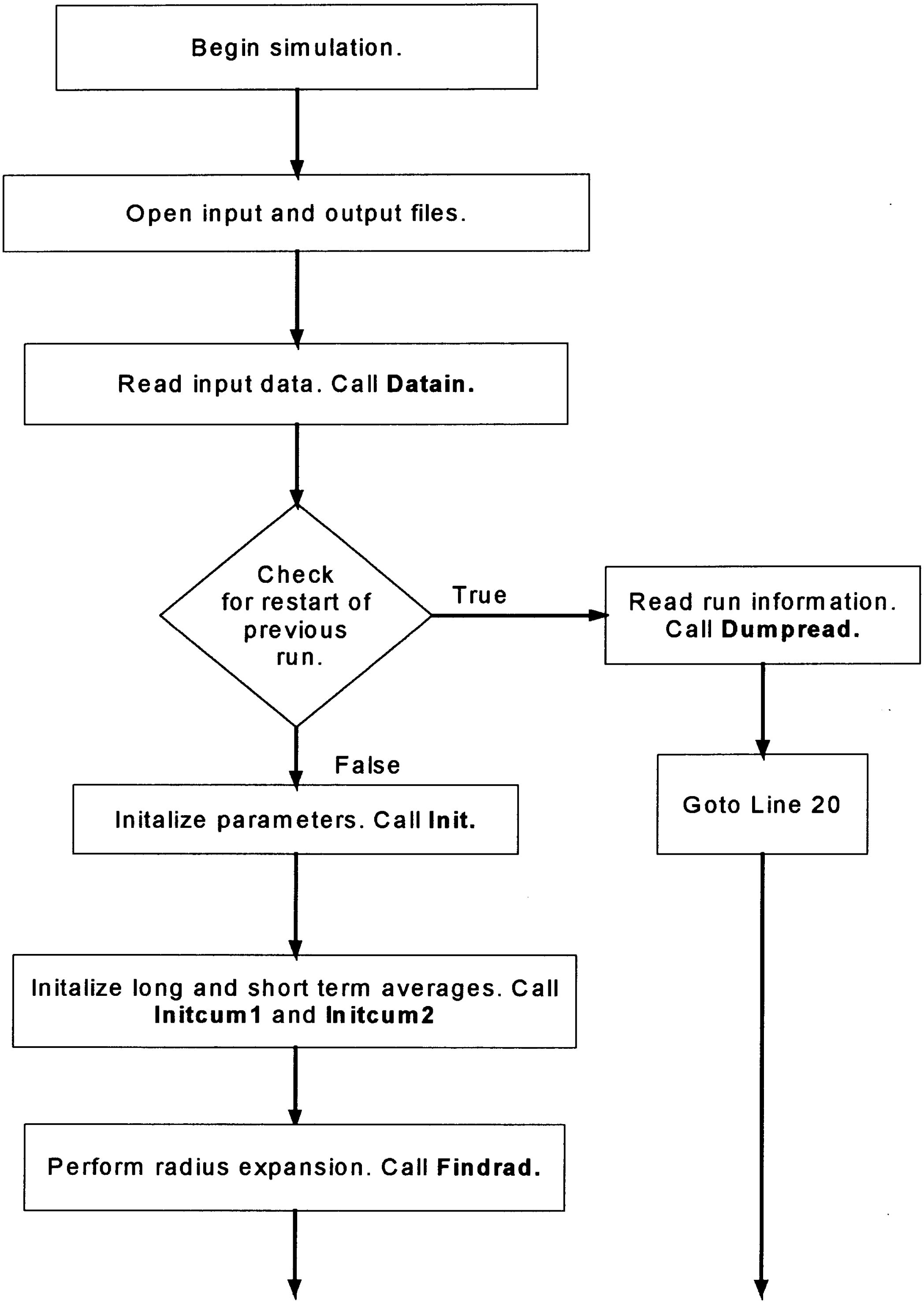

Begin simulation.
Open input and output files.
Read input data. Call **Datain.**
Check for restart of previous run.
True
Read run information. Call **Dumpread.**
Goto Line 20
False
Initalize parameters. Call **Init.**
Initalize long and short term averages. Call **Initcum1** and **Initcum2**
Perform radius expansion. Call **Findrad.**

Figure 2.2 Flowchart of the 3-D Shear code (Kim, 1992).

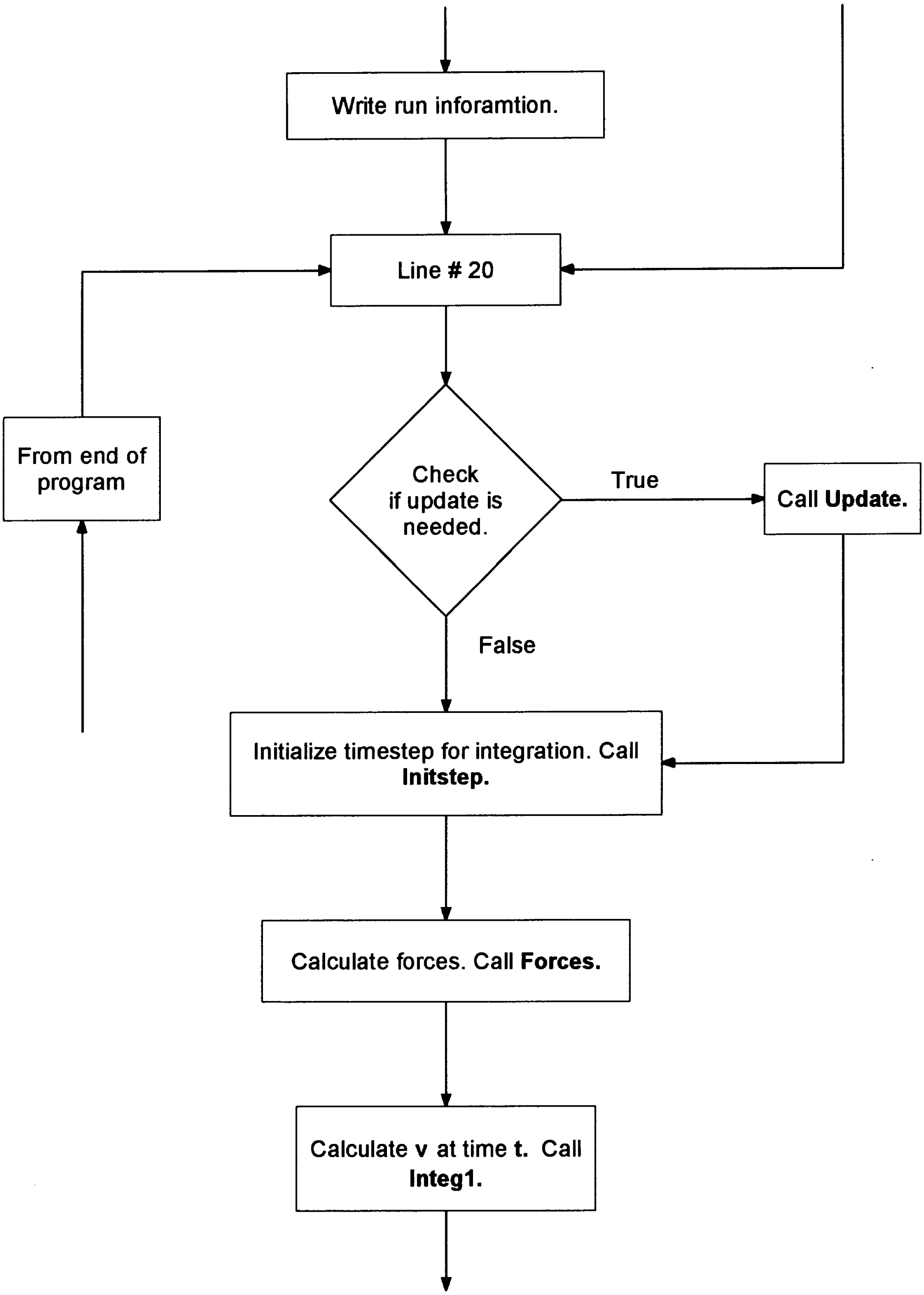

Write run inforamtion.
Line # 20
From end of program
Check if update is needed.
True
Call **Update.**
False
Initialize timestep for integration. Call **Initstep.**
Calculate forces. Call **Forces.**
Calculate v at time **t.** Call **Integ1.**

Figure 2.2 Flowchart of the 3-D Shear code (Continued) (Kim, 1992).

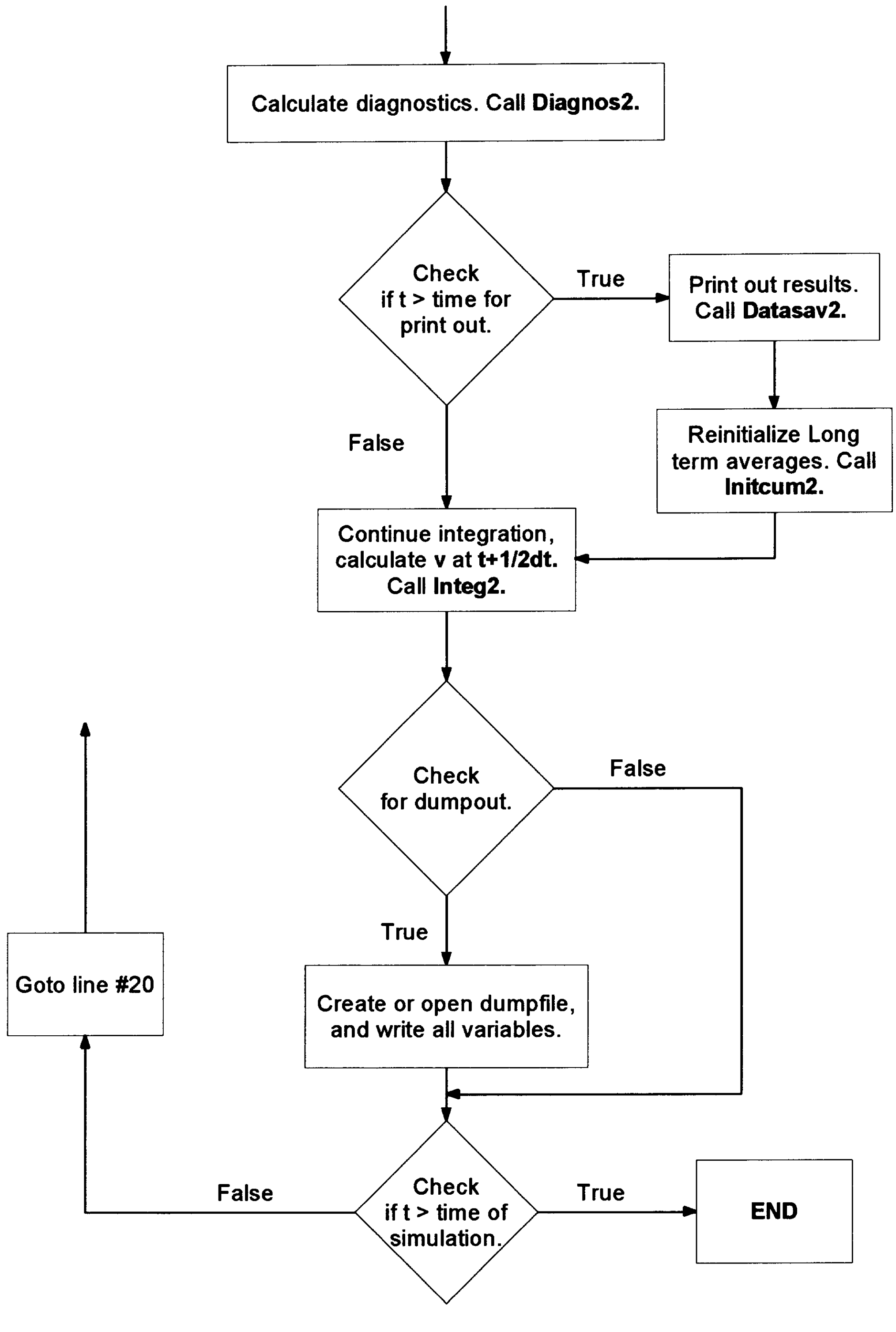

Calculate diagnostics. Call **Diagnos2.**
Check if t > time for print out.
True
Print out results. Call **Datasav2.**
Reinitialize Long term averages. Call **Initcum2.**
False
Continue integration, calculate v at **t+1/2dt.** Call **Integ2.**
Check for dumpout.
False
True
Create or open dumpfile, and write all variables.
Goto line #20
False
Check if t > time of simulation.
True
END

Figure 2.2 Flowchart of the 3-D Shear code (Continued) (Kim, 1992).

2.3 Linked-List Structure

The discrete element code utilizes a Verlet list to keep track of particle contacts. When Loup Verlet introduced this method of bookkeeping it was with the intention of reducing computational time. When Verlet was investigating the properties of a Lennard-Jones system of molecules, he found that he could reduce the computation time by a factor of ten by keeping track of collisions, rather than searching for collisions every time step. He chose a search radius, and then searched for particles that were within the designated search radius. When particle pairs were found they were added to the list of neighbors. The neighbor list was then used each time step for the purpose of calculating inter-particle forces. As long as the neighbor list was updated before any particle moved outside of the search radius, no collisions would be missed (Verlet, 1967).

The discrete element code used in this project uses a similar technique to that employed by Verlet. In addition to keeping track of the near neighbor particle pairs to reduce the frequency that the update routine is called, this code also keeps track of certain properties of each collision. These properties are used by the force routine, which requires information from the previous time step when calculating the forces. The logic used by the code in this project has been documented by Walton in a memo (Walton, 1985), by Rosato in class notes (Rosato, 1989), and by Kim in a dissertation (Kim, 1992).

The linked-list logic uses a number of variables, these variables are as follows:

- NEBOR(i)

  This array contains the memory location of the first near neighbor of particle i. A value of 0 indicates that no near neighbors have been found yet.

- MT1

  This variable stores the value of the next available empty memory location.

- JDX

  This is the subscript used for arrays of 64 bit floating point numbers.

- IDX

  This is the subscript used for arrays of 32 bit integers. On machines where integers and floating point numbers are allotted the same amount of memory, idx will have the same value as jdx.

- NDX(idx)

  This array contains the integer value j, which is a near neighbor of particle i.

- TFX(jdx)

  The x-direction component of the tangential force between particles i and j from the previous time step. This is a floating point value.

- TFY(jdx)

  The y-direction component of the tangential force between particles i and j from the previous time step. This is a floating point value.

- TFZ(jdx)

  The z-direction component of the tangential force between particles i and j from the previous time step. This is a floating point value.

- TM(jdx)

  This is the maximum tangential force before the direction of the slip between particles i and j will change. This is a floating point value.

- A(jdx)

  The virtual overlap, α, between particles i and j. This is a floating point value.

- A0(jdx)

  The value of α that corresponds to zero normal force. This is a floating point value.

- SK(jdx)

  This is the unloading spring constant between particles i and j. This is a floating point value.

- NEXT(idx)

  This is a pointer to the next entry in particle i's linked list. A zero value indicates the end of particle i's linked list. This is an integer value (Kim, 1992).

The value of idx, the index of integers, is calculated from the value of jdx, the index of floating point variables, according to the formula $\mathrm{idx} = (\mathrm{i2or1} * \mathrm{jdx}) - \mathrm{i1or0}$.

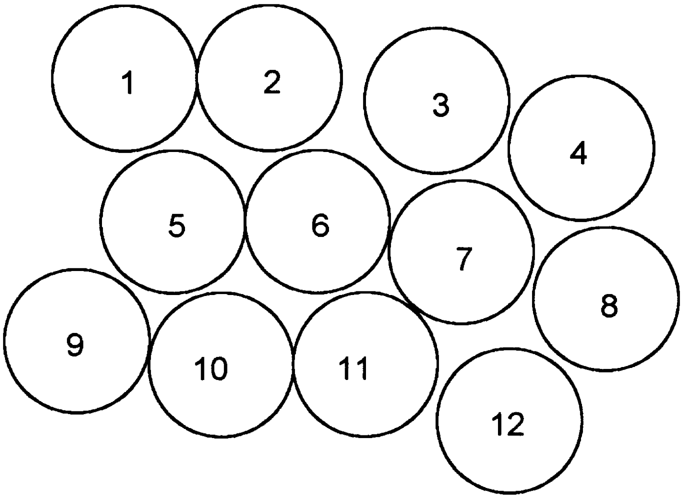


Figure 2.3 Sample system of 12 particles.

Table 2.1 List of Near Neighbors

Particle I	Near-Neighbors (J) of Particle I
1	2,5
2	3,5,6
3	4,6,7
4	7,8
5	6,9,10
6	7,10,11
7	11
8	0
9	10
10	11
11	12
12	0

On architectures where integers and floating point values are the same length, i2or1 = 1, and i1or0 = 0, making idx = jdx. On other architectures, including the Silicon Graphics machines used in this project, integers and floating points have different lengths. In this case, i2or1 = 2 and i1or0 = 0, which makes $\mathrm{idx} = (\mathrm{jdx} * 2) - 1$.

In Figure 2.3, a sample system of twelve particles is shown. A common convention is to set the value of the search radius equal to the particle radius. This generally works out to be a convenient value, that results in a large reduction in computational time, without an unacceptable waste of memory. Taking the particle radius as the search radius, the resultant list of near neighbors is shown in Table 2.1. An important point to notice is that the inter-particle forces need to be calculated only once for each particle pair, since the force acts with equal magnitude on each particle. Therefore, each particle pair is only listed once in Table 2.1, and the forces are only calculated once by the force routine.

The process of generating the linked-list begins before the update routine is called for the first time. The values of the NEBOR array are all set to zero, and MT1 is set equal to 1. The update routine is first called during the radius expansion of the particles. The twelve-particle system, shown in Figure 2.3, will be used to illustrate the procedure for the first time the update routine is called, both for the case of different and identical values of idx and jdx. For the first example, it will be assumed that floating point and integer values are stored in memory locations of the same length, implying that idx = jdx.

The update routine performs a double loop, with the outer loop starting with particle i = 1, going until i = imax, which is the last free particle, or non-boundary particle, in the system. For this example, imax = 12. The inner loop starts at j = i+1 and goes until j = imax. Since the inner loop begins at j= i+1, this ensures that no particle j, that is added to particle i's linked-list, will ever be indexed lower than particle i. This prevents particle pairs from being added to the linked list twice.

With i = 1, the first near neighbor particle j that will be located, is particle 2. The first step in making an addition to i's linked-list is to check if this is the first entry to i's linked list. This is done by checking the value of NEBOR(1). Since NEBOR(1) = 0, this means that this will be the first entry in i's linked-list. The next available memory location is retrieved by checking MT1, which is currently 1. The value of jdx is then set to 1, and since in this example all memory lengths are the same, the value of idx is also set to 1. Next the value of NDX(1) is set equal to the index of the near neighbor, which is 2. Since particle 1 now has a near neighbor, the value of NEBOR(1) is also set equal jdx. MT1 is now incremented to 8, to indicate the next empty memory location. Particle 2 has now been added to particle 1's linked-list, and the structure as it now stands is shown in Tables 2.2 and 2.3. In these tables, and all following tables concerning the Verlet table, only the pointer arrays are shown to be filled in. This is because the other arrays are filled in as needed by the force routine, and are not material to the process of setting up and reading from the Verlet list.

Table 2.2 The NEBOR Array After One Linked-List Addition

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	0	0	0	0	0	0	0	0	0	0	0

Table 2.3 The Linked-List After One Addition

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 0

The next near neighbor that is located for particle 1 is particle 5. First, the value of NEBOR(1) is checked. The value of NEBOR(1) is 1, indicating that there is already a near neighbor of particle 1, located at NDX(1). The indexes jdx and idx are set to 1, and NEXT(1) is checked. The value of NEXT(1) is 0, indicating that the current end of particle 1's linked-list has been reached. Next, MT1 is then checked to find the next empty memory block. Then, NEXT(1) is set to MT1, which currently is 8, indicating that the list for particle 1 continues at memory location 8. Then jdx and idx are set equal to MT1. The value of NDX(8) is now set to 5, indicating that particle 5 is also a near neighbor of particle 1. The memory placeholder, MT1, is then incremented to 16, and the second particle has now been added to the linked-list. The linked-list as it now stands is shown in Tables 2.4 and 2.5. Also, a flowchart illustrating the process is shown in Figure 2.4.

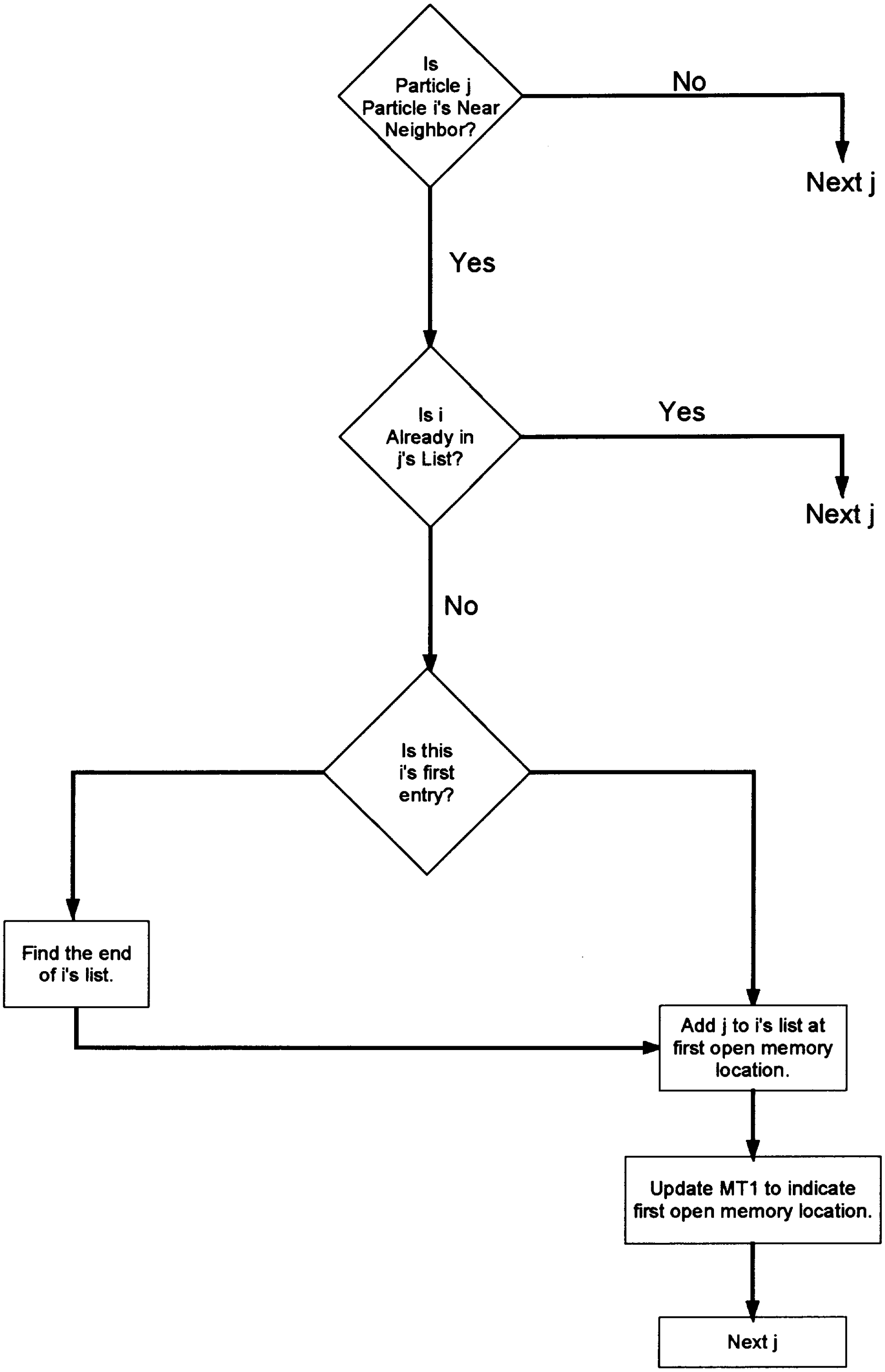

Is Particle j Particle i's Near Neighbor?
No
Next j
Yes
Is i Already in j's List?
Yes
Next j
No
Is this i's first entry?
Find the end of i's list.
Add j to i's list at first open memory location.
Update MT1 to indicate first open memory location.
Next j

Figure 2.4 The process of adding to the linked-list.

Table 2.4 The NEBOR Array After Two Linked-List Additions

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	0	0	0	0	0	0	0	0	0	0	0

Table 2.5 The Linked-List After Two Additions

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 8
8 5	9	10	11	12	13	14 0

Since there are no more near neighbors for particle 1, the search is now performed for near neighbors of particle 2. As was previously mentioned, the search for near neighbors of particle 2 starts with particle 3, since particle 2 would already have been added to the list of any near neighbors with a lower index. Tables 2.6 and 2.7 show the completed linked-list for the twelve-particle system.

Table 2.6 The NEBOR Array for the Completed Linked-List

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	15	36	57	71	92	113	0	120	127	134	0

Table 2.7 The Completed Linked-List

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 8
8 5	9	10	11	12	13	14 0
15 3	16	17	18	19	20	21 22
22 5	23	24	25	26	27	28 29
29 6	30	31	32	33	34	35 0
36 4	37	38	39	40	41	42 43
43 6	44	45	46	47	48	49 50
50 7	51	52	53	54	55	56 0
57 7	58	59	60	61	62	63 64
64 8	65	66	67	68	69	70 0
71 6	72	73	74	75	76	77 78
78 9	79	80	81	82	83	84 85
85 10	86	87	88	89	90	91 0
92 7	93	94	95	96	97	98 99
99 10	100	101	102	103	104	105 106
106 11	107	108	109	110	111	112 0
113 11	114	115	116	117	118	119 0
120 10	121	122	123	124	125	126 0
127 11	128	129	130	131	132	133 0
134 12	135	136	137	138	139	140 0

The next complexity in the organization of this linked-list occurs when different sized blocks of memory are allotted for integers and floating point numbers.

This is the case on many computer architectures, including the Silicon Graphics computers that were used for the simulations in this project. In this situation, the previously discussed variable indexes, idx and jdx, change values. The value of i2or1 becomes 2, and i1or0 is changed to 1. The value of idx is calculated by the formula $\mathrm{idx} = (\mathrm{i2or1} \cdot \mathrm{jdx}) - \mathrm{ilor0}$, which becomes $\mathrm{idx} = (\mathrm{jdx} \cdot 2) - 1$. The linked-list is structured basically the same way, although there are some small variations. The array NDX(idx) still contains the index of the particle j, that is the near neighbor of particle i. The array NEXT(idx) still contains the pointer that indicates the next entry in the near neighbor list.

Returning to the example of the twelve-particle system of Figure 2.3, particle 1 is again the first particle, and the search for near neighbors begins with particle 2. Particle 2 is again the first near neighbor of particle 1. The location of the first open memory block is given by MT1, in this case 1. The value of jdx is set to MT1, and the value of idx is calculated according to the formula $\mathrm{idx} = (\mathrm{jdx} \cdot 2) - 1 = 1$. The value of NDX(idx) is set equal to 2, to indicate that particle 2 is the first near neighbor. The value of NEBOR(1) is set to 1, indicating that the information on the first near neighbor of particle 1 is located at jdx =1. The linked-list as it now stands is shown in Tables 2.8 and 2.9.

Table 2.8 The NEBOR Array for the Linked-List After One Addition

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	0	0	0	0	0	0	0	0	0	0	0

Table 2.9 The Linked-List After One Addition

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 0
8						14

The next near neighbor that will be located for particle 1 is particle 5. The value of NEBOR(1) is 1, indicating that this is not the first near neighbor for particle 1, and the information on this near neighbor is located at jdx =1. The pointer for integers, idx, is calculated to be 1. The value of NEXT(1) is 0, which indicates that the end of the linked-list has been reached, and that the entry can now be added to the linked-list. The value of MT1 is checked to find the next empty memory location, which at this point is 8, and jdx is set to 8. NEXT(1) is changed from 0 to the current jdx, indicating the linked-list now continues. The integer pointer, idx, is calculated from the formula to be 15. The value of NDX(15) is set to 5, indicating that particle 5 is a near neighbor of particle 1. The value of NEXT(15) is set to 0, indicating that this is currently the end of the linked-list. All the other variables associated with this near neighbor pair are floating point variables, and are indexed with jdx, which is currently 8. The linked-list as it now stands is shown in Tables 2.10 and 2.11.

Table 2.10 The NEBOR Array for the Linked-List After Two Additions

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	0	0	0	0	0	0	0	0	0	0	0

Table 2.11 The Linked-List After Two Additions

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 8
8						14
15 5	9	10	11	12	13	21 0
22						28

Setting up the remainder of the linked-list follows closely with the first example, with the only exception being that jdx and idx are different. The completed linked-list is shown in Tables 2.12 and 2.13

Table 2.12 The NEBOR Array for the Completed Linked-List

I	1	2	3	4	5	6	7	8	9	10	11	12
NEBOR(I)	1	15	36	57	71	92	113	0	120	127	134	0

Table 2.13 The Completed Linked-List

NDX	TF	TM	A	A0	SK	NEXT
1 2	2	3	4	5	6	7 8
8						14
15 5	9	10	11	12	13	21 0
22						28
29 3	16	17	18	19	20	35 22
36						42
43 5	23	24	25	26	27	49 29
50						56
57 6	30	31	32	33	34	63 0
64						70
71 4	37	38	39	40	41	77 43
78						84
85 6	44	45	46	47	48	91 50
92						98
99 7	51	52	53	54	55	105 0
106						112
113 7	58	59	60	61	62	119 64
120						126
127 8	65	66	67	68	69	133 0
134						140
141 6	72	73	74	75	76	147 78
148						154
155 9	79	80	81	82	83	161 85
162						168
169 10	86	87	88	89	90	175 0
176						182
183 7	93	94	95	96	97	189 99
190						196
197 10	100	101	102	103	104	203 106
204						210
211 11	107	108	109	110	111	217 0
218						224
225 11	114	115	116	117	118	231 0
232						238
239 10	121	122	123	124	125	245 0
246						252
253 11	128	129	130	131	132	259 0
260						266
267 12	135	136	137	138	139	273 0
274						280

The completed linked-list is accessed when the force routine calculates the inter-particle forces. The mechanics of this are quite simple. The force routine performs one loop starting at particle 1, continuing through the last particle. It takes the particle and then recalls that particle's linked-list. For the example of particle 1, with idx = jdx, the first step is to check NEBOR(1). The value of NEBOR(1) is 1, meaning that the linked list begins with memory location 1, and that jdx is set equal to one. The value of jdx is set equal to idx, and that enables the recall of all information for the collision between particle 1 and its first near-neighbor, particle 2.

After the force routine has finished calculating the forces between particle one and particle 2, the value of NEXT(jdx) is checked. If this value is equal to zero, the list for this particle is finished, otherwise, the value of NEXT contains the next near neighbor. In this case, NEXT(1) = 8, indicating that there is another near neighbor at jdx = 8. This neighbor is particle 5. After the forces have been calculated between particle 1 and particle 5, the value of NEXT(8) is checked, and found to be zero, indicating the end of the linked-list. The force routine then moves to calculate the forces for particle 2.

If the computer system being used sets different lengths of memory for integer and real values, the process works the same way, except for the fact that the arrays are addressed with differently valued arguments. Using the above example, the first near neighbor of particle 1 is particle 2, located at jdx = 1. This time idx also equals 1, and the arrays are addressed with idx or jdx, for integer and real valued arrays, respectively. The next near neighbor is found at

NEXT(1), which is 8, meaning that jdx = 8 and idx =15. Again these two arguments are used to access the arrays. Finally, NEXT(15) is found equal to zero, indicating that the end of the linked-list for particle 1 has been reached.

2.4 Force Model

The normal force model used in this code was developed by Otis Walton. The model is known as the partially-latching spring model, because it uses a constant coefficient of restitution, which produces different spring constants for the loading and unloading portions of the model. The normal force is given as follows:

$$F_N = K_1\alpha = K_1 x_1 \text{ for loading} \qquad (2.1)$$
$$F_N = K_2(\alpha - \alpha_0) = K_2 x_2 \text{ for unloading}$$

In the above equations, α is the symbol representing the overlap between the two particles. The symbol α_0 represents the overlap point at which the force is set to zero, in order to account for the inelasticity of the collisions (Walton, 1991).

This model allows for a constant coefficient of restitution. The coefficient of restitution is defined as follows (Beer & Johnston, 1997):

$$e = \frac{v_2' - v_1'}{v_1 - v_2} = \frac{v_{separation}}{v_{approach}} \qquad (2.2)$$

The total energy of the system is given by:

$$E = T + V = \frac{1}{2}m(\dot{x}^2) + \frac{1}{2}K(x^2) \qquad (2.3)$$

Since the total energy of the system is constant until the energy loss occurs, due to the inelasticity of the collision, the following equation may be written.

$$E = T_{\max} = \mathrm{V}_{\max} \tag{2.4}$$

This force model uses different spring constants for the loading and unloading portions of the collision. This means that the displacement for zero force will be different for the loading and unloading portions. This leads to the following equation, valid during the loading portions of the collision:

$$m\mathrm{V}_a^2 = K_1 x_1^2 \tag{2.5}$$

A similar equation can be derived after the energy loss that will be valid during the unloading portion of the collision.

$$m\mathrm{V}_s^2 = K_2 x_2^2 \tag{2.6}$$

These equations can be solved for the velocities.

$$\mathrm{V}_a = x_1\sqrt{\frac{K_1}{m}} \qquad \mathrm{V}_s = x_2\sqrt{\frac{K_2}{m}} \tag{2.7}$$

The above equations then get substituted into Equation 2.2, the equation for the coefficient of restitution.

$$e = \frac{V_s}{V_a} = \frac{x_2\sqrt{K_2}}{x_1\sqrt{K_1}} \tag{2.8}$$

Since the equations were derived from the equation for total energy at values of maximum energy, $x_1 = x_2$, this gives a constant coefficient of restitution (Kim, 1992):

$$e = \sqrt{\frac{K_1}{K_2}} \tag{2.9}$$

The tangential force model is somewhat more complicated than the normal force model. The basic principle in this model is that the stiffness of a virtual spring, in the direction parallel to the tangential force, decreases until sliding occurs. In each time step, the tangential force is calculated separately in the direction perpendicular and parallel to the direction of the force in the last time step. The final result is then combined, and checked to make sure it has not exceeded the threshold for sliding.

As with the normal force model, the tangential force model also has different spring constants depending on whether the force is increasing or decreasing. The equation for the spring constant is:

$$K_T = \begin{cases} K_O\left[1 - \dfrac{T - T^*}{\mu F_N - T^*}\right]^\gamma & \text{for T increasing} \\ K_O\left[1 - \dfrac{T^* - T}{\mu F_N + T^*}\right]^\gamma & \text{for T decreasing} \end{cases} \tag{2.10}$$

In the above equation, *T* is the magnitude of the tangential force from the current time step. The value of T^* starts off as zero, and it is changed to the magnitude of the tangential force whenever the force changes from increasing to decreasing, or visa versa. The symbol K_o represents the initial tangential stiffness, μ is the coefficient of friction, and γ is a fixed parameter that is set based on the desired behavior for the simulation.

The first step involved in calculating the current tangential force, is to project the tangential force from the last time step, T_{old}, onto the current tangent plane. This projection, $\mathrm{T_o}$, is then normalized, as shown below, to obtain a starting value of the tangential force in the current time step.

$$\mathbf{T} = |T_{old} / T_o| \mathbf{T_o} \tag{2.11}$$

The surface displacement relative to the previous time step is calculated for both the parallel and perpendicular directions, with symbols $\Delta s_{\|}$ and $\Delta s_{\perp}$ respectively. The parameter for maximum tangential force, T^*, is rescaled by the change in the normal force as follows, with the superscript n representing the current time step.

$$T^{*^n} = T^{*^{n-1}} \left[F_N^n / F_N^{n-1} \right] \tag{2.12}$$

The spring constant K_T is then calculated using the new value of T^*. Now the new value of the tangential force can be calculated. This force is calculated first in the tangential and parallel components, and then these are combined into the total force. Since the old tangential force was projected onto the current plane, the perpendicular displacement is treated as new, which makes the spring constant equal to the initial value of K_o. The equations are as follows.

$$\begin{aligned} \mathbf{T}_{\parallel} &= \mathbf{T} + K_T \Delta\mathbf{s}_{\parallel} \\ \mathbf{T}_{\perp} &= K_o \Delta\mathbf{s}_{\perp} \end{aligned} \tag{2.13}$$

Finally, the equations are combined to give the total tangential force for the current time step *n*.

$$\mathbf{T}^n = \mathbf{T}_{\parallel} + \mathbf{T}_{\perp} \tag{2.14}$$

This value is checked to make sure it does not exceed the criteria for sliding. If it does, the magnitude of the force is scaled back so that it is equal to μF_N (Walton, 1991).

Both the normal force model and the tangential force model require information from the previous time step to complete the calculations. The normal force model saves the spring constant and the value that corresponds to zero normal force. The tangential force model needs the value of the force from the previous time step, in order to project the old force onto the current tangent plane. Also, the tangential force model needs the maximum value of the tangential force to be carried through from previous time steps. These values are specific to each colliding pair of particles, and therefore, a linked-list is necessary to efficiently store the information.

2.5 Integration Method

A common numerical integration algorithm is the Verlet integrator. This integration method is derived from two Taylor series expansions, one forward from the current time step, and one backward from the current time step. The two Taylor expansions are shown below, with *r* being the position, v being the velocity, *F* being the force, m being the mass, *t* being time, and *n* representing the time step (Becker et al., 2001).

$$r_{n+1} = r_n + \mathrm{v}_n \Delta t + \frac{1}{2}\left(\frac{F_n}{m}\right)\Delta t^2 + O\left(\Delta t^3\right) \tag{2.15}$$

$$r_{n-1} = r_n - \mathrm{v}_n \Delta t + \frac{1}{2}\left(\frac{F_n}{m}\right)\Delta t^2 - O\left(\Delta t^3\right) \tag{2.16}$$

Equation 2.15 is added to Equation 2.16 to give an expression for the position at the new time step, shown in Equation 2.17.

$$r_{n+1} = 2r_n - r_{n-1} + \frac{F_n}{m}\Delta t^2 + O\left(\Delta t^2\right) \tag{2.17}$$

Equation 2.16 is subtracted from Equation 2.15 to give an expression for the velocity at the new time step, shown in Equation 2.18.

$$\mathrm{v}_n = \frac{r_{n+1} - r_{n-1}}{2\Delta t} + O\left(\Delta t^2\right) \tag{2.18}$$

These equations can then be used to form the general integration algorithm. This algorithm first involves using the current positions to calculate the current force. Then the current force, current position, and previous position can be used to calculate the position for the next time step. Finally, the positions from the next and previous time steps can be used to calculate the current velocity.

The Verlet algorithm has a number of advantages and disadvantages. The most important advantages, in the context of this project, are that it gives an accurate approximation of position ($O(\Delta t^4)$), and is relatively fast. The main disadvantage is that the approximation of the velocity is much less accurate ($O(\Delta t^2)$). The leap-frog Verlet algorithm was developed to eliminate this disadvantage, and this is the method used in this project.

In the leap-frog Verlet method, the velocities are calculated at the half time steps, instead of a full time step, forwards and backwards. This method yields the following expressions.

$$\begin{gathered} r_{n+1} = r_n + \mathrm{v}_{n+1/2}\Delta t \\ \mathrm{v}_{n+1/2} = \mathrm{v}_{n-1/2} + \frac{F_n}{m}\Delta t \\ \mathrm{v}_n = \frac{\left(\mathrm{v}_{n+1/2} + \mathrm{v}_{n-1/2}\right)}{2} \end{gathered} \tag{2.19}$$

The algorithm to use these equations for integration is as follows; first, the current position is used to calculate the current force. Then the current force is used, along with the previous half step velocity, to calculate the velocity a half step forward. The half step velocity is then used to calculate the position one time step forward. Finally, the half step velocities are used to calculate the velocities for the current time step (Becker et al., 2001).

The choice of the time step is an important matter. A smaller time step will improve accuracy, but will increase computational time. Therefore, the time step is calculated automatically according to the following method.

The reduced mass μ of two particles is given by the formula:

$$\mu = \frac{m_1 m_2}{m_1 + m_2} \tag{2.20}$$

This reduces to ½ *m* for particles that have the same mass. Then the frequency of the restoration period is taken, since this always has the larger spring constant.

$$t = \frac{\pi}{\varpi} = \pi\sqrt{\frac{\mu}{K_2}} = \pi\sqrt{\frac{m}{2K_2}} \tag{2.21}$$

In Equation 2.21, *t* is the time for the collision, and ϖ is the natural frequency. Also, the assumption has been made that all particles are of uniform mass. Next, K_2 is removed using the coefficient of restitution.

$$t = \pi e\sqrt{\frac{m}{2K_1}} \tag{2.22}$$

Finally, Δt is obtained by dividing by the desired number of time steps.

$$\Delta t = \frac{\pi e \sqrt{\dfrac{m}{2K_1}}}{n} \tag{2.23}$$

The number of time steps, *n*, is then the value that is left as user input. Previous work has indicated that values of between forty and sixty are reasonable (Kim, 1992).

CHAPTER 3

CHAIN CELL SEARCH METHOD FOR LOCATING NEAR NEIGHBORS

3.1 Overview

Many methods of reducing computational time have been used in MD and DEM simulations. One of the earliest of these was the Verlet list, which was discussed earlier (Verlet, 1967). The Verlet list is currently being used in the code for two purposes, to reduce computation time, and reduce memory usage. Another method is the cell index, or chain cell method, which was proposed by Allen and Tildesley (Allen & Tildesley, 1987).

The chain cell method is intended as a replacement for the Verlet table. The basic problem of the Verlet list, that the chain cell method addresses, is that for systems of large particles the testing of every particle pair is inefficient. This method implements a rapid search method based on the location of the particles. The basic scheme involves placing a grid of cells with edge length *M* over the simulation cell. Although *M* will be kept the same for each dimension in this project, there is no need to do so. In fact, it is likely to be more efficient to have a different value for *M* for each dimension, if the shape of the general simulation cell does not approximate a square or a cube (Hockney & Eastwood, 1999). Provided that *M* is larger than the search radius for near neighbors, the search for near neighbors only needs to be performed in the particle's cell, and immediately adjacent cells. This results in an updating routine that scales according to $\frac{N_p \cdot N_p}{N_c}$, where N_p is the number of particles, and N_c is the number of cells. A complete search of all particle pairs would scale according to N_p^2 (Allen & Tildesley, 1987).

3.2 Creating the Chain-Cell Linked-List

The cell index method quickly sorts the particles into the given cells using a relatively simple algorithm. The variables in the algorithm are as follows. The number of cells in the grid in one direction is given by *M*. The cell inverse, or CELLI, is the inverse of the cell edge length. The value of CELLI is found by taking the real value of the integer *M*. The total number of cells, NCELL, is equal to M^2 in a two dimensional system, or M^3 in a three dimensional system. The variable ICELL is used to denote the cell number in which particle i is located. The value of ICELL is determined by the following equation:

$$\mathrm{ICELL} = 1 + \mathrm{INT}(\mathrm{RX}(\mathrm{I}) * \mathrm{CELLI}) + \mathrm{INT}(\mathrm{RY}(\mathrm{I}) * \mathrm{CELLI}) * M + \mathrm{INT}(\mathrm{RZ}(\mathrm{I}) * \mathrm{CELLI}) * M^2 \tag{3.1}$$

There are also three arrays that the program uses. The first, CELL(i), catalogs which cell particle i is placed into. The second array, HEAD(ICELL) , returns the particle i, which is the head of the chain of cell ICELL. The third array, LIST(i), returns the next particle in the linked-list of particle i. The first array, CELL(i), is merely a convenience, which enables the force routine to work through the linked-list in order of the particle index. If the force routine, or any other routine, which needs to use the linked-list, is designed to work through the particles in order of which cell it is positioned in, then the array CELL(i) is unnecessary.

The second and third arrays are the main part of the linked-list. These arrays together form the chain of particles for each cell. The code that is used to create these two arrays is very rapid, and consequently can be run every time step. The algorithm involves the following steps each time it is run.

1. Zero the arrays LIST, HEAD, and CELL.
2. Begin the loop over all particles from 1 to the number of particles.
3. Calculate which cell number (ICELL) particle i is located in.
4. Set LIST(i) = HEAD(ICELL).
5. Set HEAD(ICELL) = i.
6. End the loop.
7. END.

As previously stated, the code that performs these steps is quite rapid. The FORTRAN code is shown below.

```
        DO 100 ICELL = 1, NCELL
              CELL(ICELL)=0
              HEAD(ICELL)=0
              LIST(ICELL)=0
100    CONTINUE
       DO 200 I = 1 ,N
                    ICELL = 1 + INT((RX(I))*CELLI)
:                               + INT((RY(I))*CELLI)  * M
:                               + INT((RZ(I))*CELLI)  * M * M
       LIST(I)  = HEAD(ICELL)
       HEAD(ICELL)  = I
200    CONTINUE
```

It is important that the arrays, HEAD and LIST, are zeroed before the search routine is run each time. This is because these two arrays together form the linked-list for each cell, and it is possible that outdated information will be left in the list if it is not cleared before each run.

The array HEAD(i) contains the highest indexed particles in cell i, a zero value indicates that the cell is empty. This is the first array called when retrieving information from the linked-list. To find the next highest indexed particle in a cell, the array LIST(i) is called. The array LIST(i) will return the next highest indexed particle in the same cell as particle i. A zero value will indicate that there are no more particles in the same cell.

An example of creating the linked list, and retrieving information from the list will be given. The example will be given for a two-dimensional system in order to make the array small enough to be presented in a table format. However, the mechanics of the procedure are identical for three dimensions, with the only difference being the equation to determine the cell number containing the particle. For two dimensions, an abbreviated form of Equation 3.1 is used, as follows:

$$\mathrm{ICELL} = 1 + \mathrm{INT}(\mathrm{RX}(\mathrm{I}) * \mathrm{CELLI}) + \mathrm{INT}(\mathrm{RY}(\mathrm{I}) * \mathrm{CELLI}) * M \tag{3.2}$$

The particle positions that will be used in this example are listed in Table 3.1.

Table 3.1 Particle Positions and Cell Assignments

Particle Number	X Position	Y Position	Expected Cell Number
1	0	0	1
2	0.1	0	1
3	0.1	0.15	1
4	0.25	0	2
5	0.99	0.99	25
6	0.26	0	2
7	0.27	0.26	7
8	0	0.27	6
9	0	0.87	21
10	0.2	0.3	7
11	0.1	0.5	11
12	0.43	0.54	13
13	0.1	0.7	16
14	0.9	0.2	10
15	0.11	0.65	16
16	0.1	0.4	11
17	0.1	0.57	11
18	0.6	0	4
19	0.2	0	2
20	0.11	0.23	6
21	0.29	0.76	17
22	0.31	0.56	12
23	0.57	0.87	23
24	0.43	0.32	8
25	0.85	0.38	10

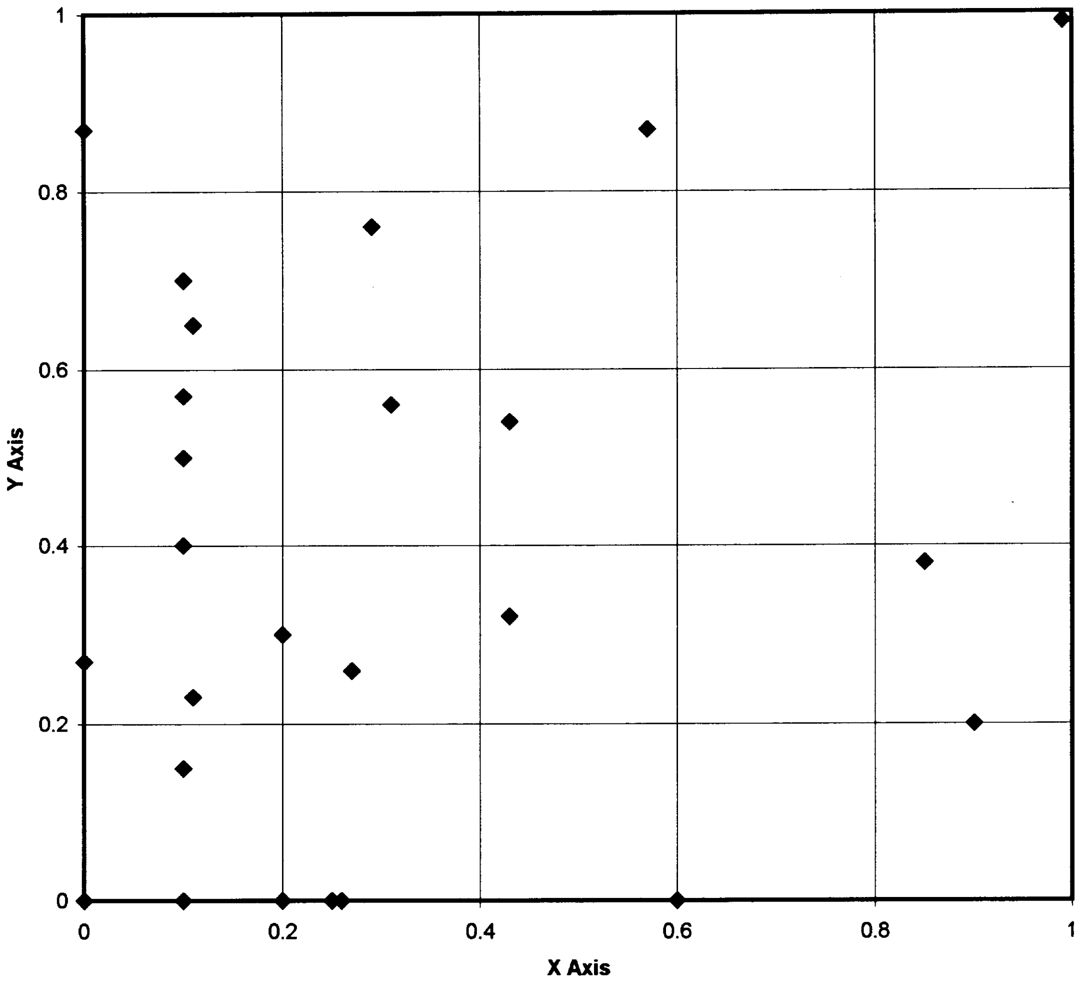


Figure 3.1 The particle positions with the grid overlaid.

The constants that define the grid size must first be set. The example will have twenty-five particles, so $N_p = 25$. The total computational area will be a unit cell. This unit cell will be divided into five cells on each axis, therefore $M = 5$. The total number of cells, NCELL, is equal to M^2, so NCELL = 25. Finally, the value of cell inverse, ICELL, is 5. The particle positions are listed in Table 3.1, and these positions are graphed with the grid overlaid in Figure 3.1.

The first thing the code does is clear the arrays HEAD, LIST, and CELL. After performing this, the code begins a loop over all of the particles, assigning the particles to the cells. The first particle, particle number 1, has coordinates of (0,0). This value of ICELL is calculated according to Equation 3.2, as follows:

$$\mathrm{ICELL} = 1 + \mathrm{INT}(0 * \mathrm{CELLI}) + \mathrm{INT}(0 * \mathrm{CELLI}) * M = 1 \qquad (3.3)$$

This means that particle 1 is assigned to cell number 1. The next step is set LIST(i) = HEAD(ICELL). The variable i represents the position in the loop running over all the particles, which is the particle number. Therefore, this means that LIST(1) = HEAD(1). Since all the arrays were initialized to zero, there is no change here. Next, HEAD(ICELL) is set equal to i, therefore HEAD(1) = 1. Finally, the cell assignment of particle i is recorded in the array CELL, so CELL(1) =1. The linked list, as it stands after the first particle is added, is shown in Table 3.2. Only the first element of each array is shown, since all other values are still zero.

Table 3.2 The Chain-Cell Arrays After the First Particle Has Been Added

I	LIST(I)	HEAD(I)	CELL(I)
1	0	1	1

The code then increments the loop by one, and moves on to particle two. The coordinates for this particle are (0.1, 0). The value of ICELL is calculated as follows.

$$\begin{gathered}\mathrm{ICELL} = 1 + \mathrm{INT}(0.1 * \mathrm{CELLI}) + \mathrm{INT}(0 * \mathrm{CELLI}) * M \\ \mathrm{ICELL} = 1 + \mathrm{INT}(0.1 * 5) + \mathrm{INT}(0 * 5) * 5 \\ \mathrm{ICELL} = 1 + \mathrm{INT}(0.5) + \mathrm{INT}(0) * 5 = 1\end{gathered} \tag{3.4}$$

Since the value of ICELL for particle 2 is also 1, this means that particle 2 becomes the second particle assigned to cell 1. The linked-list is then updated, first by setting LIST(I) = HEAD(ICELL), so LIST(2) = HEAD(1). The value of HEAD(1) was set equal to one in the previous iteration of the loop, at the time indicating that particle 1 was the highest indexed particle in cell 1. Since a higher indexed particle has now been found in cell 1, this means that particle 1 is no longer the highest indexed particle in cell 1. Particle 2's pointer will now point to particle 1, as the next highest indexed particle in cell one. Next, the array HEAD is updated to indicate that particle 2 is the highest indexed particle in cell 1. This is done by setting HEAD(ICELL) = i, therefore HEAD(1) = 2. Finally, the cell where particle 2 is assigned is stored in the array CELL, by setting CELL(2) = 1. The linked-list, as it stands after the first two particles have been assigned, is shown in Table 3.3. Again, only the first part of the arrays are shown, as all other values are zero.

Table 3.3 The Chain-Cell Arrays After the Second Particle Has Been Added

I	LIST(I)	HEAD(I)	CELL(I)
1	0	2	1
2	1	0	1

This process continues until all the particles have been assigned to a cell. At that point, the linked-list will be complete. The completed linked-list is shown in Table 3.4. This linked-list is designed to be used directly by the force routine, and the processes of creating it is rapid enough that it can be run every time step.

Table 3.4 The Complete Chain-Cell Arrays.

I	LIST (I)	HEAD(I)	CELL(I)
1	0	3	1
2	1	19	1
3	2	0	1
4	0	18	2
5	0	0	25
6	4	20	2
7	0	10	7
8	0	24	6
9	0	0	21
10	7	25	7
11	0	17	11
12	0	22	13
13	0	12	16
14	0	0	10
15	13	0	16
16	11	15	11
17	16	21	11
18	0	0	4
19	6	0	2
20	8	0	6
21	0	9	17
22	0	0	12
23	0	23	23
24	0	0	8
25	14	5	10

3.3 Reading From the Chain-Cell Linked-List

The force routine needs to be able to retrieve from the linked-list all particle pairs that are near neighbors, in order to calculate the forces between interacting particles. This is a relatively straightforward process, which is essentially a reversal of the process of creating the list. Since the particles are free to move, they can and often do interact with particles in neighboring cells. However, since the length of the cell sides is set to be at least equal to the sum of the particle diameter and the search radius, the particle will not interact with any particles that are not in its own cell, or adjacent cells.

The first step of reading from the linked-list is performed once only, at the beginning of the simulation. This step involves mapping the cells to their neighboring cells. This mapping is then used by the update routine to find neighboring cells when searching for near neighbors. The mapping function that was adapted for this project was originally written by Allen and Tildesley, and published electronically as an appendix to the book *Computer Simulation of Liquids* (Allen & Tildesley, 1981).

The general structure of the mapping function is as follows. First, a loop is made over all the cells for the purpose of finding the neighboring cells. Next, all the neighboring cells are found, and the numbers are recorded in an array. The loop is then incremented to go to the next cell.

The array which holds the numbers of the adjoining cells is called MAP(imap). In a three dimensional simulation each cell will have twenty-six adjoining cells. For simplification purposes, all the cells were included in their own mapping. This makes the process of reading from the mapping easier, as there is no need to treat the current cell any differently than the adjoining cells. Therefore, the size of the mapping is twenty-seven times the number of cells.

The MAP array is structured as follows. Each cell has a block of twenty-seven memory locations, each containing the index of a neighboring cell. The entries for the first cell are the first entries in the array, followed by the entries for the second cell, and the third cell, etc. This array is addressed by finding the location for the first entry for a cell, and then reading the next twenty-seven locations. The equation for finding the location of the beginning of the twenty-seven entries is $\text{imap} = 27(\text{CELL} - 1)$. The locations in the array MAP are then MAP(imap+1) through MAP(imap+27).

The neighboring cells may be on the opposite side of the general simulation cell if periodic boundaries are being used. Therefore, the mapping function must account for this. This causes some complications in calculating the index of the neighboring cells. Basically, the algorithm must be able to identify cells that are on the edge of the simulation cell, and then flip to the other side of the cell when needed.

Identifying neighboring cells is done by modifying the loop that runs over all of the cells. Instead of being a single loop, this is changed to be a composite of three recursive loops. With *M* being equal to the number of cells in each direction, the three loops go from one to *M* for each of the three directions. This accounts for all of the cells, and also accounts for where each cell is in relation to the edges. With IX, IY, and IZ being used to denote the number of the cell for each direction x, y, and z respectively, the cell number ICELL is calculated according to Equation 3.5 (Allen & Tildesley, 1981).

$$\begin{aligned}\text{ICELL} = {} & 1 + \text{MOD}(\text{IX} - 1 + M, M) + \text{MOD}(\text{IY} - 1 + M, M) * M \\ & + \text{MOD}(\text{IZ} - 1 + M, M) * M^2\end{aligned} \tag{3.5}$$

The function MOD(a,b) is a remainder function which returns the integer remainder of a/b. Using the remainder function ensures that when the edge of the general simulation cell is passed in any one direction, the code will flip to the other side of the general simulation cell.

After the cell number is calculated, the place in the array MAP is calculated, and then the twenty-seven entries for the cell are calculated. These numbers are calculated by passing different values for IX, IY, and IZ to Equation 3.5. The FORTRAN code that performs this operation is shown below.

```
icell(ix,iy,iz) = 1 + mod ( ix - 1 + m, m)
 :                     + mod ( iy - 1 + m, m) * m
 :                     + mod ( iz - 1 + m, m) * m * m

    Do 50 iz = 1,m

       Do 40 iy = 1,m

          Do 30 ix = 1,m

             imap = ( icell ( ix, iy, iz) -1 ) * 27

             map(imap +1 ) = icell( ix - 1, iy - 1, iz - 1 )
             map(imap +2 ) = icell( ix - 1, iy - 1, iz     )
             map(imap +3 ) = icell( ix - 1, iy - 1, iz + 1 )
             map(imap +4 ) = icell( ix - 1, iy    , iz - 1 )
             map(imap +5 ) = icell( ix - 1, iy    , iz     )
             map(imap +6 ) = icell( ix - 1, iy    , iz + 1 )
             map(imap +7 ) = icell( ix - 1, iy + 1, iz - 1 )
             map(imap +8 ) = icell( ix - 1, iy + 1, iz     )
             map(imap +9 ) = icell( ix - 1, iy + 1, iz + 1 )
             map(imap +10) = icell( ix    , iy - 1, iz - 1 )
             map(imap +11) = icell( ix    , iy - 1, iz     )
```

```
                map(imap +12) = icell( ix    , iy - 1, iz + 1 )
                map(imap +13) = icell( ix    , iy    , iz - 1 )
                map(imap +14) = icell( ix    , iy    , iz + 1 )
                map(imap +15) = icell( ix    , iy + 1, iz - 1 )
                map(imap +16) = icell( ix    , iy + 1, iz     )
                map(imap +17) = icell( ix    , iy + 1, iz + 1 )
                map(imap +18) = icell( ix + 1, iy - 1, iz - 1 )
                map(imap +19) = icell( ix + 1, iy - 1, iz     )
                map(imap +20) = icell( ix + 1, iy - 1, iz + 1 )
                map(imap +21) = icell( ix + 1, iy    , iz - 1 )
                map(imap +22) = icell( ix + 1, iy    , iz     )
                map(imap +23) = icell( ix + 1, iy    , iz + 1 )
                map(imap +24) = icell( ix + 1, iy + 1, iz - 1 )
                map(imap +25) = icell( ix + 1, iy + 1, iz     )
                map(imap +26) = icell( ix + 1, iy + 1, iz + 1 )
                map(imap +27) = icell( ix,     iy,     iz     )

30           continue

40        continue

50   continue
```

Table 3.5 shows the results of the above code for a simulation cell with M = 5. The first row contains the first iteration of all three of the loops. This is IX = 1, IY =1, and IZ =1. As expected, this is the first cell, and ICELL will return a value of 1. Then the value of imap, the location in the array MAP is found. The second line in Table 3.5 contains the first neighboring cell. This is found from IX = 0, IY =0, and IZ =0. Since cell 1 is an edge cell in all three directions, it is expected that this neighboring cell will be on the opposite side of the general simulation cell in all three directions. This is the case as shown in Equation 3.6, where the value of ICELL is calculated to be 125. The remaining mapping for the first cell is shown in Table 3.5.

$$\begin{aligned} \mathrm{ICELL} &= 1 + \mathrm{MOD}(0-1+5,5) + \mathrm{MOD}(0-1+5,5)*5 \\ &\quad + \mathrm{MOD}(0-1+5,5)*5^2 = 1+4+(4*5)+(4*25) = 125 \end{aligned} \tag{3.6}$$

Table 3.5 Cell Numbers for Various Inputs

IX	IY	IZ	Cell Number
1	1	1	1
0	0	0	125
0	0	1	25
0	0	2	50
0	1	0	105
0	1	1	5
0	1	2	30
0	2	0	110
0	2	1	10
0	2	2	35
1	0	0	121
1	0	1	21
1	0	2	46
1	1	0	101
1	1	1	1
1	1	2	26
1	2	0	106
1	2	1	6
1	2	2	31
2	0	0	122
2	0	1	22
2	0	2	47
2	1	0	102
2	1	1	2
2	1	2	27
2	2	0	107
2	2	1	7
2	2	2	32

Retrieving the list of near neighbors from the linked-list by the force routine, involves looping over all the cells, then calculating forces in order of particle position. The mechanics of this method involve four recursive loops described below.

- The outermost loop starts at cell 1, it moves through the cells, one cell at a time.
    - The next loop picks the primary particle for which near neighbors are found.
        - The next loop moves from the cell of the primary particle, through the neighboring cells.
            - The innermost loop moves through all the particles in the neighboring cells.
            - The innermost loop reads the HEAD and LIST arrays to move through all the particles in the cell that it is assigned from the above loop. It completes when it reaches the end of the chain for the given cell.
        - The loop finding neighboring cells of the primary particle moves to the next cell when the innermost loop completes.
    - The loop picking the primary particle moves through the HEAD and LIST arrays until it reaches the end of the chain for the cell passed down from the first loop.
- The outermost loop starts at cell 1 and is incremented by one until the last cell is reached.

Using the example given when the creation of the list was demonstrated in Table 3.4, the first cell to check for near neighbors is cell one. The head of the chain of cell one is retrieved by checking the array HEAD(1). The value of HEAD(1) is 3, which means that particle 3 is the highest indexed particle in cell 1, and this particle becomes the primary particle for which near neighbors are found. The first near neighbor of particle 3 is found by checking LIST(3). The value of LIST(3) is 2, meaning that particle 2 is the next highest indexed particle in cell 1 after particle 3. Additional near neighbors are found by checking the LIST variable for each additional near neighbor. The next neighbor is found to be particle 1, because LIST(2) = 1. The LIST variable is checked for particle 1, and LIST(1) = 0, indicating that the end of the chain has been reached for cell 1 at particle 1.

The neighboring cells must also be checked for near neighbors of particle 3. For particle 3, located in cell 1, the neighboring cells are cells 2, 6, and 7. Cell 2 has particle 19 at the head of the chain, but no other particles in the cell. Cell 6 is empty, but cell 7 has particle 20 heading the chain, and particle 6 is the only other cell in the chain. This completes the search for near neighbors of particle 3, which heads the chain of cell 1. The search order is demonstrated in Table 3.6. At this point only the two innermost loops have been incremented. In this table, the left column represents the outermost loop.

Table 3.6 Search Order for Near Neighbors of Particle 3

Cell	Primary Particle Cell Number	Neighboring Cells	Particles in Cell
1	3	1	3
			2
			1
		2	19
		6	0
		7	20
			6

The next step is for the second loop to increment the primary particle. This is done by checking the LIST variable for the current primary particle, particle 3. The value of LIST(3) is 2, so particle 2 becomes the new primary particle. The search for near neighbors for particle 2 proceeds in the same manner as before, except for the search in cell 1. Since particle 2 has already been checked against particle 3, there is no need to check again. After the loop has been completed for particle 2, LIST is again checked for the next particle. LIST(2) = 1, so the process is again repeated for particle 1. LIST(1) = 0, indicating that particle 1 is the last in the chain of cell 1. At this point the outermost loop will increment by one, and the process will be repeated for every cell. The complete search order for cell 1 is shown in Table 3.7. This process repeats for each cell until all cells have been checked.

Table 3.7 Complete Search Order for Cell 1

Cell	Primary Particle Cell Number	Neighboring Cells	Particles in Cell
1	3	1	3
			2
			1
		2	19
		6	0
		7	20
			6
	2	1	2
			1
		2	19
		6	0
		7	20
			6
	1	1	1
		2	19
		6	0
		7	20
			6

3.4 Implementing the Chain-Cell Linked-List in the Particle Dynamics Code

The method described in the previous sections of this chapter cannot be added to the existing simulation code as described. The main reason for this is that the force model requires information about each collision be stored and carried to the next time step. The linked-list method, described by Allen and Tildesley, does not allow for this. The information the force model needs is currently stored in a Verlet table, which was also designed to save computational time. These two methods can be combined to take the best time saving features from each method, but the amount of memory required will become unwieldy for very large systems (D'Azevedo, 1994).

With the force model used in this project, a Verlet table is needed to efficiently store the information about each colliding pair of particles that is carried from the previous time step. In addition to enabling the efficient use of memory, the Verlet table reduces computational time by reducing the amount of time spent searching for colliding particles. This is accomplished by finding the particles that are colliding, and particles that are not colliding, yet are within a search radius, and adding all of them to the Verlet table. Then the search for colliding particles, that is performed every time step, needs only to be performed within the Verlet table. The exhaustive search, performed over all particle pairs, is performed only when a particle has moved outside of the search radius (Verlet, 1967).

The cell linked-list method is designed to eliminate entirely the need to perform an exhaustive search over all particle pairs, and eliminate the need for a Verlet table, in order to save memory. This is done by hashing all the particles into cells, and using only a few arrays to save this hashing in a linked-list. The hashing is designed to be performed rapidly enough that it can be run every time step (Allen & Tildesley, 1981).

However, this cell linked-list method doesn't allow for the efficient storing of information about the colliding particle pairs. The code used in this project requires that information be stored about each collision, therefore it is necessary to use the Verlet table. The cell linked-list method can be combined with the Verlet table to further reduce computational time.

The cell linked-list method then acts as a rough search that narrows the location where near neighbors of a particle can be located. The update routine then uses the linked-list, created by the cell linked-list method, to find near neighbors to add to the Verlet table. This table is then used by the force routine to calculate inter-particle forces for a number of time steps. For larger systems, combining the two methods will save much more computational time than the Verlet method alone (D'Azevedo, 1994).

The main drawback to this approach is that the memory usage is quite high. This is because the cell linked-list method adds arrays to the memory, in addition to those already used by the Verlet table. Of the two systems, the Verlet table is generally much more memory intensive than the cell linked-list method. This is because the list of neighbors for each particle can grow rather large. Lists consisting of seventy neighbors for each particle are not unusual. However, memory will only be a problem for very large systems, because memory requirements for both of these methods scale linearly with the number of particles. If memory usage becomes an issue, reducing the search radius will reduce the amount of memory needed, but this will increase the computational time (D'Azevedo, 1994).

CHAPTER 4

RESULTS

4.1 Validation of Results

The chain cell search method described in the previous chapters was tested in a number of different ways. First, the search method had to be validated. Second, the overall computational time for a particular simulation was recorded. Finally, the computational time averages for the different sections of the program were obtained.

First, the search method was validated by checking that it was searching the desired cells. This was verified on a very small system that could be checked by hand. Next, the code containing the chain cell search method was checked to make sure it would generate the same particle trajectories that were generated by a similar code, using an exhaustive search over all particle pairs, or an N-squared search.

Although similar codes, using different search techniques, should theoretically compute the same trajectories, it would not be unexpected for there to be some differences caused by round-off errors. This is because computers store and process data in finite sizes, therefore, all arithmetic that is not done with exact numbers is subject to truncation error. Particularly, the order in which arithmetic operations are performed will have an effect on the final result, even if the order of operations has no theoretical effect. This error will then propagate exponentially over time, increasing to the point where there is great divergence from the theoretical trajectory (Becker et al., 2001; Haile, 1992).

Considering that the order in which arithmetic operations are performed during the simulation can affect the result, it would not be unexpected to find that different trajectories are calculated for the same system of particles if different search methods are used. In this project, the force routine uses the near neighbor list to find interacting particles. The force is calculated between the particle pair, and then the force is added to the total force for each particle. Different search methods will have the neighbor list ordered differently, and as a result, the total force will be summed in different orders for the respective search methods, thereby introducing truncation error.

A test of the chain-cell search method was performed by setting up a sample simulation of twenty thousand particles allowed to free fall in a box. This simulation was performed twice, once with a code using an N-squared search and the second time with a code using a chain-cell search. It was found, that for simulations up to the size of twenty thousand particles, the trajectories that were calculated by the different codes were the same for at least 1.5 seconds of real time.

4.2 Reduction in Time to Update List of Near Neighbors

The time saved by implementing the linked-list method is in the process of updating the list of near neighbors. Therefore, a system of testing the code was implemented to show the amount of time that was saved in the update routine.

As previously mentioned, the update routine is not called at every time step. It is only called when any particle moves more than half of the search radius. In order to get an idea of how much CPU is spent in the update routine, as compared to the rest of the code, the intrinsic timing function was called for the update, force, integration, and diagnostic subroutines each time the update routine was called. The CPU times returned for the force, integration, and diagnostic routines were then divided by the number of time steps since the update routine had last been called. This was to correct for the fact that the update routine is called far less than any of the other routines. The results were averaged over 0.2 seconds of real time. As before, the simulation allowed a number of particles to fall freely in a box.

According to the theory presented by Allen and Tildesley, the time needed to update by checking all possible particle pairs is proportional to the number of particles squared (Allen & Tildesley, 1987). Using *N* as the number of particles, and *t* as time, the following equation may be written.

$$t \propto N^2 \tag{4.1}$$

Inserting a constant *k*:

$$t = kN^2 \tag{4.2}$$

Taking the natural log of both sides of the equation:

$$\ln(t) = \ln\left(N^2\right) + \ln(k) \tag{4.3}$$

Finally, bringing the exponent out of the natural log:

$$\ln(t) = 2\ln(N) + \ln(k) \tag{4.4}$$

This would suggest that the slope of a graph of the natural log of the CPU time for the update routine, against the natural log of the number of particles would have a slope of approximately 2. This graph is shown in Figure 4.1. The slope of the linear trend line placed through the data is 2.0375. This value is in close agreement with the theory (Allen & Tildesley, 1987).

Figure 4.2 shows the same graph as Figure 4.1, except it is for the chain-cell search method. Here the slope of the trend line equals 1.2085, indicating that the chain-cell search method has indeed changed the dependence of CPU time from being dependent on N-squared, to a much lower exponent.

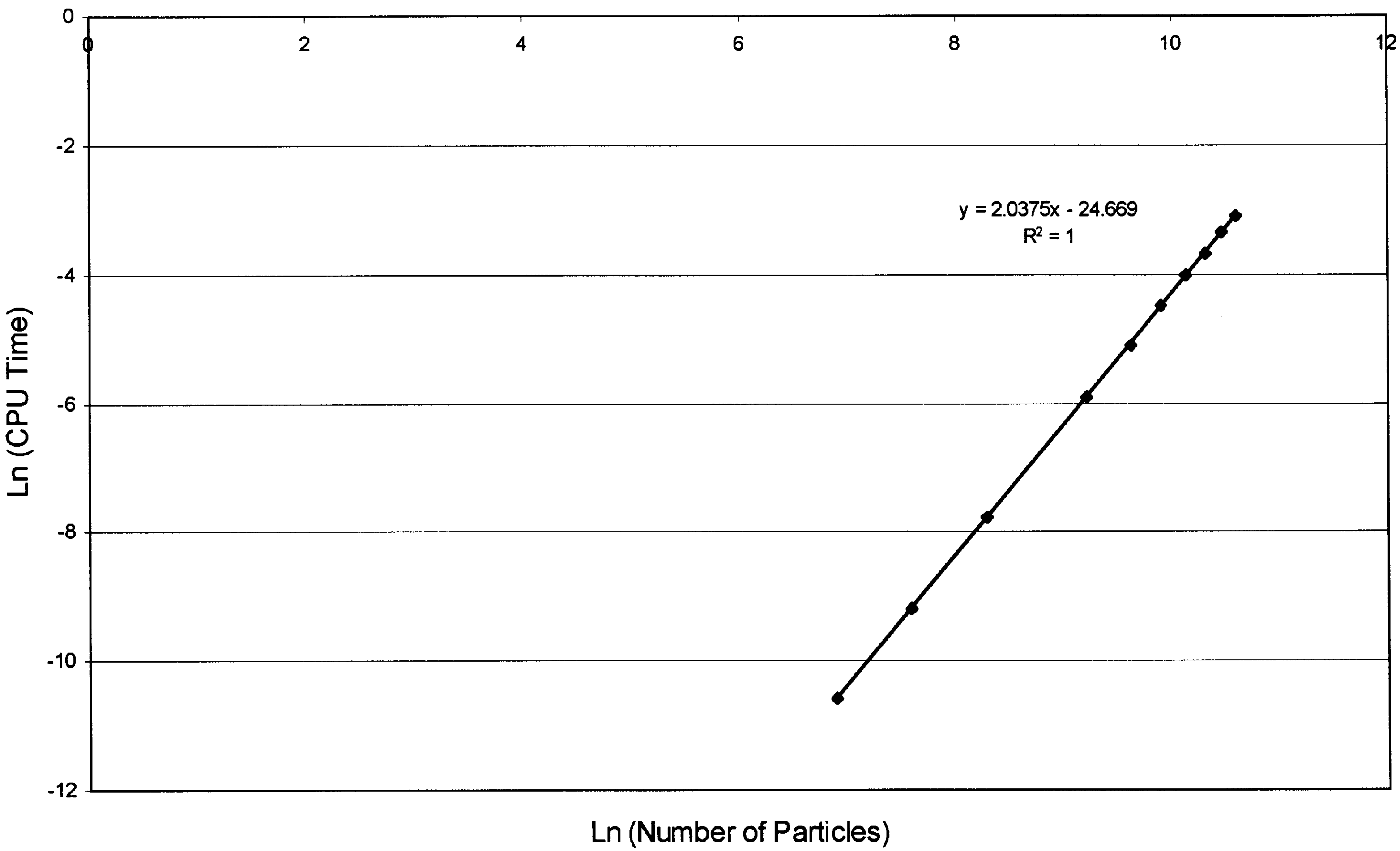


Figure 4.1 Natural log of CPU time vs. the natural log number of particles for the update routine using an N-squared search method.

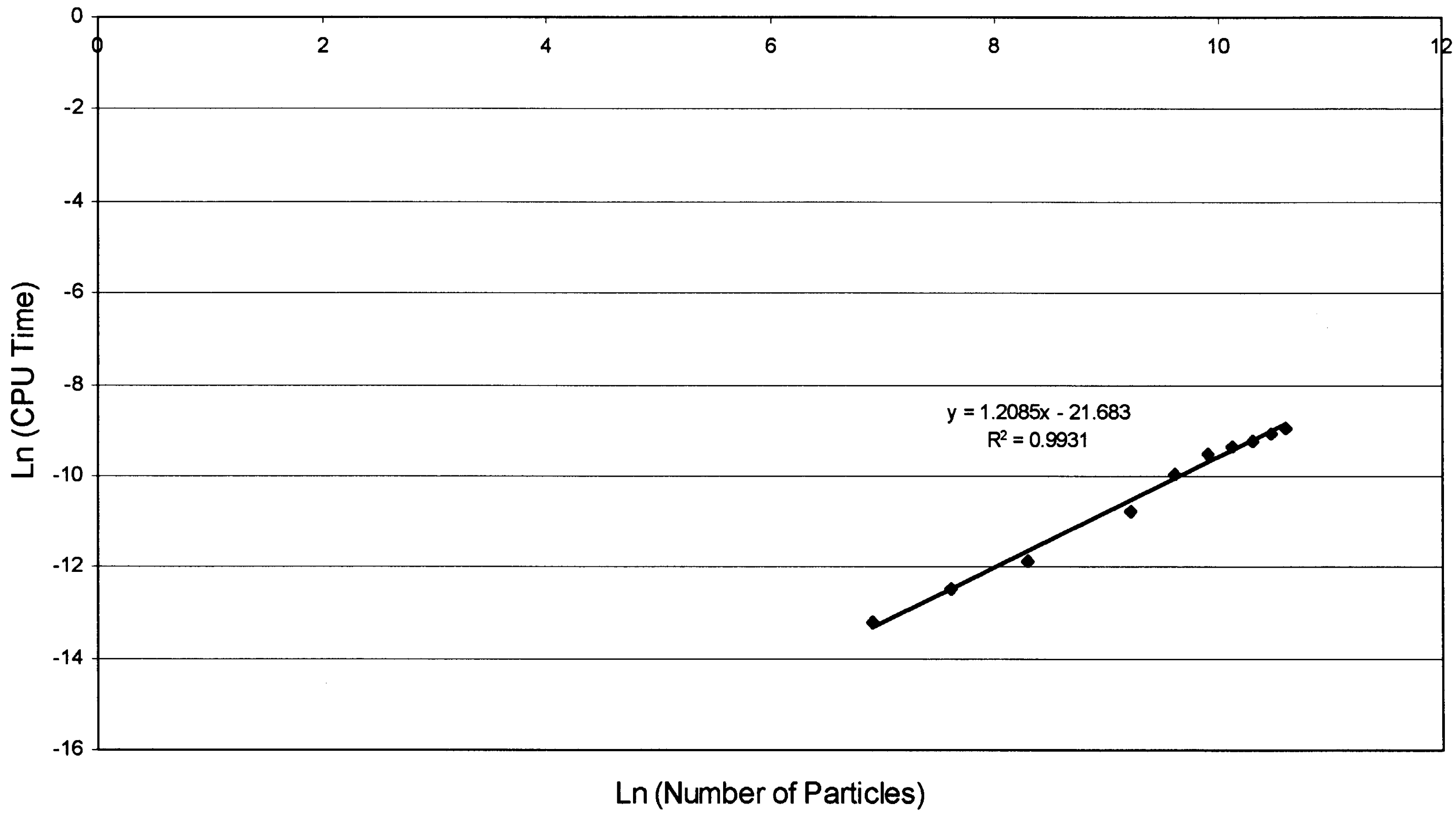


Figure 4.2 Natural log of CPU time vs. the natural log number of particles for the update routine using a chain-cell search method.

Allen and Tildesley also give a formula for estimating the time for the chain-cell method. They suggest that CPU time is proportional to the number of particles multiplied by the number of particles per cell (Allen & Tildesley, 1987).

$$t \propto \frac{(N)(N)}{ncell} \tag{4.5}$$

Using a similar method as before, this can be rearranged as follows:

$$\ln(t * ncell) = 2\ln(N) + \ln(k) \tag{4.6}$$

The graph of this equation is shown in Figure 4.3. The data fell into two distinct groups, each with its own apparent trend line. The reason for this is that the number of cells used in the trials was not picked in the same way for each simulation. There is not any general rule for picking the number of cells, it is left to the user. In these trials, for the simulations up to 10,000 particles, the number of cells was held constant. For the simulations above 10,000 particles, the number of cells was scaled upwards at a rate somewhat approximating the number of particles. The number of cells is shown in Table 4.1.

Table 4.1 Number of Cells and Particles for Chain-Cell Search

Number of Particles	Number of Cells
1000	47952
2000	47952
4000	47952
10000	47952
15000	893952
20000	1124864
25000	1135680
30000	1124864
35000	1135680
40000	1124864

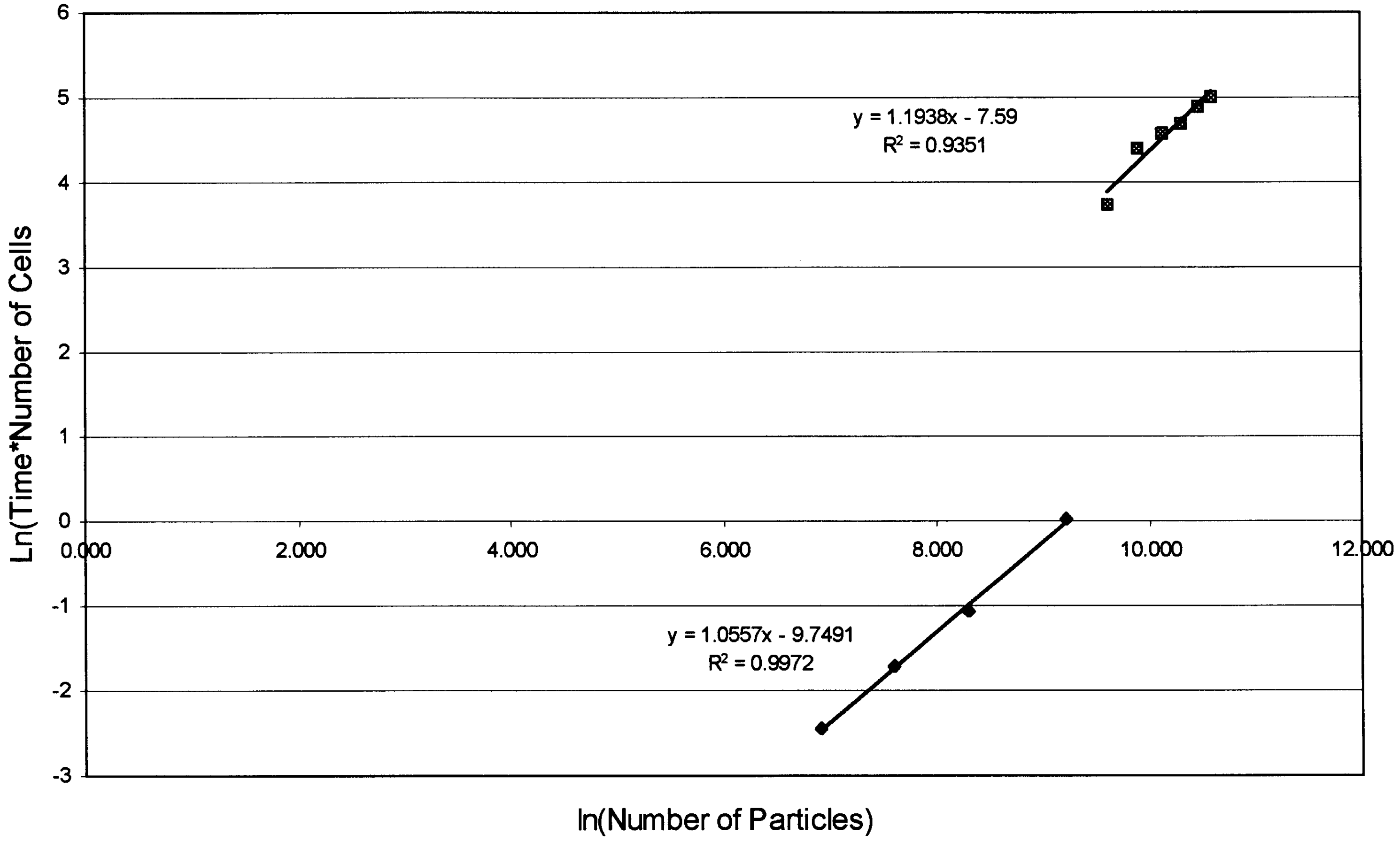


Figure 4.3 Natural log of CPU time multiplied by the number of cells vs. the natural log number of particles for the update routine using a chain-cell search method.

As can be seen in the graph, the relationship suggested by Allen and Tildesley is not apparent. The slope of the graph would be 2, according to Allen and Tildesley, but one section of the graph has a slope of 1.1938, while the other section has a slope of 1.0557.

While this result does not agree with the proportion proposed by Allen and Tildesley, it is not unexpected. The proportion $t \propto \frac{(N)(N)}{ncell}$, fails to take into account that at some point there will certainly be a diminishing return in CPU time reduction for additional cells added. It can be intuitively understood, and is indicated by Figure 4.3, that t is more heavily dependent on the number of particles, than on the number of cells. The conclusion is supported by Hockney and Eastwood who say that $t \propto (N_n)(N) \approx \frac{(N)(N)}{ncell}$. Here N_n is the subset of the total number of particles N that are checked each time step by the chain-cell search. The value of N_n can be estimated by dividing N by the number of cells, but in practice the value of N_n can differ significantly from that estimation (Allen & Tildesley, 1987; Hockney & Eastwood, 1999).

4.3 Reduction in Computational Time

The chain-cell search method was further tested by running a number of simulations. The goal of these simulations was to compare the overall computational time of a DEM simulation with an N-squared search method against simulations using a chain cell search method. The results of this are shown in Figure 4.4 and Table 4.2.

One point to remember when viewing the results presented in this section, is that the chain-cell search method is being benchmarked against a code employing a Verlet table. The Verlet table also provides substantial improvements in computational time (Verlet, 1967). Therefore, any reductions in CPU time are in addition to those realized by the Verlet table.

Table 4.2 Computational Time for the N-Squared Search and Chain-Cell Search

Number of Particles	Time in Hours for the N-Squared Method	Time in Hours for the Chain-Cell Search Method	Percent Improvement
1000	1.60	1.55	2.8%
2000	3.27	3.09	5.4%
4000	7.01	6.59	6.0%
10000	20.73	17.62	15.0%
20000	60.44	40.79	32.5%
30000	138.48	98.88	28.6%
40000	304.08	209.24	31.2%

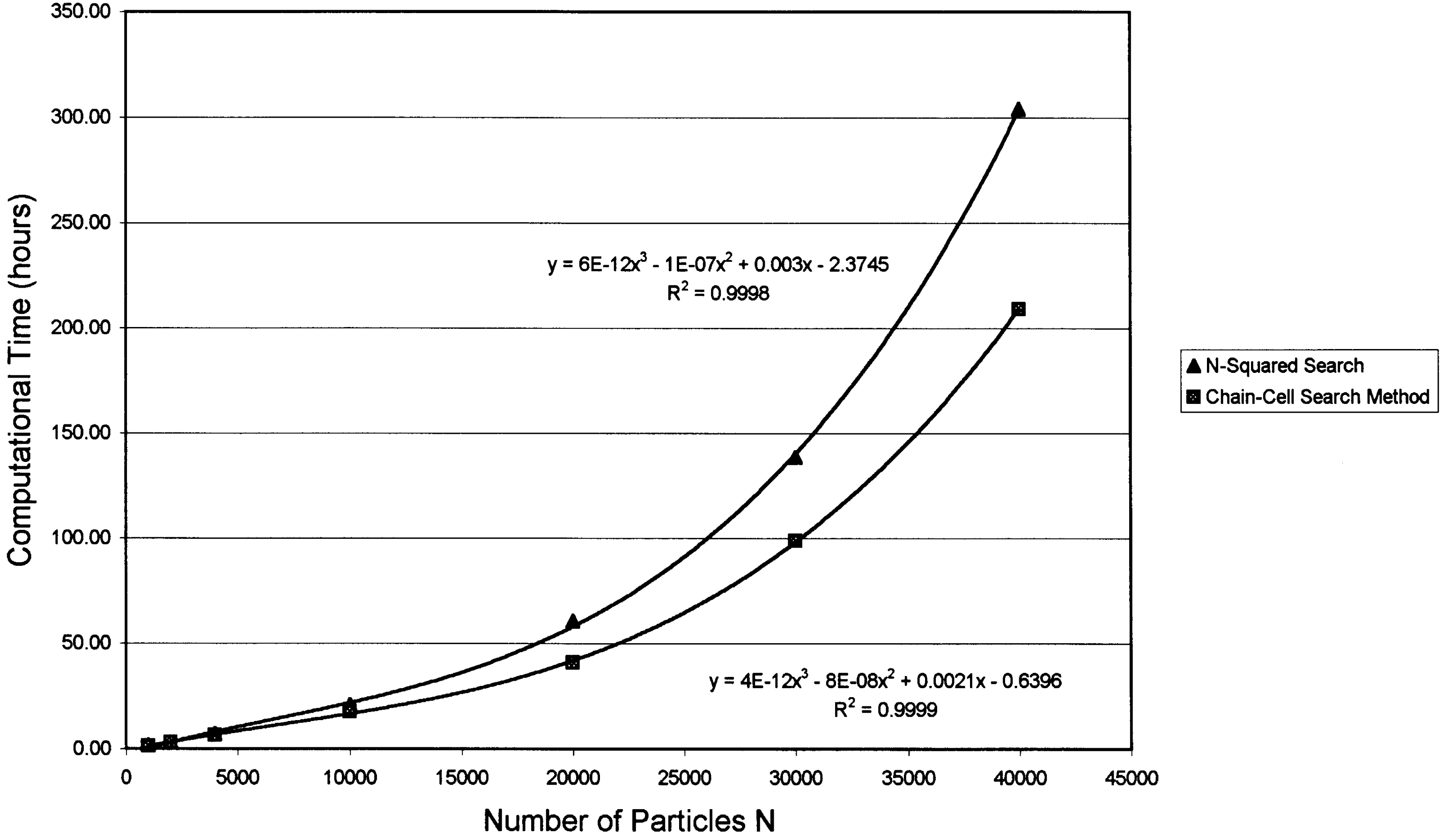


Figure 4.4 Computational time versus the number of particles for both an N-squared search and a chain cell search.

The results shown in the previous table and figure are for a number of simulations that were run for 1.5 seconds, with a varying number of particles. As can be seen from the data, it would rarely be worth the extra complexity in the programming of the simulation to implement a chain-cell search for any simulation with less than 1,000 particles. For simulations of intermediate size, 1,000 to 10,000 particles, it probably is indeed worth the added effort, as the CPU time savings range from between approximately 5% and 15%. For simulations larger than 10,000 particles, an algorithm of this nature is almost mandatory, as the magnitude of time will generally be so large that even a small percentage savings would be quite beneficial.

Another way of looking at the results of adding the chain-cell search method, is to look at the time spend updating the linked-list, as compared to the time spent in the rest of the code. This data is presented in Figures 4.5 and 4.6.

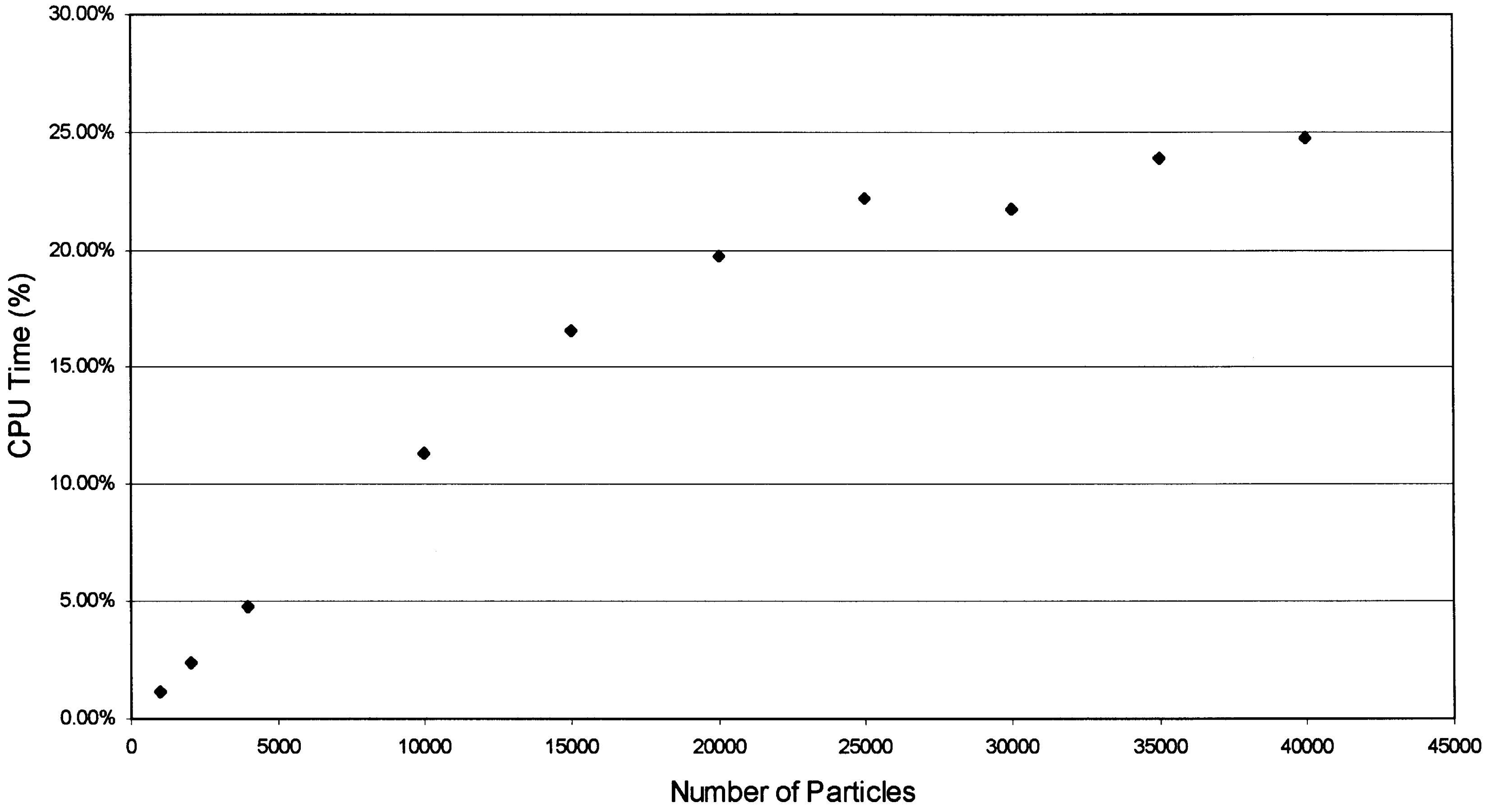


Figure 4.5 CPU time for the update routine for an N-squared search as a percentage of total CPU time vs. the number of particles.

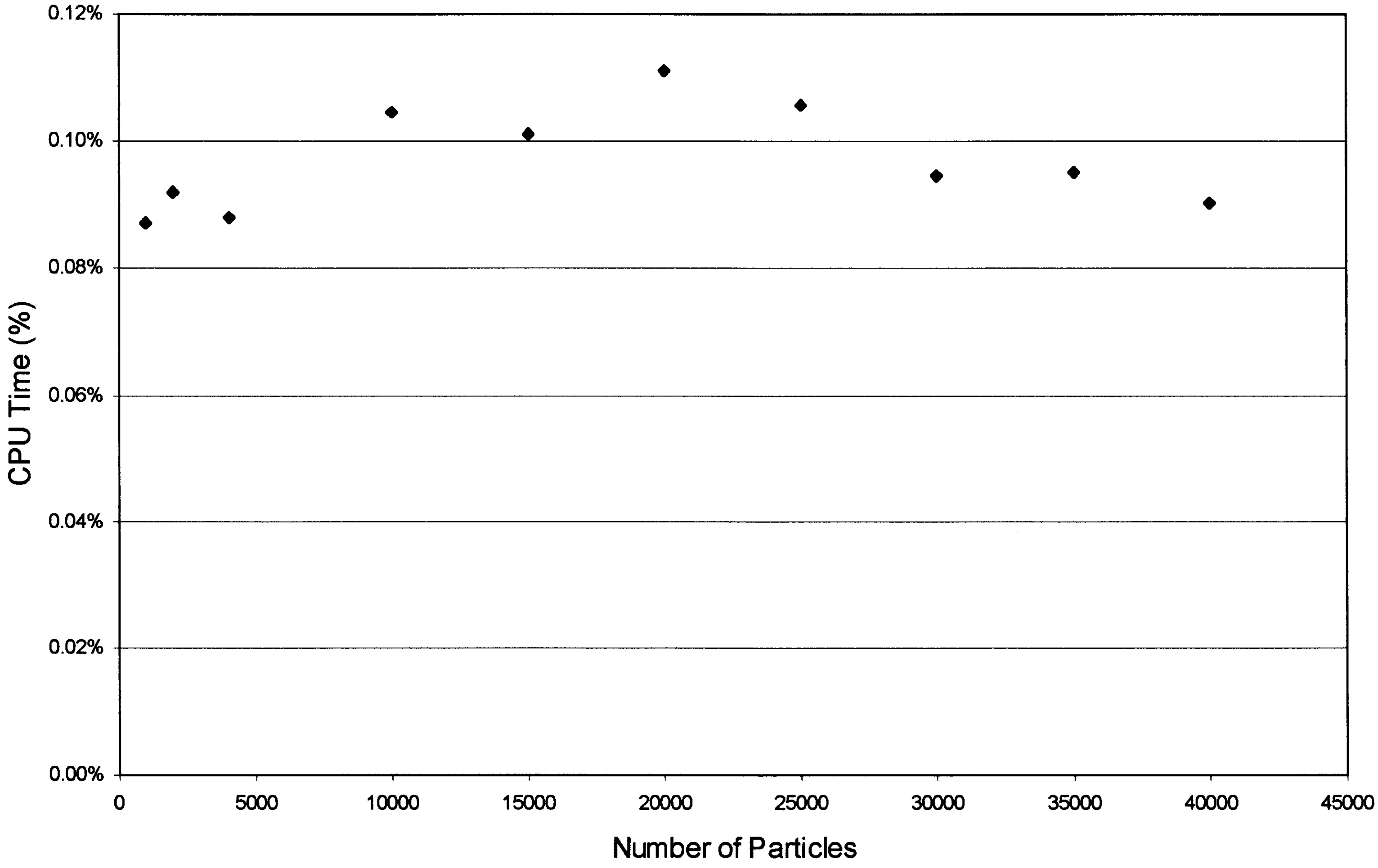


Figure 4.6 CPU time for the update routine for a chain-cell search as a percentage of total CPU time vs. the number of particles.

In Figure 4.5, it can be seen that for the N-squared search, the percentage of total time spent updating the list of near neighbors steadily grows, as the number of particles increases to around 25%. This is contrasted with Figure 4.6, for the chain-cell search, where the updating, as a percentage of total time, remains nearly constant at 0.1%. Clearly, the chain-cell search method is an effective method of searching for near neighbors that scales well as the number of particles increases.

In Figures 4.7 and 4.8, the percentages of total time are shown for all the subroutines, for the N-squared search and the chain-cell search, respectively. In Figure 4.7, the percentage of total time decreases for all the routines except update, as the number of particles increases. This is largely because the time for update is increasing dramatically. In Figure 4.8, by comparison, the percentage of total time stays relatively constant as the number of particles increases.

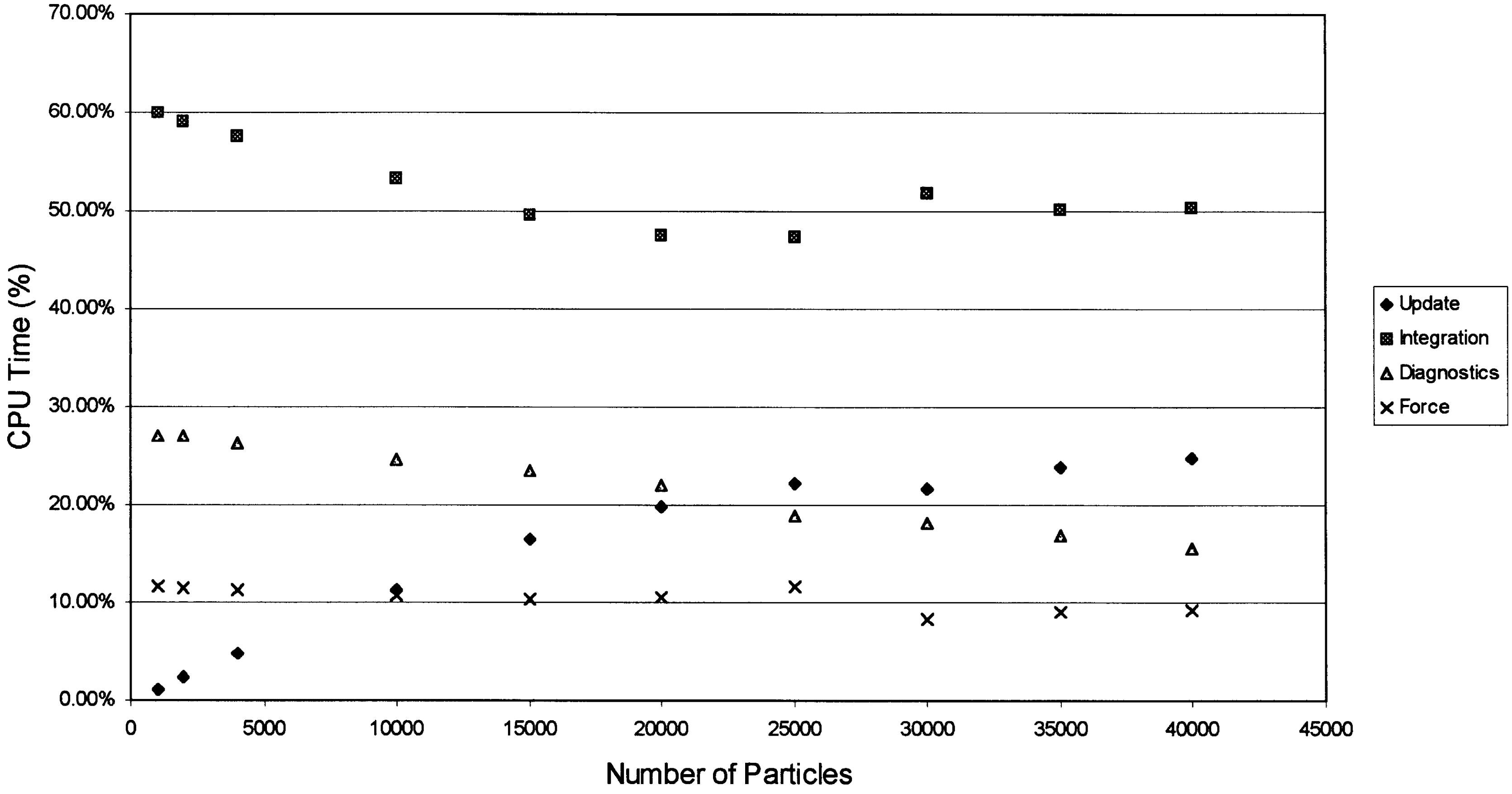


Figure 4.7 CPU time for all the routines with an N-squared search as a percentage of total CPU time vs. the number of particles.

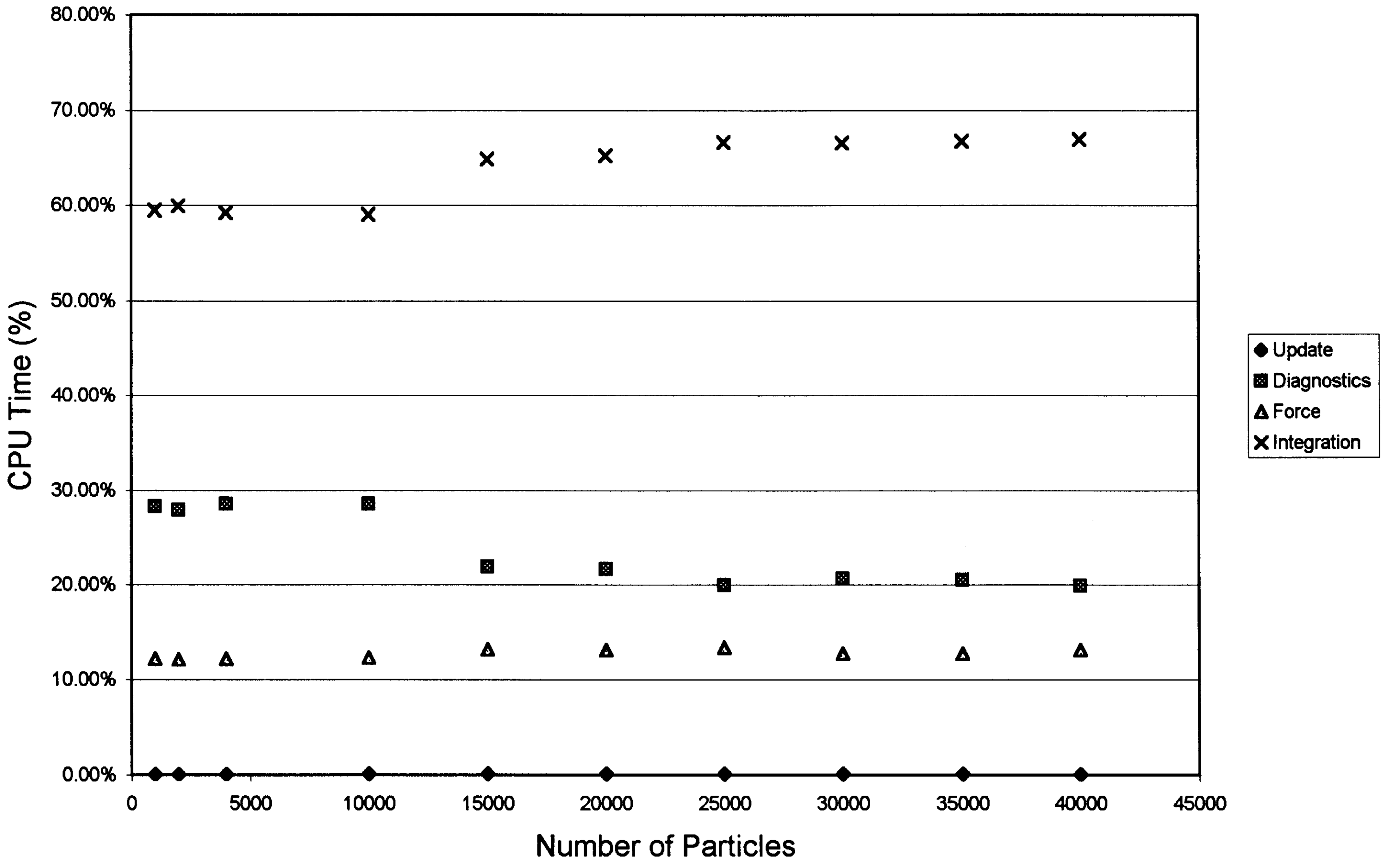


Figure 4.8 CPU time for all the routines with a chain-cell search as a percentage of total CPU time vs. the number of particles.

CHAPTER 5

VAN DER WAALS FORCE MODEL

5.1 Literature Survey

In many particle simulations, the dominant forces are the gravitational force and the particle contact forces. However, as the particles become smaller, other forces begin to dominate the interactions between particles. These forces include the capillary, or surface tension force, the electrostatic force, and the Van der Waals force. Of these forces, the Van der Waals force is often the most significant. This is because the surface tension force, while significant in liquids, liquid suspensions, and drying powders, generally has little effect on dry powders (Rietema, 1991). The electrostatic force is also generally insignificant when compared to the Van der Waals force in powders, and usually it can be safely ignored for particles larger than 1μm (Yang, Zou, & Yu, 2000). However, the literature is somewhat contradictory on this point, as it has been experimentally shown that in some situations the electrostatic force can become significant in particles sized larger than 1μm (Hayden, Park, & Curtis, 2003). Therefore, in general, the Van der Waals force is the only force that will affect very fine dry powders, however, this assumption must be used with caution.

The Van der Waals force is a very short range force that exists between molecules. This force is caused by the instantaneous fluctuations in the charge of individual atoms and molecules (Arunachalam, Marlow, & Lu, 1998). The resultant force between a pair of particles is found by integrating the molecular potential over the whole particle, and then differentiating the resulting potential function with respect to the distance between the particles. This integration was first performed by Hamaker (Hamaker, 1937, cited in Israelachvili, 1992), but it has also been performed by Israelachvili and by Rietema, and is shown in the next section (Israelachvili, 1992; Rietema, 1991). Xie has also performed integration similar to Israelachvili. Although he was chiefly concerned with the effect of gas absorption, he formulated a similar model (Xie, 1997). An interesting phenomenon that occurs with this force, and other short range intermolecular forces, is that the resultant force from the whole particle is much less dependent on distance than the intermolecular force. Consequently, the resultant Van der Waals force between particles acts over a much greater distance than the force between the molecules of which the particles consist (Israelachvili, 1992). The result of this force is that very small particles attract each other. This restricts the movement, affects the packing, and results in the formation of agglomerates (Yu, Feng, Zou, & Yang, 2003).

The model that is used in the theoretical derivation of the Van der Waals force is that of a perfectly round hard sphere, and it contains a strong repulsive component that dominates as the particles move closer together (Rietema, 1991). This model, however, contains a number of problems. First, it contains a singularity when the particles touch, which is physically unrealistic. Second, the model is strongly dependant on both radius and separation distance, so the model is strongly affected by surface asperities. The third problem, which is largely a result of the second problem, is that accurate predictions for the average force are difficult to obtain (Rietema, 1991).

The first problem, the singularity at particle contact, is most often handled by picking a maximum force, and not allowing the force to go above that value, by maintaining a minimum separation distance. The problem then becomes one of selecting a maximum force. Rietema finds the maximum force by using the Lennard-Jones potential function as a model for the Van der Waals potential. This results in a model that has a clear maximum attractive force, and that value can be used to find a minimum separation distance (Rietema, 1991).

The second problem concerns how to handle surface asperities. Closely related to this problem, is the question of how to handle particle deformation. The actual attractive force will generally be much less than predicted by the hard sphere model, because the force due to surface asperities will often dominate the interaction. A common correction for the effect of the surface asperities is to substitute the diameter of the asperities into the model instead of the particle diameter (Castellanos, Ramos, Valverde, & Watson, 2000; Rietema, 1991). Israelachvili suggests calculating the diameter to use in the model based on the experimentally determined adhesion force (Israelachvili, 1992).

There is also a very strong repulsive component to the Van der Waals force that acts at very small distances. However, it is likely that this repulsive component of the Van der Waals force doesn't play a very large role in the particle interaction. This is because the surface asperities will deform as a result of the particle contact, and this contact force will balance the adhesion force

(Rietema, 1991). Forsyth and Rhodes, among others, have presented a model that incorporates the asperities and deformation into the theoretical derivation (Forsyth & Rhodes, 2000).

Yang, Zou, and Yu present another approach to this problem, that is easily adaptable to computer simulation. This model incorporates all the usual forces present in a DEM simulation, the normal contact force and the tangential force. In addition to these forces, the simulation incorporates a Van der Waals force model. The Van der Waals force model uses a diameter smaller than the particle diameter to account for asperities. The model also has a minimum separation distance. It is assumed that the particles begin deforming according to a non-linear spring-dashpot model at the minimum separation distance. This contact force will then act to balance the Van der Waals force (Yang et al., 2000).

The third main problem is that the results produced by the above models can vary widely from each other and from reality. There are a number of causes for this, including the assumptions that are used to produce the models. The principle assumption, which is incorporated into most models, is that the contribution from each molecule to the force exerted by the particle is additive, thereby allowing integration over the entire particle. This is not actually the case, although in general it is a good approximation (Israelachvili, 1992).

The error introduced by the above assumption is small compared with the uncertainty introduced by the surface asperities. Even if models are developed which accurately account for the effects of the surface asperities, the fact remains that there is usually great variation in the sizes of the asperities.

Consequently, there will also be great variations in the cohesive forces of similarly sized particles (Rietema, 1991). This was demonstrated in experiments concerning particle pickup velocity. Pickup velocity is the flow velocity required for a stream to pick up a particle initially at rest. It was shown that in particle sizes where the Van der Waals forces dominated, there was a much higher variation in pickup velocity than in regions where other forces dominated (Hayden et al., 2003).

5.2 Derivation of the Van der Waals Force Model

This section describes the derivation of the model that describes the Van der Waals force. The two particular situations that are of interest in this project are the force between a flat plate and sphere, and the force between two spheres. The derivation will be presented as follows. First, the outline of the derivation will be sketched. Next, the potential function will be found for the cases of a sphere and a flat plate, and two flat plates. Then these potential functions will be used to find the force between a sphere and a flat plate, and between two spheres.

The model that describes the Van der Waals potential between two molecules is the Lennard-Jones model. This model is shown below in Equation 5.1.

$$V_{ss} = C_{ss}\left(\frac{-1}{r^6} + \frac{r_{ss}^6}{2r^{12}}\right) \tag{5.1}$$

The symbols C_{ss} and r_{ss} are material constants, and r is the radius of the molecule. To find the potential between two particles, Equation 5.1 must be integrated across the volume of the particles with respect to the molecule densities, represented by ρ, the number of molecules per unit volume.

$$U_m = \int_{V_1} d\rho_1 \int_{V_2} V_{ss} d\rho_2 \tag{5.2}$$

Finally, the force between the two particles can be found by differentiating the potential function, Equation 5.2, with respect to the distance between the particles, denoted by α (Rietema, 1991).

$$F = \frac{\partial}{\partial \alpha}(U_m) \tag{5.3}$$

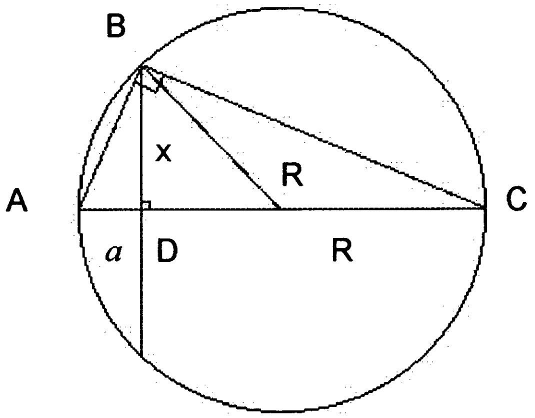


Figure 5.1 The chord theorem (Israelachvili, 1992).

Before beginning the actual derivation, a special relationship, known as the "chord theorem" will be derived. This relationship is a geometric relationship, valid for all spheres, that relates the area of the surface of a plane, located a distance a from the edge of a sphere to the distance a, for a sphere with a radius

R. This area is used in the derivation of the Van der Waals force model as the effective contact area between two large spheres. The dimensions are marked in Figure 5.1. From the Pythagorean theorem:

$$(\mathrm{AC})^2 = (\mathrm{AB})^2 + (\mathrm{BC})^2 = (\mathrm{AD})^2 + (\mathrm{BD})^2 + (\mathrm{BD})^2 + (\mathrm{DC})^2 \tag{5.4}$$

Putting Equation 5.4 in terms of the circle dimensions:

$$4R^2 = a^2 + 2x^2 + (2R-a)^2 \tag{5.5}$$

Equation 5.5 is then simplified and solved as follows:

$$x^2 = (2R-a)a \approx 2Ra \tag{5.6}$$

In situations where R is much larger than a, the approximation that $x^2=2Ra$ can be made. The effective interaction area can then be given by:

$$\pi x^2 = \pi(2R-a)a \approx 2\pi Ra \tag{5.7}$$

The integration of Equation 5.1 can be accomplished by breaking the equation into two sums, and integrating each sum separately. This integration is illustrated and performed for a number of situations by Israelachvili (Israelachvili, 1992). Each part of Equation 5.1 is of similar form, and can be arranged into the form:

$$w(r) = \frac{C}{r^n} \tag{5.8}$$

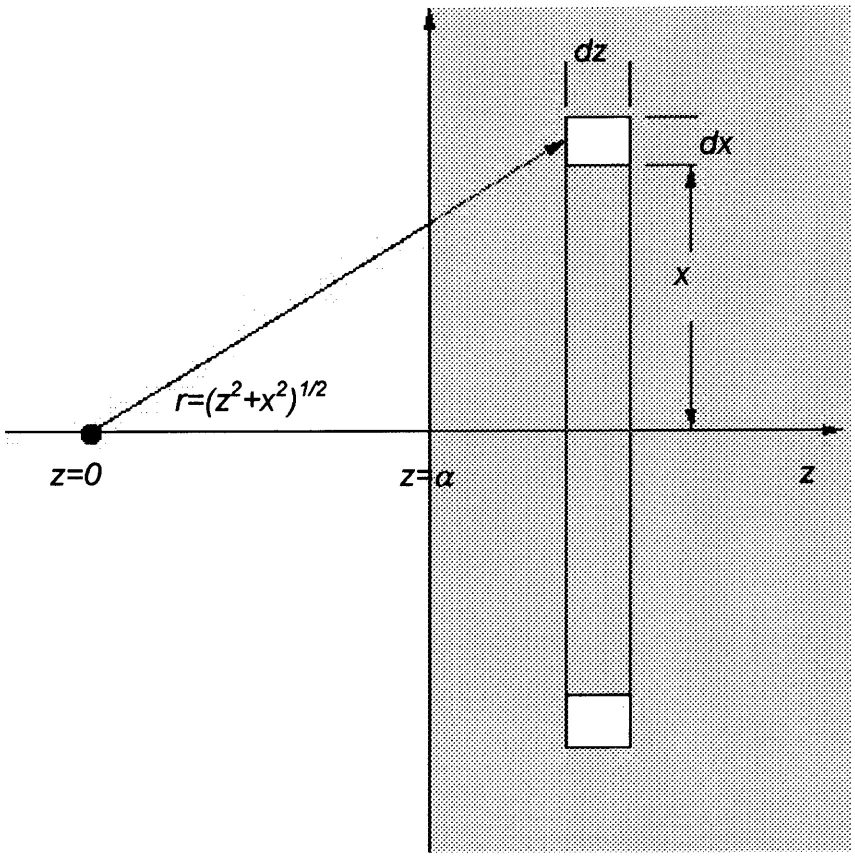


Figure 5.2 Integration between a molecule located at z = 0, and a wall beginning at z = α (Israelachvili, 1992).

The first situation that will be integrated is that of a single molecule near a wall composed of like molecules, as illustrated in Figure 5.2. The distance between the molecules is denoted by *r*. The element that will be integrated is a ring, of radius *x*, thicknesses of dx and dz, and a volume of *2πx dx dz*. The number of molecules per unit volume is denoted by ρ, therefore, the number of molecules in the ring will be $2\pi\rho$x dx dz. The integral is as follows:

$$w(r) = 2\pi\rho C \int_{z=\alpha}^{z=\infty} dz \int_{x=0}^{x=\infty} \frac{xdx}{\left(z^2 + x^2\right)^{\frac{n}{2}}} \tag{5.9}$$

This integral can be found in integral tables (Selby, 1969), and after the first integration the result is:

$$w(r) = \frac{2\pi\rho C}{(n-2)} \int_{z=\alpha}^{z=\infty} \frac{dz}{z^{n-2}} \qquad \text{for } n > 2 \tag{5.10}$$

Finally, the result is:

$$w(r) = \frac{2\pi\rho C}{(n-2)(n-3)\alpha^{n-3}} \qquad \text{for } n > 3 \tag{5.11}$$

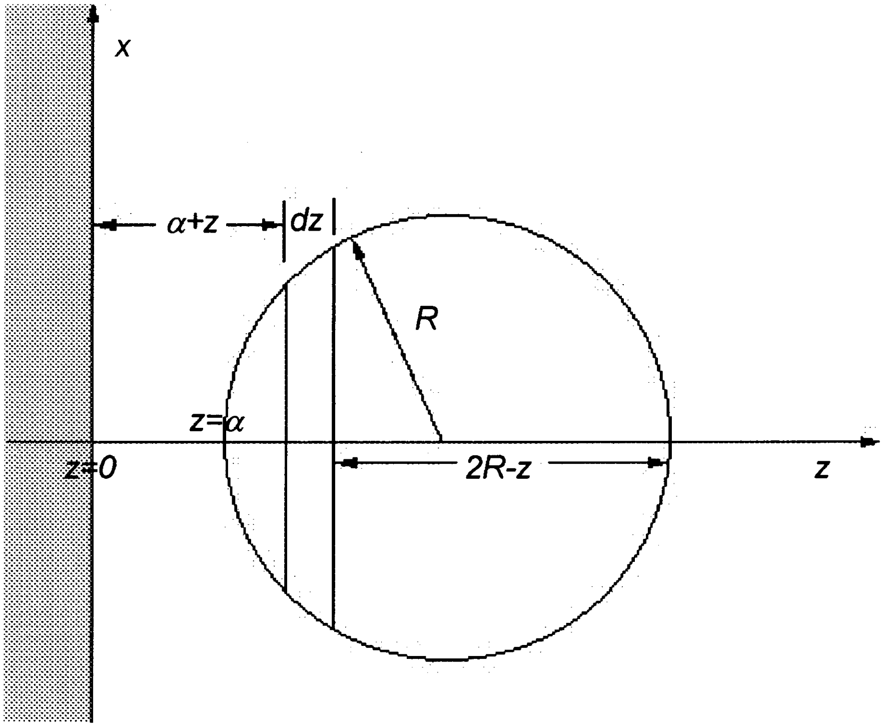


Figure 5.3 Integration between a particle beginning at z = α and a wall beginning at z = 0 (Israelachvili, 1992).

This result can now be used to find the interaction potential for an interaction between a sphere and a flat plate. This situation is shown in Figure

5.3. The form of the potential function is the same as in Equation 5.8. The interaction potential between a molecule and a flat plate is given in Equation 5.11. This potential, when integrated over a sphere, will give the interaction potential between a sphere and flat plate. Using the chord theorem, from Equation 5.7, to give the interaction area, and integrating the potential across the sphere, the equation becomes:

$$w(r) = \frac{2\pi^2\rho^2 C}{(n-2)(n-3)} \int_{z=0}^{z=2R} \frac{(2R-z)z}{(\alpha+z)^{n-3}} dz \tag{5.12}$$

Restricting this to the situation where R is much larger than the interaction area, the equation may be simplified as follows:

$$w(r) = \frac{2\pi^2\rho^2 C}{(n-2)(n-3)} \int_{z=0}^{z=2R} \frac{2Rz}{(\alpha+z)^{n-3}} dz \tag{5.13}$$

Finally, the solution to the integral is as follows (Selby, 1969):

$$w(r) = \frac{4\pi^2\rho^2 CR}{(n-2)(n-3)(n-4)(n-5)\alpha^{n-5}} \quad \text{for n> 5} \tag{5.14}$$

Next, the integration will be performed for the situation of two flat plates. Obviously, the interaction potential between two infinite plates will be infinite, therefore, it is necessary to consider the interaction potential per unit surface area. The interaction will be found for a flat surface of unit area with another flat surface of infinite area. The situation is illustrated in Figure 5.4. The area that will be integrated is a thin sheet of unit area and a thickness of *dz*. The interaction energy can then be found from Equation 5.11. The interaction energy is:

$$w(r) = \frac{2\pi\rho C(\rho dz)}{(n-2)(n-3)\alpha^{n-3}} \tag{5.15}$$

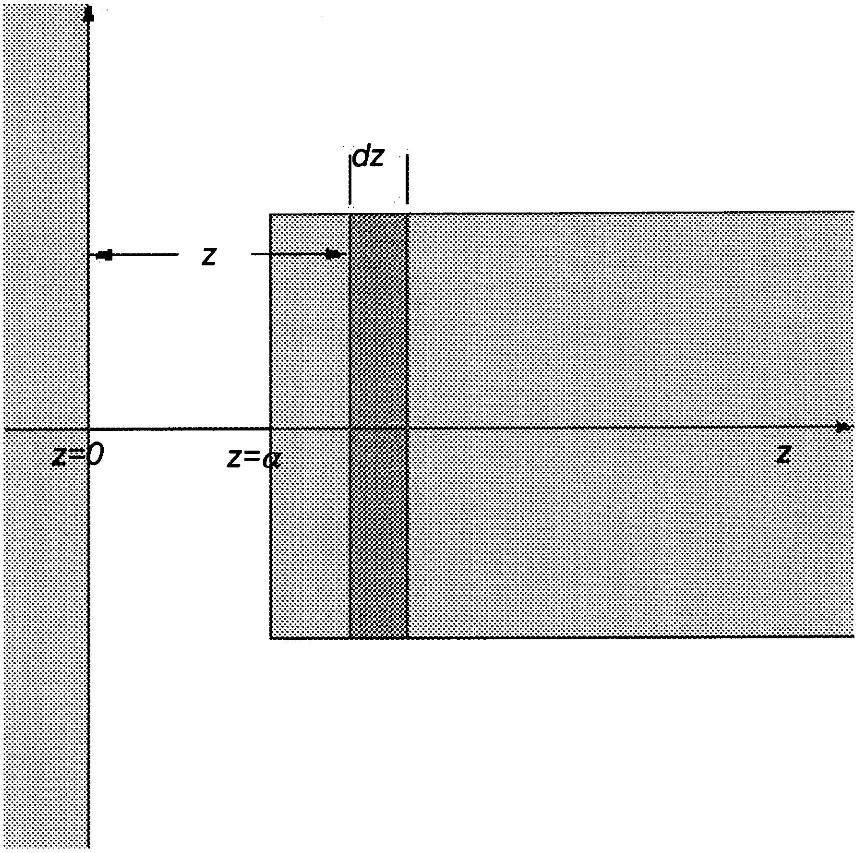


Figure 5.4 Integration between two walls (Israelachvili, 1992).

The integral then becomes:

$$w(r) = \frac{2\pi\rho^2 C}{(n-2)(n-3)} \int_{z=\alpha}^{z=\infty} \frac{dz}{z^{n-3}} \tag{5.16}$$

That can be solved for the interaction potential as follows:

$$w(r) = \frac{2\pi\rho^2 C}{(n-2)(n-3)(n-4)\alpha^{n-4}} \quad \text{for n > 4} \tag{5.17}$$

Now the forces caused by the interaction need to be found. The first case for which this will be done is the sphere and the flat plate. First, the interaction potential V_{ss}, given in Equation 5.1, needs to be substituted into the solution to the integration, given in Equation 5.14. The resulting equation, which expresses the potential between the particles is:

$$U_m = \frac{\pi^2 C_{ss} \rho^2 R}{6\alpha}\left[-1 + \frac{r_{ss}^6}{420\alpha^6}\right] \tag{5.18}$$

The Hamaker constant can be defined as follows:

$$A = \pi^2 \rho^2 C_{ss} \tag{5.19}$$

This allows Equation 5.18 to be simplified.

$$U_m = \frac{AR}{6\alpha}\left[-1 + \frac{r_{ss}^6}{420\alpha^6}\right] \tag{5.20}$$

The force between the plate and the sphere is found by differentiating Equation 5.20 with respect to the distance separating them, according to Equation 5.3.

$$F = \frac{\partial U_m}{\partial \alpha} = \frac{AR}{6\alpha^2}\left[1 - \frac{r_{ss}^6}{60\alpha^6}\right] \tag{5.21}$$

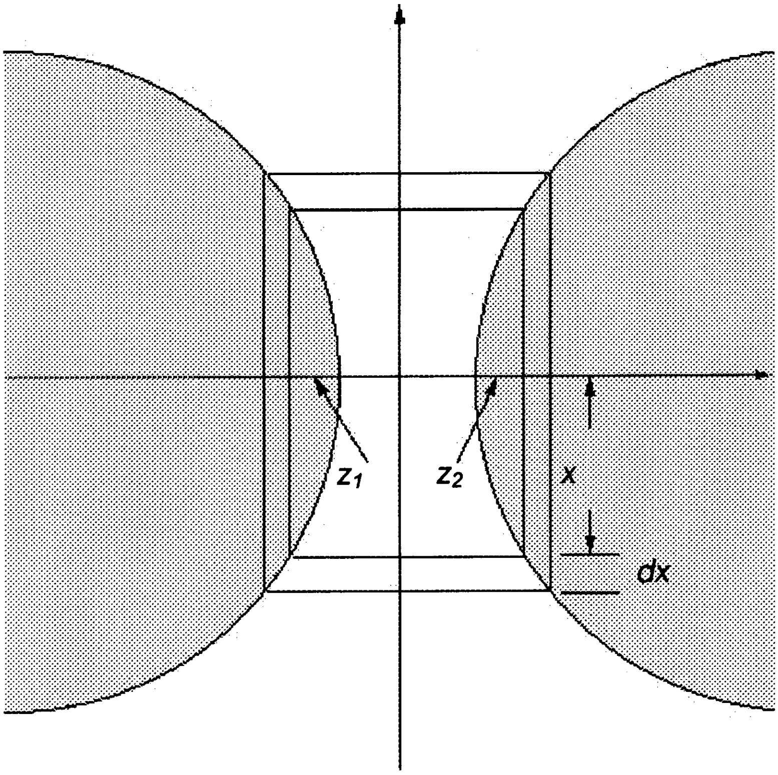


Figure 5.5 The Derjaguin approximation (Israelachvili, 1992).

The other relationship desired is the force between two spheres. This is obtained through the use of the Derjaguin approximation (Derjaguin, 1934, cited in Israelachvili, 1992). This approximation is obtained as follows, and is illustrated in Figure 5.5. Assuming two spheres, of diameter R_1 and R_2, are separated by a distance α, which is much smaller than the radius of both particles, the force between the particles can be found by integrating the force between circular areas at a distance of $Z = \alpha + z_1 + z_2$ away from each other. The areas that will be integrated are circles of dimension $2\pi x\ dx$, and are assumed to be flat. The assumption that the area is flat allows for the force between the spheres to be written in terms of the force between flat plates.

The force can be given by:

$$F(\alpha)_{spheres} = \int_{Z=\alpha}^{Z=\infty} 2\pi x dx f(Z)_{plates} \tag{5.22}$$

From the chord theorem:

$$x^2 \approx 2R_1 z_1 = 2R_2 z_2 \tag{5.23}$$

This relation can be used to equate dx to dZ.

$$Z = \alpha + z_1 + z_2 = \alpha + \frac{x^2}{2}\left(\frac{1}{R_1} + \frac{1}{R_2}\right) \tag{5.24}$$

Differentiating Equation 5.24.

$$dZ = \left(\frac{1}{R_1} + \frac{1}{R_2}\right) x dx \tag{5.25}$$

Substituting Equation 5.25 into 5.22 yields:

$$F(\alpha) \approx 2\pi \left(\frac{R_1 R_2}{R_1 + R_2}\right) \int_{Z=\alpha}^{Z=\infty} f(Z) dZ \tag{5.26}$$

The remaining integral in Equation 5.26 is equal to the interaction energy between two flat plates. The equation can now be written to relate the force between two spheres to the interaction energy between two flat plates.

$$F(\alpha)_{spheres} = 2\pi\left(\frac{R_1R_2}{R_1+R_2}\right)w(\alpha)_{plates} \tag{5.27}$$

For two spheres of equal radius this reduces to:

$$F(\alpha)_{spheres} = \pi R w(\alpha) = \frac{AR}{12\alpha^2}\left(1-\frac{r_{ss}^6}{60\alpha^6}\right) \tag{5.28}$$

5.3 Adapting the Van der Waals Model for Computer Simulation

In the previous section the model for the Van der Waals force was derived for the various situations that will be encountered in a computer particle dynamics simulation. The two equations of interest are Equation 5.21, which expresses the force between a sphere and a wall, and Equation 5.28, which expresses the force between two spheres. These equations need to be incorporated into the particle dynamics code. In order to do this, the Van der Waals force model needs to be combined with a contact force model that will counteract the Van der Waals model when the particles come into contact.

A common convention is to include only the attractive portion of the Van der Waals force. This is the approach taken in most of the literature reviewed in Section 5.1. The reasoning for this is that in practice the force generally remains attractive until the particle contact forces balance it. This is caused mostly by the effects of the surface asperities, as explained below.

Although the model assumes that the interaction is happening between perfectly round spheres, in reality, the surfaces are rough and have imperfections. Therefore, there are three separate reactions combining to form the overall attractive force; the interaction between the two particles themselves, between the particles and the asperities, and between the two asperities. The result of this is that as the particles move closer together, the force between the asperities becomes repulsive and the asperities deform, and that deformation brings the forces into balance (Rietema, 1991; Yu et al., 2003). Since the asperity interaction is often dominant, this gives rise to another common convention of substituting an asperity diameter for the particle diameter whenever it appears in the model (Israelachvili, 1992; Rietema, 1991; Yang et al., 2000; Yu et al., 2003).

One point that can be easily observed about the equations is that there is a singularity when the particles touch. This singularity is generally handled by picking a minimum separation distance and not allowing the particles to move closer than the separation distance. A contact force model is then used to calculate the forces as the particles press against the separation distance. In many cases however, the separation distance is chosen somewhat arbitrarily (Israelachvili, 1992; Yang et al., 2000).

Rietema, however, suggests using the repulsive portion of the Van der Waals force model to find the minimum separation distance. This provides an estimate that is in the general range of other values used (Rietema, 1991). Generally, the chosen minimum separation distance is between 0.1 and 1 nm (Israelachvili, 1992; Xie, 1997; Yang et al., 2000).

This also helps to solve the problem of adding the Van der Waals force model to the code in this project. The normal force model that will be used with the Van der Waals force model is the partially latching spring model, given in Equation 2.1. This model has a constant coefficient of restitution, and as a result, it uses different spring constants for loading and unloading. However, the model will not properly release particles under an externally applied force. This can be handled however, if the minimum separation distance is chosen to coincide with the point of zero force. In order to simplify the calculations, the Van der Waals force will be held constant at any distance less than the minimum separation distance, and the particles allowed to pass the minimum separation distance. Since this distance is generally very small in comparison to the virtual overlap that will occur between the particles during the application of the spring force, this will have very little effect on the calculations.

Therefore, the following model is proposed for the total normal force on the particle pairs, with R representing the radius of the asperities:

$$F(\alpha)=\begin{cases}\alpha>\alpha_{cutof} & F(\alpha)=0\\ \alpha_{cutoff}>\alpha>\alpha_{\min} & \begin{cases}\text{Particle - Particle} & F(\alpha)=\dfrac{AR}{12\alpha^2}\left(1-\dfrac{r_{ss}^6}{60\alpha^6}\right)\\ \text{Particle - Wall} & F(\alpha)=\dfrac{AR}{6\alpha^2}\left(1-\dfrac{r_{ss}^6}{60\alpha^6}\right)\end{cases}\\ \alpha_{\min}\ge\alpha\ge 0 & F(\alpha)=0\\ \alpha<0 & \begin{cases}\text{For loading} & F(\alpha)=-K_1\alpha\\ \text{For unloading} & F(\alpha)=-K_2(\alpha-\alpha_0)\end{cases}\end{cases} \tag{5.29}$$

5.4 Results of the Addition of the Van der Waals Model

The results of adding the Van der Waals force model were illustrated by running a simulation where two particles collided and separated once. In order to simplify the situation, and isolate the effects of the collision, the effect of gravity was not included in this simulation. The material constants and simulation parameters are shown in Table 5.1.

Table 5.1 Parameters Used in Test of Van der Waals Model

Parameter	Value
Time step	2.82×10^{-10} s
Spring constant	1×10^{-7} N/m
Coefficient of restitution	.95
Mass of particle with unit radius.	5026 kg
Coefficient of friction	.1
Hamaker constant	1×10^{-18} J
Acceleration of gravity	0
Radius	3.175×10^{-5} m
Initial velocity	0

The first three graphs presented, Figures 5.6-8, show the force calculated by the Van der Waals force model during the collision of two particles versus the separation distance α. These three graphs show the same information, however, the scale of the horizontal axis is reduced by a factor of ten on each succeeding graph. All of these values were calculated by the particle dynamics code according to the combined model shown in Equation 5.29.

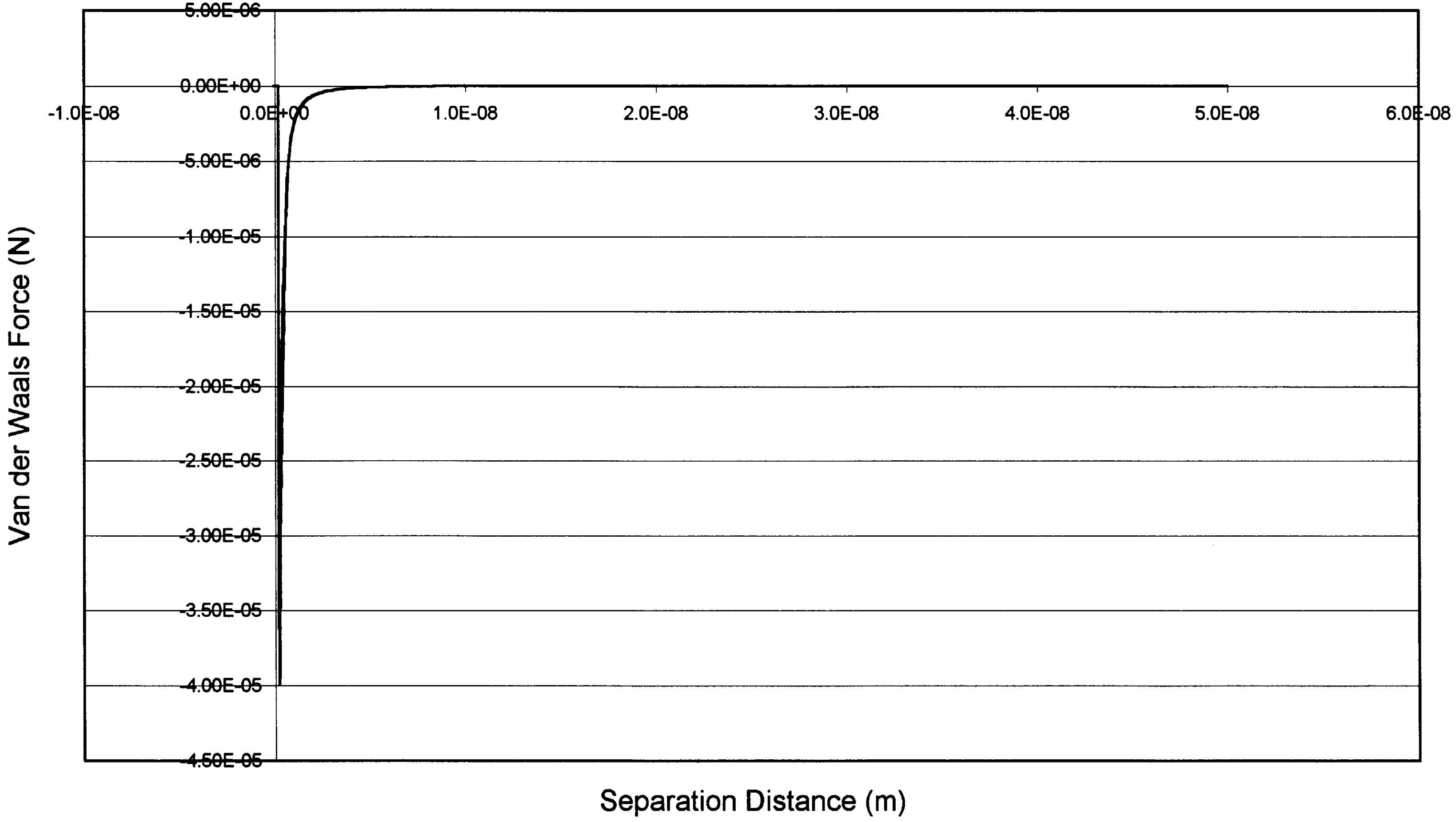


Figure 5.6 Van der Waals force (N) vs. distance (m).

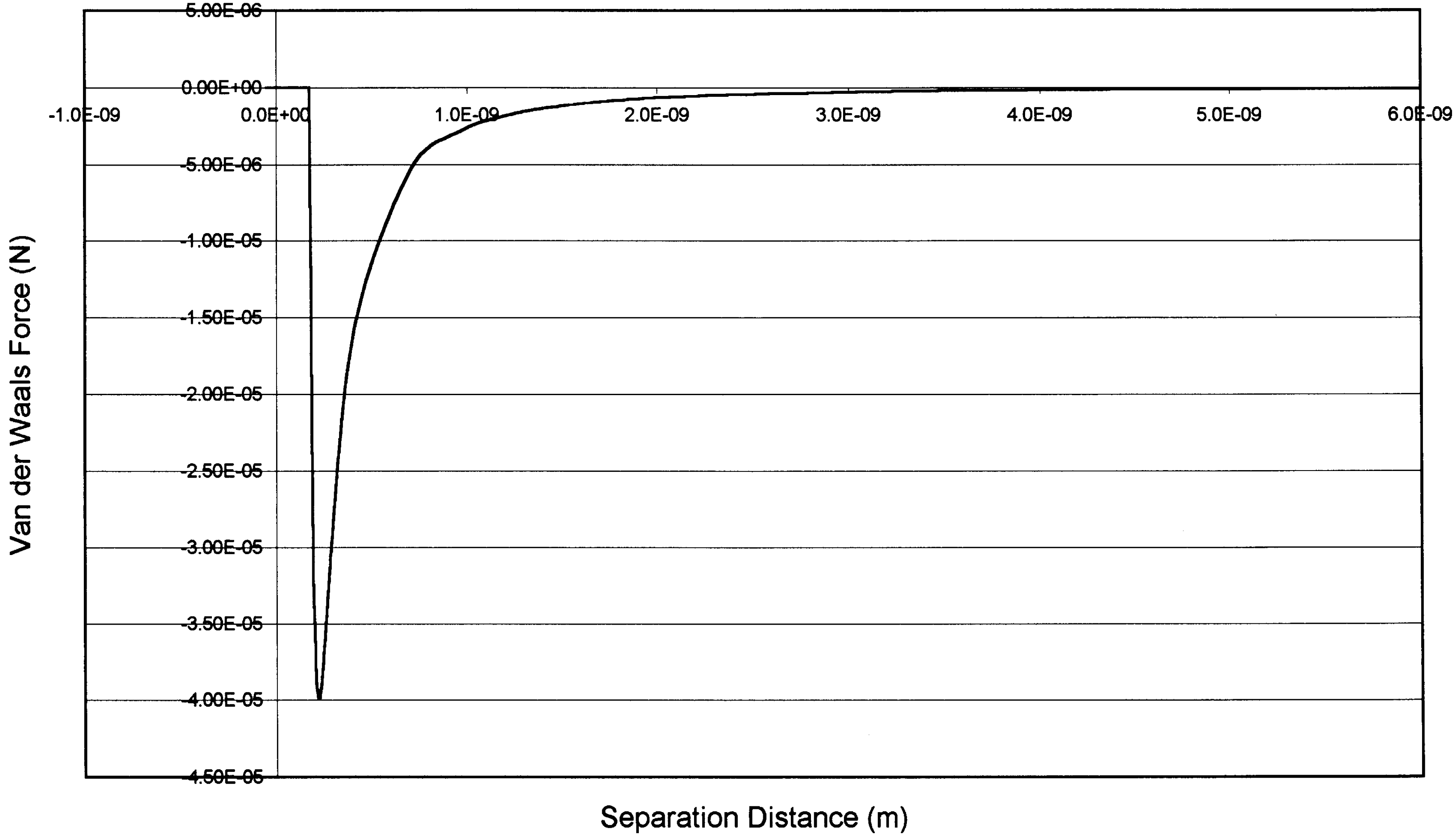


Figure 5.7 Van der Waals force (N) vs. distance (m).

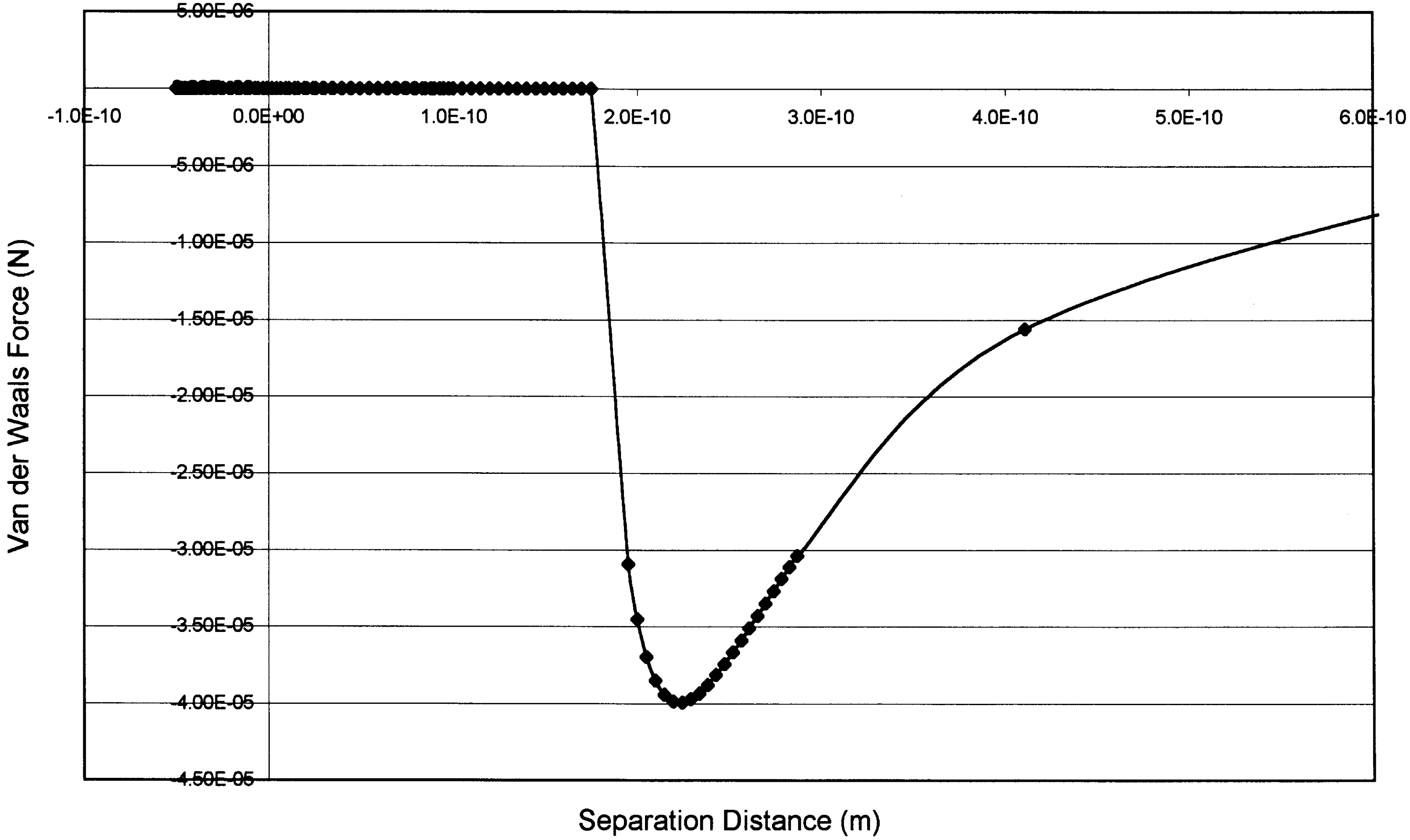


Figure 5.8 Van der Waals force (N) vs. distance (m).

Figure 5.9 shows the force calculated by the simulation, calculated according to the model given in Equation 5.29, along with the theoretical values calculated by the theory given in Equation 5.28. Figure 5.10 shows values calculated according to the model given in Equation 5.29 graphed along with the same theory as Equation 5.28, albeit neglecting the repulsive component. Neglecting the repulsive component of Equation 5.28, and choosing a minimum separation distance based on experimental data is another common way of developing a Van der Waals force model. The values in Figures 5.9 and 5.10 were graphed against the distance given as a percentage of the particle radius.

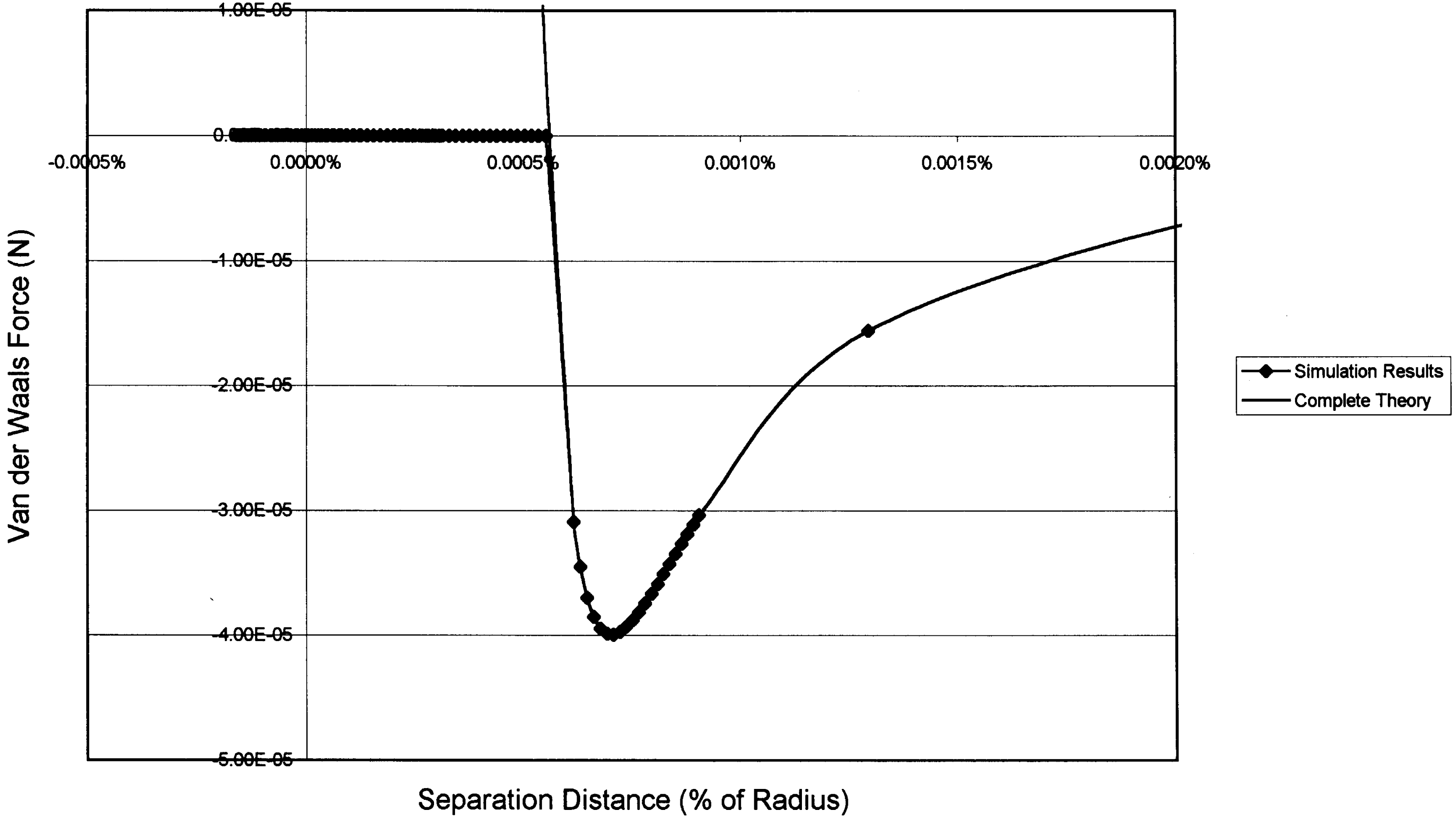


Figure 5.9 Van der Waals force (N) vs. distance (% of radius); points calculated by the code are shown along with the Van der Waals theory that includes both attractive and repulsive components.

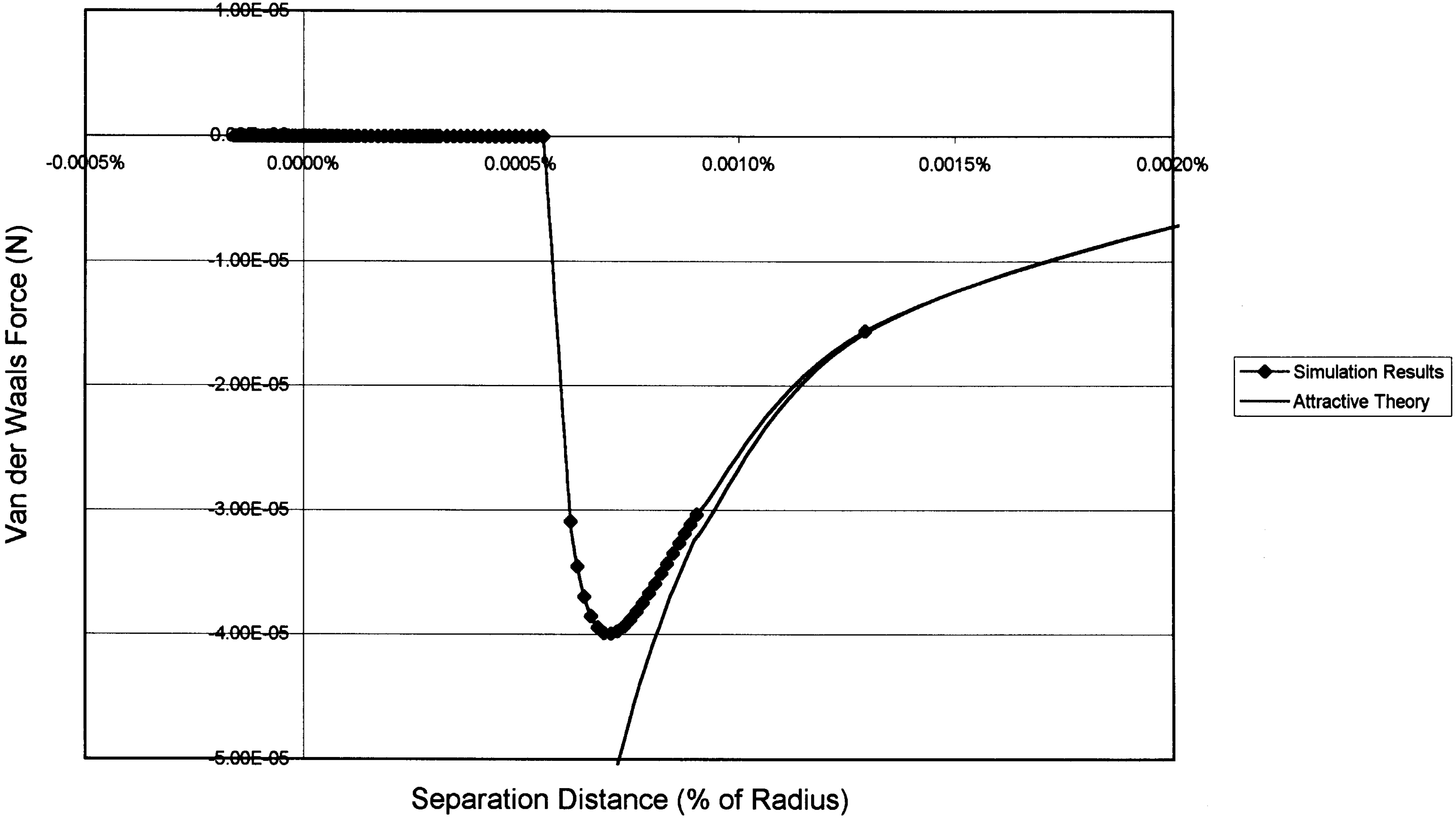


Figure 5.10 Van der Waals force (N) vs. distance (% of radius); points calculated by the code are shown along with the Van der Waals theory that includes only the attractive component.

The spring force is shown in Figure 5.11. The effect of the different spring constants for loading and unloading can be clearly observed in the two distinct curves on the graph. In this figure the separation distance is given as a negative value as there is overlap. Figure 5.12 shows the total force graphed against the separation distance. The contact force has a much larger magnitude, but it acts over a smaller distance. Figure 5.13 shows the total force versus the collision time, and as expected, the curve is roughly symmetric around the peak value of the force.

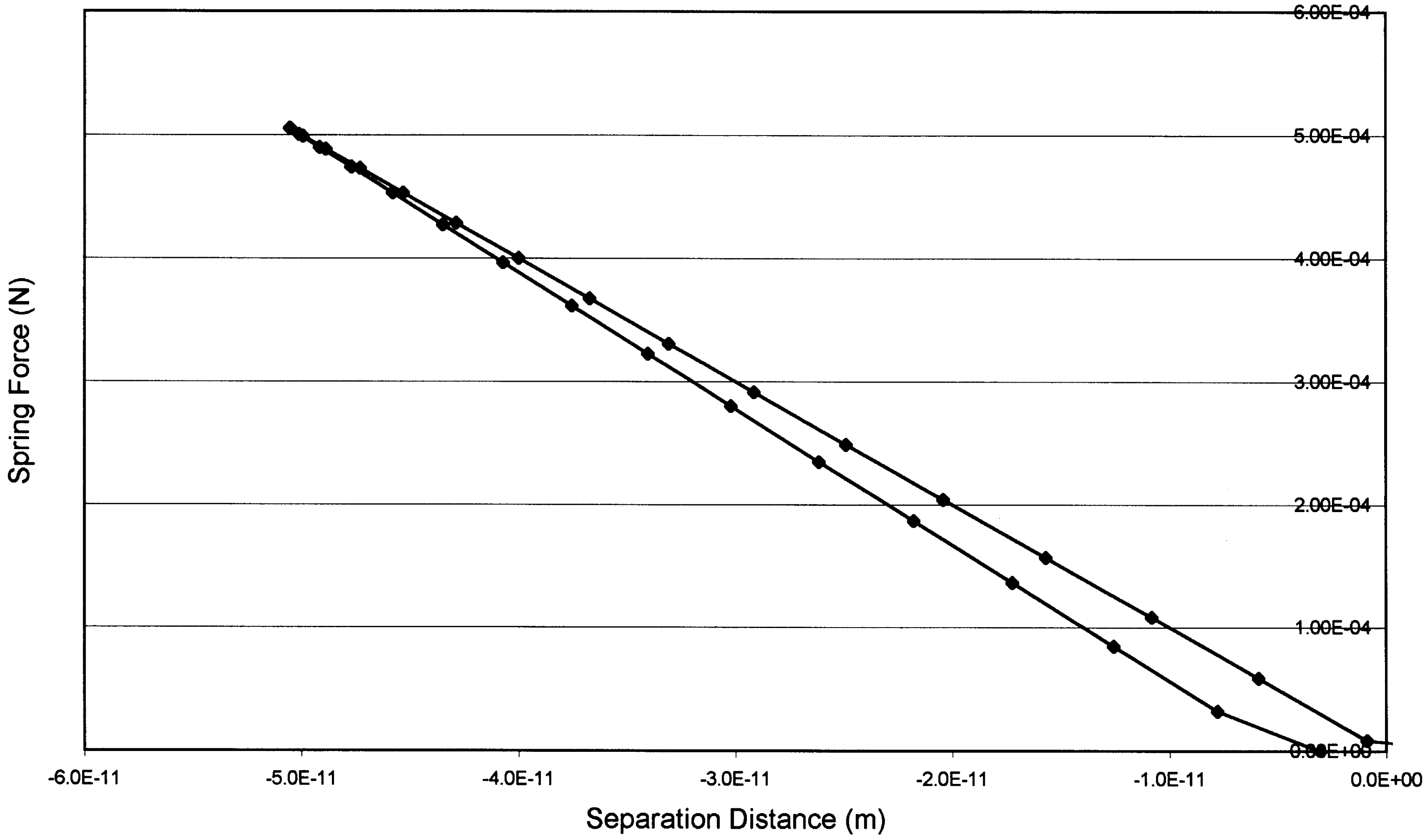


Figure 5.11 Spring force (N) vs. separation distance (m).

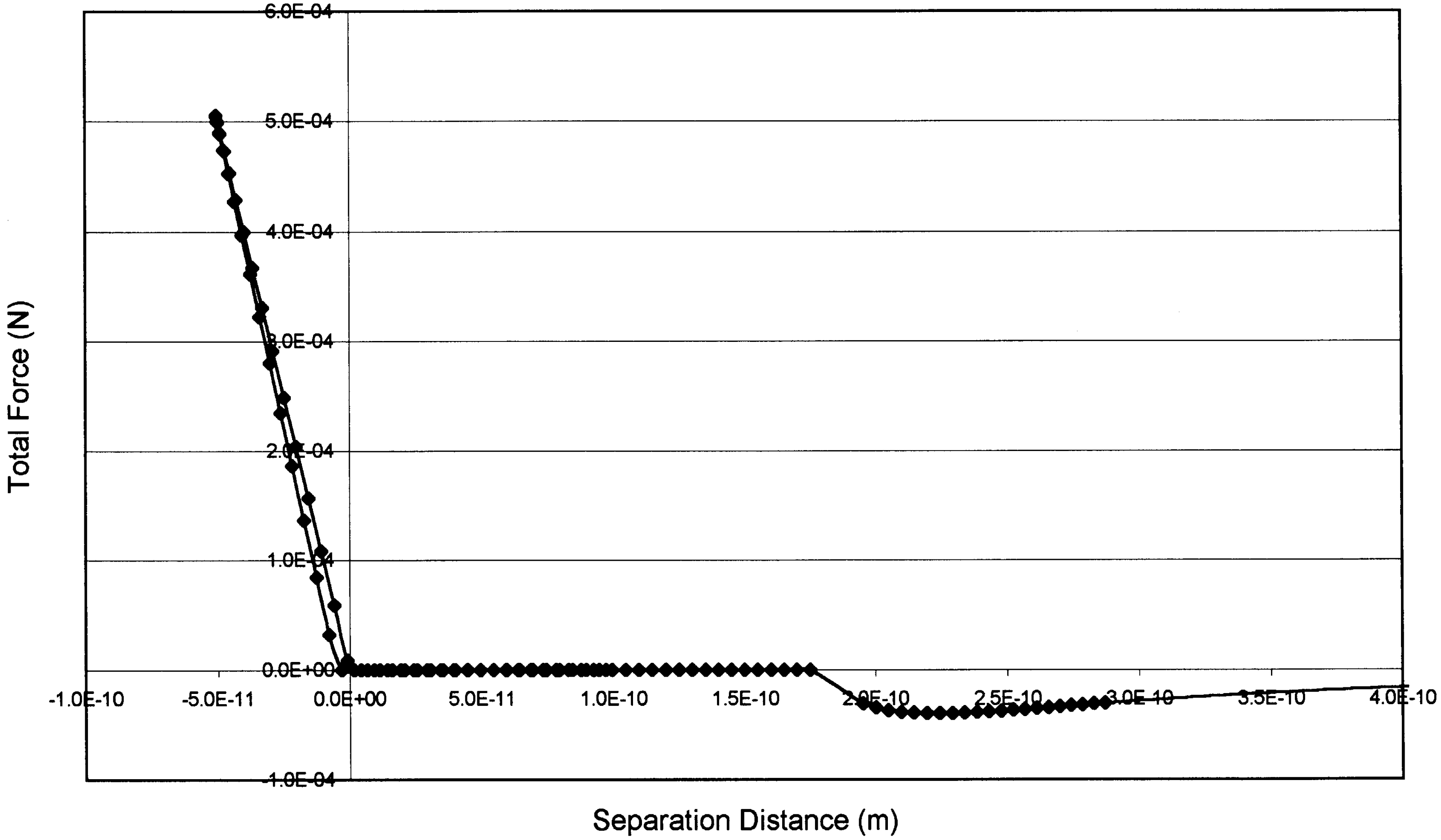


Figure 5.12 Total force (N) vs. separation distance (m) for one collision.

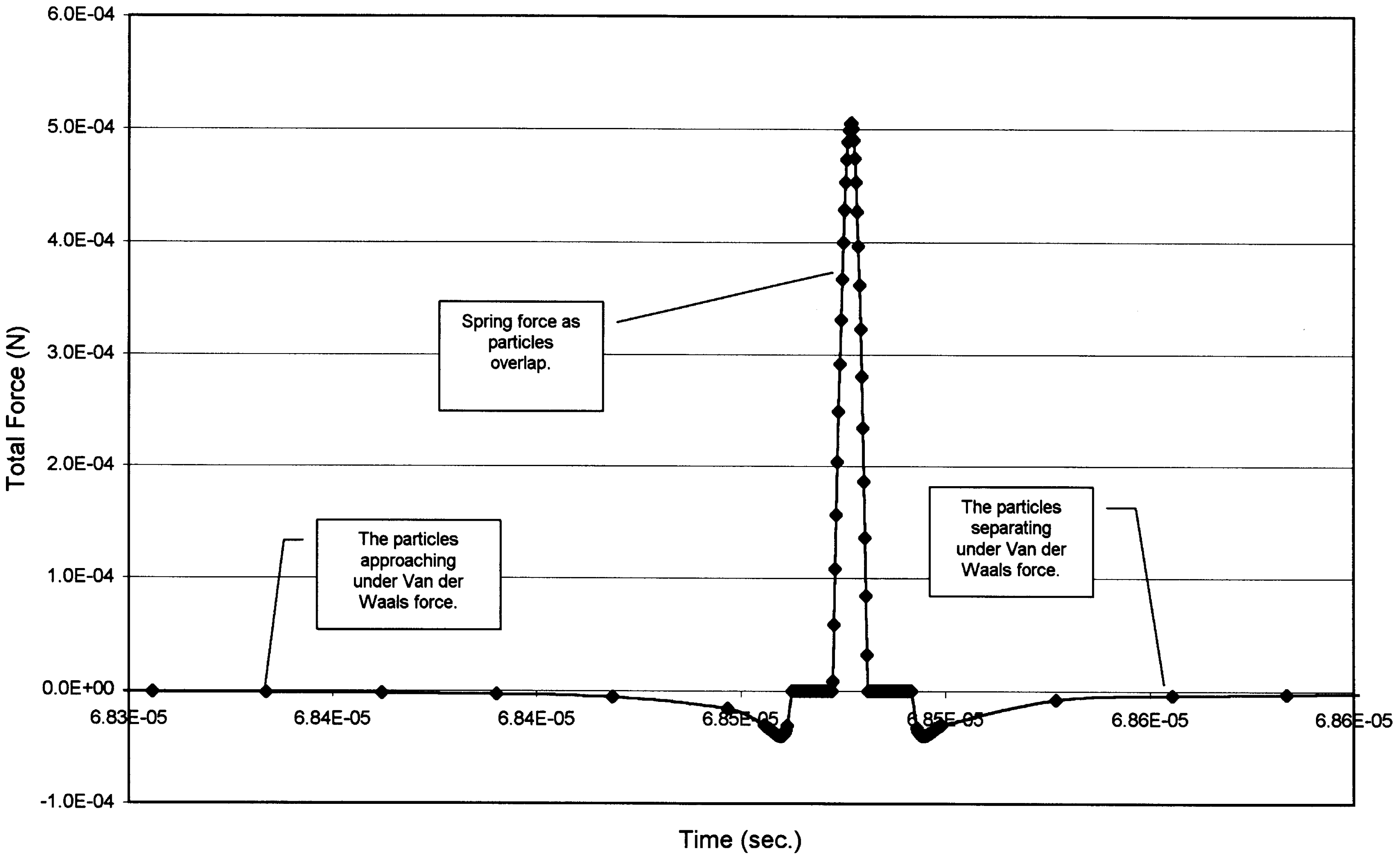


Figure 5.13 Total force (N) vs. time (sec.) for one collision.

In order to verify that the simulation is generating realistic results, given the model proposed in Equation 5.29, the simulation was allowed to continue for some time after the first collision. Since the particles had no initial velocity and the spring model dissipates energy, it would be expected that the particles would be unable to escape the Van der Waals attraction. Therefore, they would begin to oscillate with the force decreasing in magnitude with each oscillation. This behavior was exhibited, and is graphed in Figure 5.14. This is a graph of total force versus time. Since the magnitude of the spring force is much greater than that of the Van der Waals force, the only quantity visible on the graph is the peak value of the spring force, and this value does decrease steadily with time.

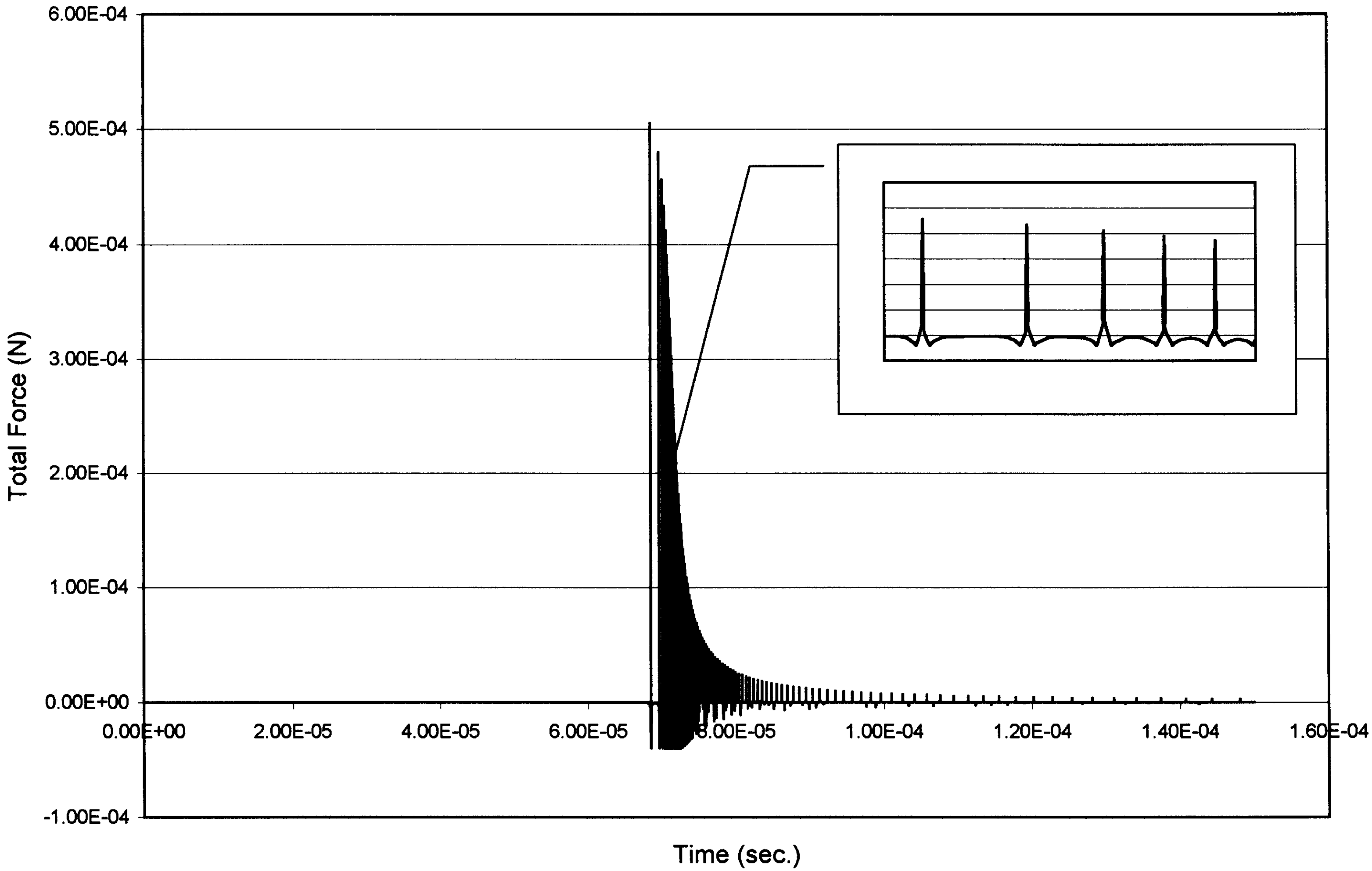
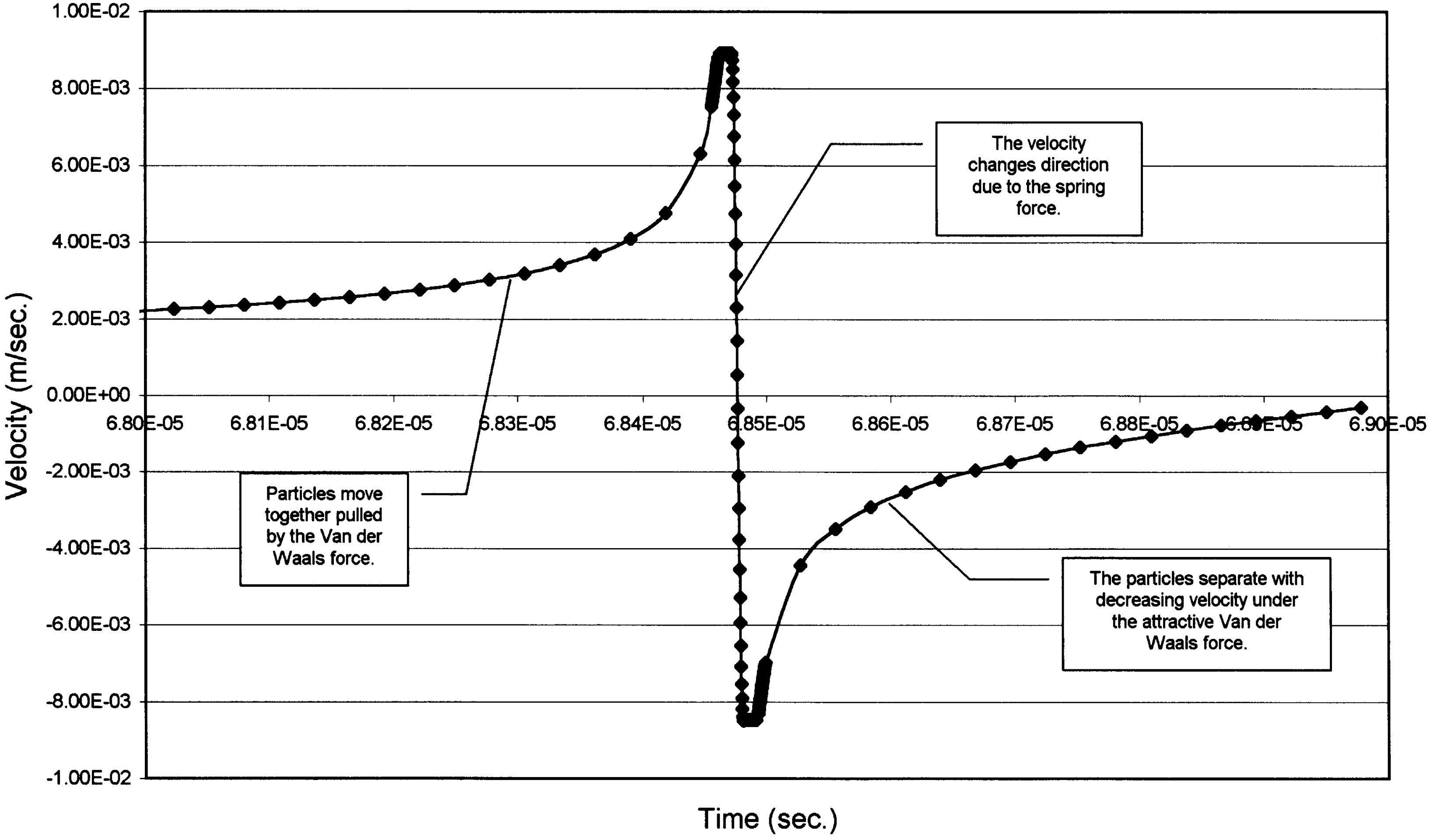


Figure 5.14 Total force (N) vs. time (sec.) for a large number of collisions, the inset shows the curve detail.

The velocity of the particles was also studied. In the test, the particles were started with no velocity, but were under a slight Van der Waals attractive force. Since the particles that collided were under equal but opposite forces the entire time their absolute velocities were also of equal magnitude, but of opposite direction. Therefore, in the following graphs the velocity of only one of the particles is shown.

Figure 5.15 graphs the velocity versus time during one collision. First, the particles are moving together under an attractive Van der Waals force. Then, when the spring force activates, the particles rapidly slow to a stop, reverse direction, and begin moving rapidly away from one another. Finally, the particles begin to slow, as they are separating under an attractive Van der Waals force.

Figure 5.16 shows the velocity profile throughout multiple collisions. Since the spring force dissipates energy, the particles move away from one another with less kinetic energy than they collided with. Also, since all of the original kinetic energy was from the Van der Waals force, the particles should not be able to escape the attractive Van der Waals force. This behavior is exhibited in Figure 5.16. It can be seen from the inset graph that the particles eventually stop moving away from one another, and begin again to move towards each other, albeit with less energy each oscillation. The main graph shows that the oscillations eventually decrease to a negligible value.



The velocity changes direction due to the spring force.

Particles move together pulled by the Van der Waals force.

The particles separate with decreasing velocity under the attractive Van der Waals force.

Figure 5.15 Velocity (m/s) vs. time (sec) for one collision.

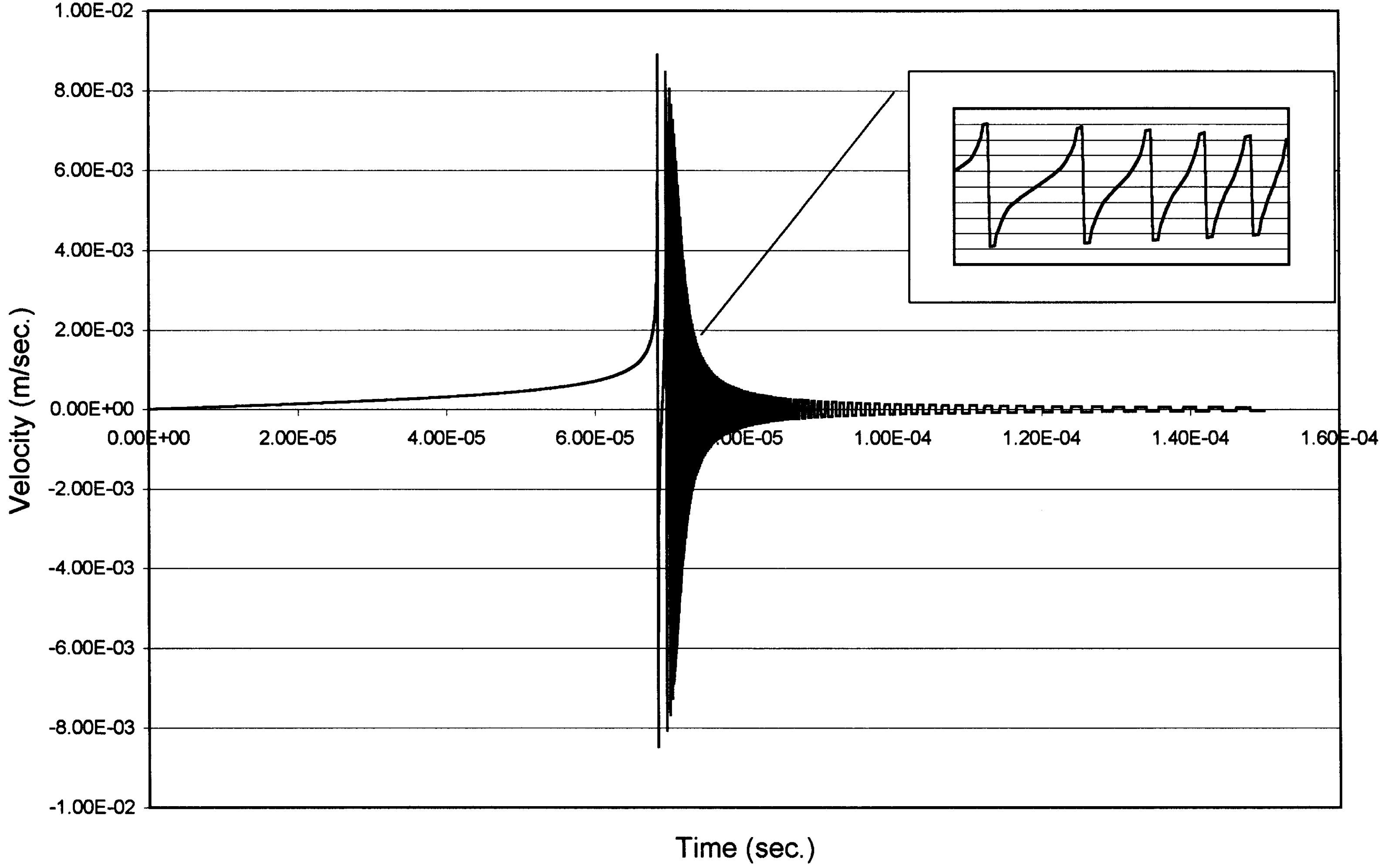


Figure 5.16 Velocity (m/s) vs. time (sec) for a large number of collisions, the inset shows the curve detail.

CHAPTER 6

CONCLUSIONS AND FURTHER WORK

6.1 Chain-Cell Search Method

In a particle dynamics simulation, the amount of time spent finding colliding particle pairs can often be the most computationally expensive part of the simulation. Therefore, various methods can be applied to reduce the time spent on this task. One of the first tools for reducing the amount of time spent locating near neighbors was introduced by Verlet, and this method is known as the Verlet list (Verlet, 1967). Another method shown by Allen and Tildesley is called the chain-cell search method (Allen & Tildesley, 1987).

This project studied the combination of both of the above mentioned methods. It was found that a substantial time savings could be achieved using the chain-cell search method combined with the Verlet list compared with a simulation using only a Verlet list. Using only the Verlet list, the computational time spent updating the list of neighbors is proportional to the number of particles squared (N^2). The computational time savings is realized as a result of lowering the power to which N is raised. In this project the time to update was found to be dependent on $N^{1.2}$.

Therefore, there are a number of factors to keep in mind when considering use of the chain-cell search method. First, it only reduces the time spent updating the neighbor list. Therefore, if the time spent updating the neighbor list is small then the time savings will be minimal. Second, the size chain-cell mesh must be scaled to the largest particle in the system. For a system with many different sized particles, the effectiveness can be reduced, as the mesh size must be increased to fit the largest particle. This difficulty can be mitigated for systems with only a few large particles by ignoring large particles when sizing the mesh, and performing a separate search over all particles checking for interaction with only the large particles. This second search will scale proportionally to $(N_{large})(N_p)$, and it will, therefore, only help when there are a very small number of large particles. Finally, linked-lists can grow quite large, therefore the amount of available memory, rather than computational time, may become a limiting factor when considering how large to make a simulation.

One further technique that may result in additional savings, which was not explored in this work, is adapting the chain-cell method to multiprocessing. Symmetric multiprocessing (SMP) allows for the execution of a computer program on several processors simultaneously. The chain-cell search method is adaptable to multiprocessing (Sullivan, Mountain, & O'Connell, 1985).

There are a number of significant factors to consider when adapting a program for multiprocessing. First, it makes little sense to adapt a code for SMP unless significant portions of the entire code can be adapted for SMP. Therefore, unless the force model and integration method can also be processed in parallel, the search method should not be adapted for SMP. This however, is generally not a problem as most force and integration methods are adaptable to SMP. Second, while SMP reduces the time to execute a program by using multiple processors, it increases the absolute computational time, because there is significant additional overhead involved in the job of dividing the execution between the processors. Finally, adapting a code for SMP increases the complexity, which increases the likelihood of introducing errors into the code (D'Azevedo, 1994).

6.2 Van der Waals Force Model

A model for calculating the Van der Waals force between micrometer-sized particles was introduced and added to a particle dynamics code. It was shown that the particle dynamics code was accurately simulating the force model introduced. A substantial amount of additional work is needed to validate the model before useful experimentation with the model can begin.

In developing the model, the principles explained by Rietema were followed. These included the assumption that the Van der Waals force is additive, and therefore, it can be integrated over a particle to obtain a resultant force between particles (Rietema, 1991). In developing the computer model the work of Yang, Zou, and Yu was followed. Their study simulated the packing of particles using a Van der Waals model with a non-linear spring-dashpot collision model. In this project the partially latching spring model was substituted for the spring and the dashpot (Yang et al., 2000).

Therefore, this model must be tested to show that it gives reasonable predictions concerning the packing of particles. There have been theoretical studies done on xerographic toner with continuum models that will likely be useful in validating the model. The behavior exhibited by the xerographic toner is similar to that of wet sand, in that the cohesion between the grains can be dominant over the effect of gravity (Castellanos et al., 2000).

Since there is a very large deviation in behavior in particles that are in the micrometer size range, it is unlikely that exact equations can be formulated to predict the packing behavior. This deviation has been exhibited experimentally in studies on particle pickup velocity (Hayden et al., 2003). However, particle packing in this size region has not been studied very extensively, and it is likely that numerical simulations will be useful in enhancing the understanding of particle dynamics.

REFERENCES

Allen, M. P., & Tildesley, D. J. (1987). *Computer Simulation of Liquids*. Oxford: Clarendon Press.

Allen, M. P., & Tildesley, J. (1981). *Routines to Construct and Use Cell Linked-List Method*. Available: http://www.ccl.net/cca/software/SOURCES/FORTRAN/allen-tildesley-book/f.20.shtml [February 8, 2003].

Arunachalam, V., Marlow, W. H., & Lu, J. X. (1998). Development of a picture of the van der Waals interaction energy between clusters of nanometer-range particles. *Physical Review E, 58*(3), 3451-3457.

Becker, O. M., MacKerell, A. D., Jr., Roux, B., & Watanabe, M. (Eds.). (2001). *Computational Biochemistry and Biophysics*. New York: Marcel Dekker, Inc.

Beer, F. P., & Johnston, E. R. J. (1997). *Vector mechanics for engineers: dynamics*. New York: WCB McGraw-Hill.

Castellanos, A., Ramos, A., Valverde, J. M., & Watson, P. K. (2000). Avalanches in fine, cohesive powders. *Physical Review E, 62*(5), 6851-6860.

D'Azevedo, E. F. (1994). *A New Shared-Memory Programming Paradigm for Molecular Dynamics Simulations on the Intel Paragon* (ORNL/TM-12890). Oak Ridge, TN: Oak Ridge National Laboratory.

Derjaguin, B. V. (1934). *Kolloid Zeits, 69*, 155-164.

Duran, J. (2000). *Sands, Powders, and Grains. An Introduction to the Physics of Granular Materials*. New York: Springer.

Forsyth, A. J., & Rhodes, M. J. (2000). A Simple Model Incorporating the Effects of Deformation and Asperities into the van der Waals Force for Macroscopic Spherical Solid Particles. *Journal of Colloid and Interface Science, 223*(1), 133-138.

Haile, J. M. (1992). *Molecular Dynamics Simulation: Elementary Methods.* New York: John Wiley & Sons, Inc.

Hamaker, H. C. (1937). *Physica, 4*, 1058-1072.

Hayden, K. S., Park, K., & Curtis, J. S. (2003). Effect of particle characteristics on particle pickup velocity. *Powder Technology, 131*(1), 7-14.

Hockney, R. W., & Eastwood, J. W. (1999). *Computer simulation using particles.* Philadelphia: Institute of Physics Publishing.

Israelachvili, J. N. (1992). *Intermolecular & Surface Forces.* New York: Harcourt Brace & Company, Publishers.

Kim, H.-J. (1992). *Particle Dynamics Modeling of Boundary Effects in Granular Couette Flow.* , New Jersey Institute of Technology, Newark, New Jersey.

O'Connor, J. J., & Robertson, E. F. (2003). *Charles Augustin de Coulomb,* [internet]. Available: http://www-gap.dcs.st-and.ac.uk/~history/ Mathematicians/Coulomb.html [Charles Augustin de Coulomb, March 27, 2003].

Rietema, K. (1991). *The Dynamics of Fine Powders.* New York: Elsevier Applied Science.

Roberts, A. W. (1998). Particle Technology - Reflections and Horizons: An Engineering Perspective. *Trans IChemE, 76*(A), 775.

Rosato, A. D. (2001). Linked List Logic. Class notes for ME 624 Microlevel Modeling in Particle Technology .

Schinner, A. (1999). Fast Algorithms for the Simulation of Polygonal Particles. *Granular Matter, 2*(1), 35-43.

Selby, S. M. (Ed.). (1969). *CRC Standard Mathematical Tables: Seventeenth Edition* (17th ed.). Cleveland, Ohio: The Chemical Rubber Co.

Sullivan, F., Mountain, R. D., & O'Connell, J. (1985). Molecular Dynamics on Vector Computers. *Journal of Computational Physics, 61*(1), 138-153.

Underwood, J. R. (1998). *Ralph A. Bagnold: Soldier, Explorer, and Scientist.* Available: www.weru.ksu.edu/symposium/proceedings/underwoo.pdf [2002, September 14].

Verlet, L. (1967). Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review, 159*(1), 98-103.

Walton, O. R. (1984). Application of Molecular Dynamics to Macroscopic Particles. *International Journal of Engineering Science, 22*(8-10), 1097-1107.

Walton, O. R. (1985). Linked-list near-neighbor arrays in 2DSHEAR. Unpublished notes.

Walton, O. R. (1991). *Numerical simulation of inclined chute flows of monodisperse, inelastic, frictional spheres.* Paper presented at the Second U.S.-Japan Seminar on Micromechanics of granular Materials, Potsdam, NY.

Xie, H.-Y. (1997). The role of interparticle forces in the fluidization of fine particles. *Powder Technology, 94*(2), 99-108.

Yang, R. Y., Zou, R. P., & Yu, A. B. (2000). Computer simulations of the packing of fine particles. *Physical Review E, 62*(3), 3900-3908.

Yu, A. B., Feng, C. L., Zou, R. P., & Yang, R. Y. (2003). On the relationship between porosity and interparticle forces. *Powder Technology, 130*(1-3), 70-76.