8-1-2010

Event-driven molecular dynamics simulations of protein mixtures

Marek Cyran

Event-Driven Molecular Dynamics Simulations of Protein Mixtures

by

Marek A. Cyran

Submitted to the School of Mathematical Sciences

in partial fulfillment of the requirements for the degree of

Masters of Science

at the

ROCHESTER INSTITUTE OF TECHNOLOGY

August 2010

Author .....
Marek A. Cyran
School of Mathematical Sciences

Accepted by .....
Anthony A. Harkin
School of Mathematical Sciences

Accepted by .....
David S. Ross
School of Mathematical Sciences

Accepted by .....
George M. Thurston
Department of Physics

Event-Driven Molecular Dynamics Simulations of Protein Mixtures

by

Marek A. Cyran

Submitted to the School of Mathematical Sciences
on August 13, 2010, in partial fulfillment of the
requirements for the degree of
Masters of Science

Abstract

The structure of liquids is central to their thermodynamic properties and is described in a probabilistic manner. The structure is a consequence of the forces between the molecules and may be investigated with the use of many techniques. One of these techniques is the use of computer simulation, and in particular the techniques are called Monte Carlo Statistical Thermodynamic simulation, and Molecular Dynamics. In this thesis we construct a program that is capable of carrying out Event-Driven Molecular Dynamics simulation of mixtures of particles that have stepwise constant pair potential energies. We have implemented our simulation for the case of square-well particles that have a hard impenetrable core surrounded by a attractive potential well. Such mixtures are important for understanding the behavior of biological macromolecules at the high concentrations that occur in living cells. To test our implementation we have compared the resulting pair correlation functions with those that result from Monte Carlo simulations. While these pair correlation functions are in rather close agreement there remain discrepancies that remain to be resolved.

Thesis Supervisor: Anthony A. Harkin, School of Mathematical Sciences
Thesis Supervisor: David S. Ross, School of Mathematical Sciences
Thesis Supervisor: George M. Thurston, Department of Physics

Acknowledgments

Working on this project has been one of the most challenging and stimulating undertakings that I have had a chance to be a part of throughout my academic career. It could not have happened without my advisors. I would like to sincerely thank you for giving me the opportunity to work on this project, and for supporting me throughout the process of completing this thesis. I would also like to thank my parents for their constant encouragement and support, without you I would not have been in a position to participate in this interesting work.

Contents

Chapter 1

Introduction

There is no general theory of interactions between proteins suitable for application to the high concentration mixtures that occur inside living cells. An example is the formation of cataract disease, in which liquid protein mixtures of volume fraction above 20%, found in the cells of the lens of the eye, can undergo phase transitions and aggregation that scatter light and impair vision. In this thesis, through the use of event-driven Molecular Dynamics simulations, we set up a model that will be suitable for studying how the interactions of mixtures of protein molecules at high concentrations affect key thermodynamic properties that can lead to important phase transitions including liquid-liquid phase separation and crystallization, as well as other thermodynamic properties that are relevant for the control of certain biochemical reactions.

The type of simulation we have created is termed a coarse-grained molecular dynamics simulation. This approach, which considerably simplifies the potential energy between the molecules, is important for a number of reasons. First, the types of molecules that we are simulating have highly complicated structures, charge distributions, and interactions, many of which remain unknown from a quantitative point of view. Under these circumstances, the use of a coarse-grained model that incorporates key features of molecular attraction and repulsion can aid in the process of discovering quantitative constraints on these unknown potentials. Second, the number of molecules needed to model collective effects to an accuracy suitable for comparison

with experiment can be in the tens of thousands, and precludes detailed treatment of the interaction between each configuration of neighboring molecules. In order to model their interactions it is necessary to simplify the problem for several reasons. Third, for any highly complicated model, it is very difficult to determine specific origins of observed collective effects, even in simulations, due to the large number of parameters that would have to be used to create the complicated intermolecular potential. For all these reasons, square-well particles are used in our simulations because they give us a good chance at determining the features of the potential, which leads to the structure of the liquid as a whole. The square well has the additional advantage of enabling one to simply count the numbers of pairs of molecules that are undergoing any attractive interaction, at a given instant.

The theory of the liquid state involves the use of statistical mechanics to infer properties of what is termed the "structure" of a liquid. This structure refers to the necessarily probabilistic description of the conditional probabilities for certain classes of molecular adjacencies, e.g. pairs of molecules within a given interval of radial separation between their centers, and how these probabilities respond to the underlying potential energies of interaction for such classes. However, except in limiting cases of dilute solutions, the theoretical treatments typically involve approximations, not entirely controlled in nature, that themselves need to be tested through comparison with simulations based on the same underlying molecular potential. This is another motivation for setting up the simulation program treated in this thesis.

Our long-term goal is to create a simulation tool that will enable us to test theories and interpret experiments about the liquid structure of highly concentrated protein mixtures, which not only have hard-core repulsions between molecules of different size, but also have a spectrum of attractive or repulsive interactions between specific molecular types. For instance, the radial distribution function (RDF) between such types can be inferred with the use of X-ray or neutron scattering, and our program will permit us to create useful, simplified models for such experimental data. Second, there is at present no quantitative theory of the crystallization of proteins from mixtures, and our program will enable us to more confidently proceed with such theories. While

the crystallization of proteins is of crucial importance for ascertaining their structure by X-ray crystallography and thus the relationships between their chemistry and their workings as mechanical machines, the discovery of conditions that are suitable for crystallizing a given macromolecule are at present discovered largely by trial and error, such a theory would be useful, and our program is designed to help us make important strides towards its development.

The potential that describes the square-well interaction is given as

$$\phi(x) = \begin{cases} \infty, & 0 < x < r \\ -\varepsilon, & r < x < \lambda r \\ 0, & \lambda r < x < \infty \end{cases}$$

where r, is the radius of the hard-sphere and λ is the square-well width. Consider the following expression, the interaction force is equal to the change in potential energy.

$$m\frac{d^2x}{dt^2} = -\frac{d\phi}{dx}$$
$$\int m\frac{d^2x}{dt^2}\frac{dx}{dt} = -\int \frac{d\phi}{dx}\frac{dx}{dt}$$

We find that

$$\frac{1}{2}m\left(\frac{dx}{dt}\right)^2 + \phi(x) = C$$

Though we are not directly integrating Newton's laws of motion in the simulation, this shows that we are using an integral of them, and that the energy is conserved.

Chapter 2

Hard-Sphere Molecular Dynamics

In the hard-sphere simulation we are modeling the interactions of coarse grained particles interacting within a periodic container, in this case a box. The simulation is structured as an event driven model, where the time dependent states of the simulation are determined by the collision of the next two particles. Since the aim of the simulation is to study the structure of the particles as a consequence of Newton's laws, it would not be suitable for the states to be at fixed time intervals because it would be infeasible to capture all of the possible interactions. There are two main components to any event-driven simulation: event detection and event handling. Event detection is responsible for determining collision times and which is the next to occur and event handling is responsible for taking that most recent event and changing the state of the simulation according to the event type. Building on the idea of event detection, is event management, which is critical in Molecular Dynamics because the significant number and frequency of events.

2.1 Event Detection

The core challenge in event detection is modeling the period boundary condtions in a way that is both accurate and computationally feasible. The challenge arises when two particles, both with random velocities, are moving around in the container in such a way that they collide but only after crossing the box several times. This is

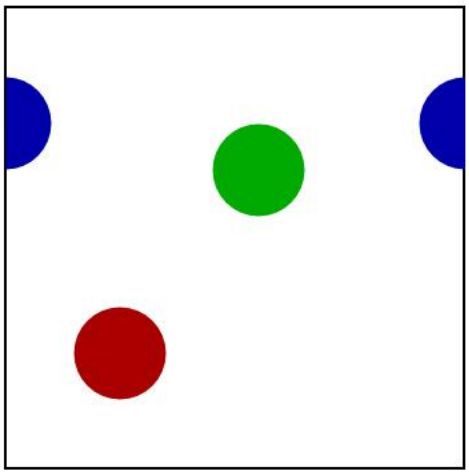


Figure 2-1: Periodic Box in 2-D

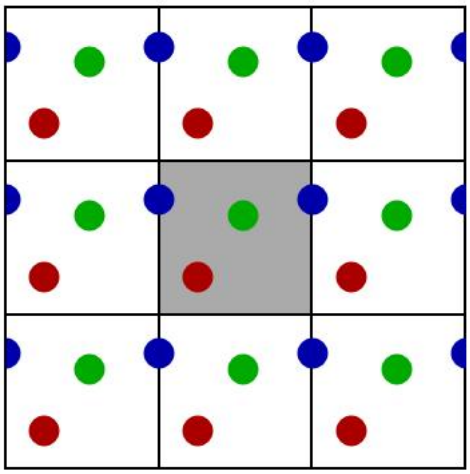


Figure 2-2: Primary Box with Copies

shown in Figure 2-1, in two dimensions.

The standard method used to handle this type of scenario is presented in Figure 2-2. It involves copying the box, along with its contents, 26 times and placing all of the copies on all the sides and corners of the original box. [1] This gives the ability to detect if the two particles have collided after one crossing of the box. Since it is possible for particles to collide after numerous crossings of the box, there is a need to develop a mechanism to handle these types of scenarios. This is accomplished by defining events that are not particle-particle interactions, but ones that act as place holders in time, where the simulation does not have enough information to keep going. This process generates a great deal of redundancy, since each box holds an exact copy of the particles in the original. In order to minimize the total computation needed to model this, only one particle is generated throughout the 27 boxes because the positions of its 26 copies can easily be determined. In this sense, every generated particle within the 27 boxes acts as a place holder for a family of 27 particles. Initially, all the generated particles are placed in the primary box, but are able to move about any of the 27 boxes. Since this is an event-driven simulation, the aim is to be able to efficiently compute collision times. They are calculated by looking at all the possible pairings of particles, and determining when the next collision occurs. For each pairing of particles, the the collision time calculation is performed in the rest frame of one of

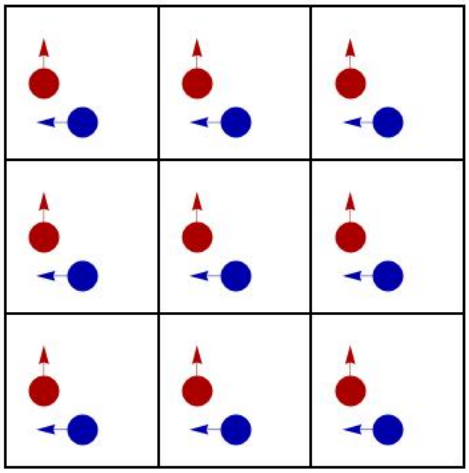


Figure 2-3: Velocity in lab frame

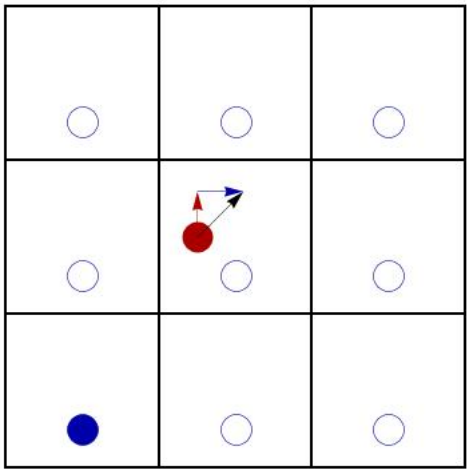


Figure 2-4: Velocity in blue ball rest frame

the two particles, call it particle a. [3] Because particle a represents a family of 27 particles, all of its siblings are generated, and a collision time is computed between each of them and the other particle in the original pairing. The final collision time, if one exists, between the two particles is the smallest non-negative collision time.

2.1.1 Ball Collision Detection

Consider two balls, a and b, with positions and velocities, $\bar{p}_a$, $\bar{v}_a$, and $\bar{p}_b$, $\bar{v}_b$, respectively. We define the velocity of ball b in the rest frame of ball a, to be $\bar{v}_r = \bar{v}_b - \bar{v}_a$. In the rest frame of ball a, the path of ball b can be parameterized by the line

$$\bar{p}(t) = \bar{p}_b + \bar{v}_r t, \quad 0 \leq t \leq t_{a,b} \tag{2.1}$$

The collision time, $t_{a,b}$ is a time at which the equation

$$\|\bar{p}\,(t_{a,b}) - \bar{p}_a\| = \gamma \tag{2.2}$$

is satisfied, where $\gamma = r_a + r_b$. By expanding the expression on the left hand side of Eq. 2.2 yields

$$((\bar{p}_b + \bar{v}_r t_{a,b}) - \bar{p}_a) \cdot ((\bar{p}_b + \bar{v}_r t_{a,b}) - \bar{p}_a) = \gamma^2$$

By rearranging we obtain

$$((\bar{p}_b - \bar{p}_a) + \bar{v}_r t_{a,b}) \cdot ((\bar{p}_b - \bar{p}_a) + \bar{v}_r t_{a,b}) = \gamma^2$$

By simplifying we obtain

$$(\bar{v}_r \cdot \bar{v}_r)\, t_{a,b}^2 + 2\left[(\bar{p}_b - \bar{p}_a) \cdot \bar{v}_r\right] t_{a,b} + (\bar{p}_b - \bar{p}_a) \cdot (\bar{p}_b - \bar{p}_a) - \gamma^2 = 0$$

By solving we obtain the collision time

$$t_{a,b} = -\xi \pm \sqrt{\xi^2 - \frac{(\bar{p}_b - \bar{p}_a) \cdot (\bar{p}_b - \bar{p}_a) - \gamma^2}{\bar{v}_r \cdot \bar{v}_r}}, \text{ where } \xi = \frac{(\bar{p}_b - \bar{p}_a) \cdot \bar{v}_r}{\bar{v}_r \cdot \bar{v}_r}$$

The above equation gives us the collision time for ball b and any copy of ball a. Since there are 27 copies of ball a, we must solve this expression for each one and pick the smallest positive root. In the case that this type of root exists, we can conclude that a hard-sphere collision has occurred. However, if it does not, we need a way to find out if they collide at some point in the future. The most natural way of handling this situation, is by saying that if the one ball does not hit any of the 27 copies, it must hit the outer boundary of the 27 containers. This type of collision, because it is in the rest frame of one of the balls, we will call a *Virtual Boundary* collision. It symbolizes a point a which we dare not go any further because we do not have any information about what lies in the future beyond that point.

2.1.2 Boundary Collision Detection

The six boundaries of our computational domain are

$$\begin{aligned} \text{left (B1)}: &\ x = -1 \\ \text{right (B2)}: &\ x = 2 \\ \text{front (B3)}: &\ y = -1 \\ \text{back (B4)}: &\ y = 2 \\ \text{bottom (B5)}: &\ z = -1 \\ \text{top (B6)}: &\ z = 2 \end{aligned}$$

Consider a ball with position and velocity, $\bar{p}$ and $\bar{v}$. In order to determine when it collides with a boundary whose equation is

$$ax + by + cz + d = 0$$

We need to first determine which side of the boundary we want the ball to be on. This will determine how we modify the parameter d in the above equation. Let γ be our interaction distance, if the plane lies below the origin, we want to subtract from d by one factor of γ and the reverse if the plane lies above the origin. So we have

$$a\left(p_x + v_x t\right) + b\left(p_y + v_y t\right) + c\left(p_z + v_z t\right) + d + \Delta\gamma = 0$$

Solving this yields

$$t_c = \frac{-\Delta\gamma - d - ap_x - bp_y - cp_z}{av_x + bv_y + cv_z}$$

In the case that $av_x + bv_y + cv_z = 0$, there is no collision.

Since there are a total of six boundaries, we need to check all of them and pick the smallest collision time. Now that we have a collision time, either of a hard-sphere collision or a *Virtual Boundary* collision, we can talk about how events such as these

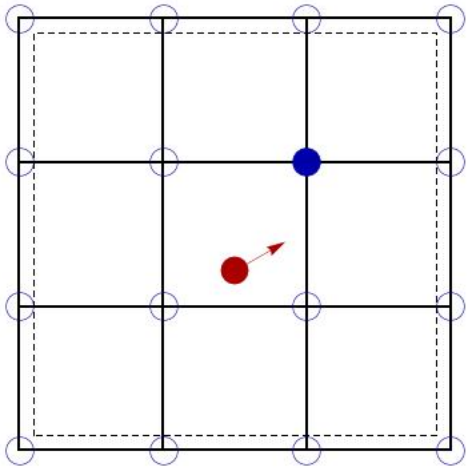


Figure 2-5: Outer boundaries are reduced

are handled. Before we delve into that, we need to mention one other type of event that arises, though it may not be fully clear yet why we need it. That is the *Hard Boundary* collision, which arises from the fact that we do not want balls in our 27 containers leaving them. The collision detection works exactly the same way as it does for the Virtual Boundary, except physically a Virtual Boundary collision is a boundary collision in the rest frame of another ball, whereas a *Hard Boundary* collision happens in the lab frame.

2.1.3 Details

A large part of what makes this simulation difficult is in the details, specifically, events that happen very rarely. Though these events occur sporadically, it is imperative to incorporate a certain amount of flexbility into the algorithm in order to account for them. For example, one could say that the probability of two collisions happening at the exact same time is zero in nature, but this is certainly a possibility when you are dealing with a finite amount of precision. Another case would be of a multi-ball collision. The simulation needs to be designed in such a way that it allows for cases such as these. Consider the case outlined in Figure 2-5, where a ball is on the boundaries of several containers. For simplicity this is illustrated in 2-D.

	b_1	b_2	b_3	$\cdots$	b_{n-2}	b_{n-1}	b_n
b_1	0	0	0	$\cdots$	0	0	0
b_2	$t_{2,1}$	0	0	$\cdots$	0	0	0
b_3	$t_{3,1}$	$t_{3,2}$	0	$\cdots$	0	0	0
b_4	$t_{4,1}$	$t_{4,2}$	$t_{4,3}$	$\cdots$	0	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
b_{n-1}	$t_{n-1,1}$	$t_{n-1,2}$	$t_{n-1,3}$	$\cdots$	$t_{n-1,n-2}$	0	0
b_n	$t_{n,1}$	$t_{n,2}$	$t_{n,3}$	$\cdots$	$t_{n,n-2}$	$t_{n,n-1}$	0
B_1	$t_{n+1,1}$	$t_{n+1,2}$	$t_{n+1,3}$	$\cdots$	$t_{n+1,n-2}$	$t_{n+1,n-1}$	$t_{n+1,n}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
B_6	$t_{n+6,1}$	$t_{n+6,2}$	$t_{n+6,3}$	$\cdots$	$t_{n+6,n-2}$	$t_{n+6,n-1}$	$t_{n+6,n}$
MIN	t_1	t_2	t_3	$\cdots$	t_{n-2}	t_{n-1}	t_n

Figure 2-6: Collision Matrix with Minimum Time Array

This case highlights the need to generate 16 balls (64 in 3-D), instead of the usual 9 (27 in 3-D). Since we want the algorithm to be as efficient as possible, we want to avoid such cases. In order to handle this scenario, there are two options. The first is to generate the additional copies, but there is a more practical and computationally simpler way of dealing with it. That is to limit the size of the outer boundaries of the cells in a way that will trigger a Virtual Boundary collision before the missed collision were to occur.

2.2 Event Management

The main goal in this simulation is determining when the next event occurs. In the case of two balls, finding the smallest collision time is trivial. However if there are several thousand or perhaps even a million balls, determining a minimum requires formulating a strategy. Futhermore, what simplifying assumptions can we make about the state of the simulation before and after an event has occured that would reduce total computation.

2.2.1 Collision Matrix

We have developed what has been dubbed a *Collision Matrix*, which as its name may suggest, is a matrix of collision times. The main body of the matrix is lower triangular, because there are $\frac{n(n-1)}{2}$ possible pairings of the balls, plus a 6 by n addition at the bottom to allow for Boundary Collisions which involve a single ball and a boundary. Each t_{ij}, in the first n rows symbolizes the collision time, either hard-sphere or Virtual Bound, between balls i and j. The last 6 rows are the collision times of boundary $i-n$ and ball j. The usefulness of the matrix comes to light in the following scenario. Suppose balls k and m collide, since not all future ball collisions are necessarily affected by the result of this collision, we would only have to update the rows and columns corresponding to each ball. Consequently the remaining entries of the matrix are completely unaffected. Of course the matrix by itself does not give us a minimum collision time, to obtain it we must scan the columns of the matrix and pick a minimum.

2.2.2 Minimum Time Array

In combination with the matrix there is a *Minimum Time Array*, which stores the smallest collision time from each column. After each update of the matrix, the time array is updated according to what the new minimum for that column is, assuming there is one. Once this is done, a minimum is picked from the whole array, this becomes our minimum collision time.

2.3 Event Handling

Event Handling is responsible for changing the state of the simulation according to the information provided to it by Event Management. The state changes are dependent on the type of interaction because each event has a different purpose and is therefore handled differently. Before we mention how specific events are handled, it is important to address one more performance modification. As was discussed in the

previous section, it is only necessary to update the entries of the Collision Matrix which correspond to last event. If that is the case then as time is advanced in the form of the next event taking place, could we get away with not updating any of the balls not involved in the event? It turns out that because all of the unaffected balls have constant velocities, by giving them all local times, we can calculate their current positions based on the collision time. In short, we only have to update the balls that a given event corresponds to.

Virtual Boundary

Both balls are advanced in the direction of their respective trajectories according to the collision time and then shifted into their corresponding positions in the primary container.

Hard Boundary

Since this collision is between a single ball and a boundary, it is handled by advancing the ball by the collision time and shifting it into the primary container.

Hard-Sphere

Both balls are advanced by the collision time and shifted into the primary container, at which point they should be touching. Next, a hard-sphere collision calculation is performed to determine their new velocities.

In all of these cases, the local time of each ball is updated to the collision time.

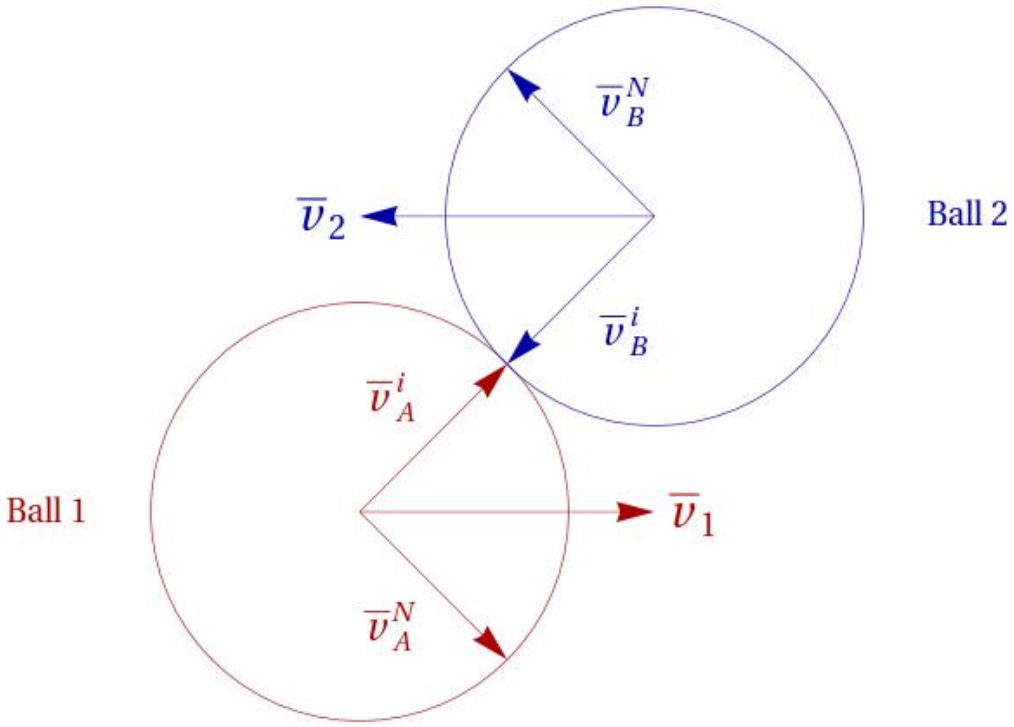


Figure 2-7: Vector decomposition

2.3.1 Hard-Sphere Collision Dynamics

The simplest method for determining the final velocities of the balls in 3-D is to reduce to a 1-D problem. Given, position vector, velocity vector, and mass: $\bar{x}_1$, $\bar{v}_1$, m_1 and $\bar{x}_2$, $\bar{v}_2$, m_2, for balls 1 and 2, respectively, at the time of collision. We can compute the unit vector $\hat{a}$, pointing from ball 1 to ball 2.

$$\hat{a} = \frac{\bar{x}_2 - \bar{x}_1}{\|\bar{x}_2 - \bar{x}_1\|}$$

From here we can project the vectors v_1 and v_2 onto $\hat{a}$. By doing so we are decoupling the component of the velocity that we want to change. The magnitude of these projected vectors are the velocities that will allow us to reduce this to a 1-D problem.

$$\bar{v}_A^i = \underbrace{(\bar{v}_1 \cdot \hat{a})}_{v_A^i}\hat{a} \qquad \bar{v}_B^i = \underbrace{(\bar{v}_2 \cdot \hat{a})}_{v_B^i}\hat{a}$$

The velocity vectors $\bar{v}_1$ and $\bar{v}_2$ can be decomposed as

$$\bar{v}_1 = \bar{v}_A^i + \bar{v}_A^N \qquad \bar{v}_2 = \bar{v}_B^i + \bar{v}_B^N$$

where $\bar{v}_A^N$, $\bar{v}_B^N$ are the vectors normal to the collision, and $\bar{v}_A^i$, $\bar{v}_B^i$ are the vectors pointing in the direction of the collision. Since $\hat{a}$ is a unit vector pointing in the direction of the collision, the collision problem reduces to solving a scalar equation using v_A^i and v_B^i.

$$v_A^f = \frac{(m_1 - m_2)\, v_A^i + 2m_2 v_B^i}{m_1 + m_2}$$
$$v_B^f = \frac{(m_2 - m_1)\, v_B^i + 2m_1 v_A^i}{m_1 + m_2}$$

The final velocities are given by

$$\bar{v}_1^f = v_A^f \hat{a} + \bar{v}_A^N$$
$$\bar{v}_2^f = v_B^f \hat{a} + \bar{v}_B^N$$

Chapter 3

Square-Well Molecular Dynamics

The square-well simulation retains everything from the hard-sphere simulation but adds in a new type of event. The square-well event is more complicated than a hard-sphere collision because the result depends on the energies of the balls and the strength of the square-well.

3.1 Event Detection

This is largely the same as in the hard-sphere case, but now we are also detecting for square-well interactions. The square-well is a pair property between two balls, so its width and depth is dependent on the two types of balls that are colliding. The same equation that was derived in the previous section is used to detect these interactions with $\gamma = \lambda (r_1 + r_2)$, where λ is the square-well width.

3.2 Event Handling

The events discussed remain the same, but we now add a square-well event which will symbolize either the entering or exiting of a pair of balls from a Square-Well. In the event that the balls are entering, their kinetic energies increase by a total of the square-well depth. If they are leaving, the result depends on if they have enough energy to leave or not. If they do not, the balls undergo an elastic collision against

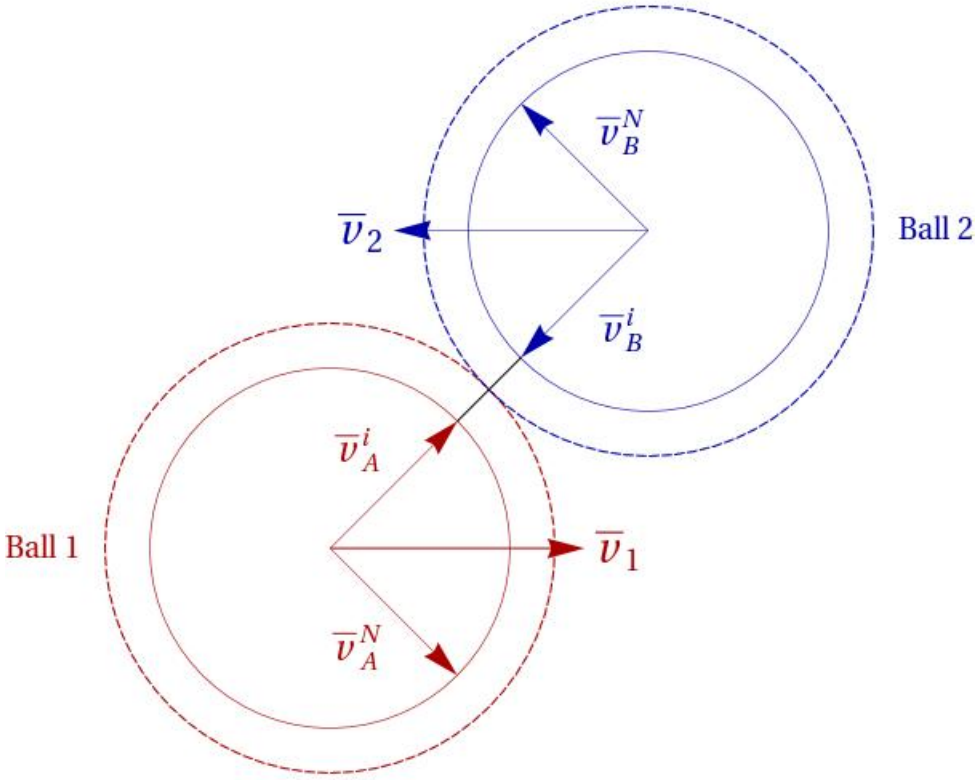


Figure 3-1: Vector decomposition of square-well collision

the square-well boundary. [3] If they do, then their total kinetic energies decrease by the square-well depth.

3.2.1 Square-Well Collision Dynamics

Given, position and velocity vectors $\bar{x}_1$, $\bar{v}_1$ and $\bar{x}_2$, $\bar{v}_2$, for balls 1 and 2, respectively, at the time of collision. We can compute the unit vector $\hat{a}$, pointing from ball 1 to ball 2.

$$\hat{a} = \frac{\bar{x}_2 - \bar{x}_1}{\|\bar{x}_2 - \bar{x}_1\|}$$

From here we can project the vectors v_1 and v_2 onto $\hat{a}$. By doing so we are decoupling the component of the velocity that we want to change. The magnitude of these

projected vectors are the velocities that will allow us to reduce this to a 1-D problem.

$$\bar{v}_A^i = \underbrace{(\bar{v}_1 \cdot \hat{a})}_{v_A^i}\,\hat{a} \qquad \bar{v}_B^i = \underbrace{(\bar{v}_2 \cdot \hat{a})}_{v_B^i}\,\hat{a}$$

The velocity vectors $\bar{v}_1$ and $\bar{v}_2$ can be decomposed as

$$\bar{v}_1 = \bar{v}_A^i + \bar{v}_A^N \qquad \bar{v}_2 = \bar{v}_B^i + \bar{v}_B^N$$

where $\bar{v}_A^N$, $\bar{v}_B^N$ are the vectors normal to the collision, and $\bar{v}_A^i$, $\bar{v}_B^i$ are the vectors pointing in the direction of the collision. Since $\hat{a}$ is a unit vector pointing in the direction of the collision, the collision problem reduces to solving a scalar equation using v_A^i and v_B^i. The logic that determines the resulting velocities after the collision is as follows.

IF ENTERING square-well, in other words if $v_A^i > v_B^i$

$$v_A^f = \frac{m_A\left(m_A v_A^i + m_B v_B^i\right) + \sqrt{m_A^2 m_B^2\left(v_A^i - v_B^i\right)^2 + 2m_A m_B\left(m_A + m_B\right)\Delta \mathrm{u}}}{m_A\left(m_A + m_B\right)}$$

$$v_B^f = \frac{m_B\left(m_A v_A^i + m_B v_B^i\right) - \sqrt{m_A^2 m_B^2\left(v_A^i - v_B^i\right)^2 + 2m_A m_B\left(m_A + m_B\right)\Delta \mathrm{u}}}{m_B\left(m_A + m_B\right)}$$

IF EXITING square-well, in other words if $v_A^i < v_B^i$

IF the balls have enough kinetic energy, in other words if $\Delta \mathrm{u} \leq \frac{m_A m_B\left(v_A^i - v_B^i\right)^2}{2(m_A + m_B)}$

$$v_A^f = \frac{m_A\left(m_A v_A^i + m_B v_B^i\right) - \sqrt{m_A^2 m_B^2\left(v_A^i - v_B^i\right)^2 - 2m_A m_B\left(m_A + m_B\right)\Delta \mathrm{u}}}{m_A\left(m_A + m_B\right)}$$

$$v_B^f = \frac{m_B\left(m_A v_A^i + m_B v_B^i\right) + \sqrt{m_A^2 m_B^2\left(v_A^i - v_B^i\right)^2 - 2m_A m_B\left(m_A + m_B\right)\Delta \mathrm{u}}}{m_B\left(m_A + m_B\right)}$$

ELSE they have an elastic collision against the square-well boundary.

$$v_A^f = \frac{(m_A - m_B)\, v_A^i + 2m_B v_B^i}{m_A + m_B}$$
$$v_B^f = \frac{(m_B - m_A)\, v_B^i + 2m_A v_A^i}{m_A + m_B}$$

The final velocities are given by

$$\bar{v}_1^f = v_A^f \hat{a} + \bar{v}_A^N$$
$$\bar{v}_2^f = v_B^f \hat{a} + \bar{v}_B^N$$

Chapter 4

Implementation

This type of simulation lends itself towards an object-oriented approach in terms of implementation. Given the large number of computations needed, speed played an essential role, so C++ was the natural choice for programming language. Although the Boost Libraries were considered at one point because of thread support, ultimately only the Standard Library was used.

Since this is an object-oriented design, let us start off by defining some objects.

Ball

Position, Velocity, Mass, Radius, Local Time, Type

Boundary

Contains the 4 coefficients of the plane equation: $ax + by + cz + d = 0$

Collision

2 Balls, Type, Time

CollisionMatrix

This object is responsible for generating and managing collisions. It is structured as a matrix of Collision objects, which has n columns and $n + 6$ rows, where n is the

number of balls. Attached to this matrix is a Minimum Time Array, which holds the minimum collisions time for each column.

Note that the Square-Well is not mentioned anywhere, that is because it is a pair property and is implemented as a table where the width and depth can be determined based on the ball types.

4.1 Initialization

4.1.1 Input File

It would be cumbersome to use commandline arguments to set the large number of parameters needed, so an input file is used instead. There are three main sections in it, the initial information, ball specifications, and Square-Well specifications. The initial information comprises of Simulation Length, the number of the first type of ball, and sampling parameters for the radial distribution functions. The ball specifications are relative radius, volume fraction, and mass. The number of Square-Well parameters depends on the number of ball types used, in general for n ball types, there should be $\frac{n(n+1)}{2}$ entries for both width and depth. The piece of code that reads the data file is designed to be scalable, so it is relatively simple to add more entries to any of the three sections.

4.1.2 Generating Balls and Boundaries

After the input file is read, the balls have to be generated based on the user's specifications. In order to keep thing as simple as possible, the user specifies relative radii of all the balls, R_i, the desired volume faction, ρ_i, plus the number of the first type of ball, N_1. Based on this information we can calculate the actual radii and number

of all the ball types. The actual radii are given by

$$r_1 = \sqrt[3]{\frac{3\rho_1}{4\pi N_1}}, \qquad r_i = \frac{R_i}{R_1} r_1, \quad i \geq 2$$

The number of each ball type is given by

$$N_i = \frac{3\rho_i}{4\pi r_i^3}, \quad i \geq 2$$

At this point we have determined all the attributes we need to generate the balls inside the container. There was a fair amount of experimentation done to determine what the best packing arrangement would be to use. Unfortunately, since we are dealing with potentially multiple types of balls it would be very difficult to design the perfect packing arrangement for every parameter combination. For example, if the container is partitioned, where different parts contain different types of balls, the time for the simulation to properly equilibrate goes up. Since we will rarely exceed total volume fractions higher than 25%, a random packing is used to ensure that the equilibration will occur as quickly as possible. The pseudo code for ball generation is as follows.

Generate Balls

```
FOR EACH ball type i
    FOR EACH ball of type i
        placed = false
        WHILE placed = false
            place = true
            p = random position
            k = 0
            WHILE k is less than number of placed AND place = true
                IF the distance between p and the kth ball is less than the Square-Well
                    place = false
                k++
```

END

IF place = true

v = random velocity

generate ball based on p, v, r_i, mass$_i$, time 0, and i

END

END

END

END

The boundary is defined by the four coefficients of the plane equation. Since the balls will be moving around within the 27 containers, it is only necessary to generate the boundaries of the containers on the edges. Furthermore, the boundaries will be shifted in by one factor of the maximum interaction distance of the balls. The pseudo code is as follows.

Generate Boundaries

mid = 0

FOR EACH possible ball pair

IF interaction distance for given pair is greater than mid

mid = interaction distance for given pair

END

END

The array is of the form $\{a, b, c, d\}$, corresponding to $ax + by + cz + d = 0$.

array = $\{0, 0, 0, 0\}$

FOR $i = 0, 1, 2$

array[i] = 1

array[3] = 1-mid

Create boundary based on array

array[3] = -2+mid

Create boundary based on array

array[i] = 0

END

4.1.3 Initializing CollisionMatrix

Once the balls and boundaries are generated, the process of populating the CollisionMatrix can begin. Initially we have to examine exactly how a collision time is determined for two balls. Before we can do this we have to mention the function that generates the 27 copies of each ball.

Generate Ball Copies

p[3] = current position of ball we are generating copies of
s_array = $\{-2, -1, 0, 1, 2\}$
start_index[3]
FOR EACH dimension d
  IF coordinate of p is between the lower boundary and 0
    start_index[d] = 2;
  IF coordinate of p is between the 0 and 1
    start_index[d] = 1;
  IF coordinate of p is between the 1 and the upper boundary
    start_index[d] = 0;
END
FOR i = start_index[0],start_index[0]+1,start_index[0]+2
  FOR j = start_index[1],start_index[1]+1,start_index[1]+2
    FOR k = start_index[2],start_index[2]+1,start_index[2]+2
      Create ball with position:
      p[0] + s_array[i], p[1] + s_array[j], p[2] + s_array[k]
    END
  END
END
RETURN array of created balls

Ball Collision Detection

This function is effectively responsible for determining what the next type of collision occurs between the two ball specified. It returns the type of collision which fills the $\frac{n(n-1)}{2}$ portion of the CollisionMatrix. Checking whether two balls have a Hard Sphere or Square-Well collision is the same process but with different interaction distance. In the Hard Sphere case, the interaction is the sum of their radii. In the Square-Well case, it is the sum of their radii times the Square-Well width. This function takes in as two parameters the two balls that are colliding.

The current positions of both balls are determined, along with the velocity of the second in the rest frame of the first.

Copies of Ball 1 are generated.

Distance Collision is called with $\lambda = 1$, this returns the closest Hard Sphere collision, if there is one. Collision assigned to variable *hard*.

Distance Collision is called with λ equal to the Square-Well width of both ball types, this returns the closest Square-Well collision, if there is one. Collision assigned to variable *sph*

IF hard is not null AND (sph is null OR (sph is not null AND time of hard LESS THAN time of sph))

RETURN hard

ELSE IF sph is not null AND (hard is null OR (hard is not null AND time of sph LESS THAN time of hard))

RETURN sph

ELSE

A ball with the rest frame velocity of Ball 2 is generated

FOR EACH Boundary

Boundary Collision is called for each boundary

IF the collision is not null and sooner than the last one

Save collision

END

END

RETURN closest Virtual Boundary collision

END

Now we have to define that actual function that determines the Hard Sphere or Square-Well collision time.

Distance Collision

This function is responsible for detection of either Hard Sphere or Square-Well collisions. The detection process is identical for both with the only difference being in their interaction distances. The parameters this functions takes are both balls, the position of Ball 2, the rest frame velocity of Ball 2, the array of copies of Ball 1, and λ, which is the Square-Well width ($\lambda = 1$ in the Hard Sphere case.)

a = the rest frame velocity of Ball 2, dotted with itself

minimum collision time = max value of double

FOR EACH of the 27 copies of Ball 1

dp = position vector of Ball 2 minus the position vector of current Ball copy

sr = interaction distance, λ(Radius of Ball 2 + Radius of Copy)

b = 2(dp dotted with Ball 2's velocity vector)

c = dp dotted with itself minus sr squared

discriminant = $b^2 - 4ac$

IF discriminent GREATER THAN 0

roots = -b+SQRT(discriminent)/(2a),-b-SQRT(discriminent)/(2a)

This next bit of logic is critical. We want the smallest positive root, however suppose that two balls have just collided. We are asking the question: when will they collide again? In theory, since they are touching, zero should be a root that comes up. Unfortunately, because we only have a finite amount of precision, we will get a root that is very close, but not exactly zero. It is therefore critical to have a way to sort out which collision time we care about, and which we do not. The way that this is tackled, is by using a floating point comparison function that has fudge factor built

in. The function used here is called GThan, which determines if one number is greater than the other, but only returns true if the numbers are a certain ε apart. Currently, $\varepsilon = 10^{-15}$.

IF roots[0] GThan 0 AND roots[1] GThan 0
Pick the minum of roots[0], roots[1] as collision time and check to see if the is smaller than the current minimum collision time.

ELSE IF roots[0] GThan 0 AND NOT(roots[1] GThan 0)
Check to see if roots[0] is small than the current minimum collision time.
ELSE IF roots[1] GThan 0 AND NOT(roots[0] GThan 0)
Check to see if roots[1] is small than the current minimum collision time.
END
END
RETURN minimum collision time IF one exists OR null if it does not

Boundary Collision Detection

This is the function that determines the collision times of balls and boundaries. It generates the type of collisions that make up the bottom six rows of the CollisionMatrix. It takes as parameters: a ball and a boundary.

boundary_parameters = holds the four coeficients of plane equation that describes that boundary
d = ball's velocity vector dotted with the first three elements of boundary_parameters
r = radius of the ball
IF d does not equal 0
IF the last coefficient of the boundary is less than 0, thus the boundary is an upper boundary
collision time = (-r - (position vector dotted with the firts three elements of boundary_parameters))/d
ELSE, therefore this is a lower boundary

collision time = (r - (position vector dotted with the firts three elements of boundary_parameters))/d

END

END

RETURN the collision time if one exists, and null otherwise

4.1.4 Initializing Minimum Time Array

This is an array of Minimum Time objects. Each of these objects contains a time and a index. There are as many of these objects in the Minimum Time Array as there are columns in the matrix, with an Minimum Time object assigned to each column in the CollisionMatrix. The array is populated by finding the collision with smallest time in each column. The index of the Minimum Time object is set to point to that entry in the column along with the collision time. In short, every Minimum Time object represents a collision time and where we can find it in the CollisionMatrix.

4.2 Simulation Loop

The simulation itself, once initialized, has three main components that it cycles through. The fist step is to retrieve the next collision that occurs, this collision time becomes the new GlobalTime. Next, the Update Collision function is called, which changes the state of the simulation based on the type of collision that was returned. Finally, the CollisionMatrix is updated to account for the now altered state of the simulation. This process repeats for as long as the user specifies.

4.2.1 Update Collision

The update collision function takes in the current collision as a parameter. The collision object identifies what type of collision it is and gives the function all the appropriate information to change the state of the relevant balls.

Update Collision

Take the first (and possibly only) ball in the collision and place it in the primary container, also changing its local time to the current time

IF the collision is of type: Virtual Boundary, Hard Sphere, or Square-Well, then do the same to the second ball as to the first

IF it is a Hard Sphere collision then call Update Ball Collision with a $du = 0$

ELSE IF it is a Square-Well collision the call Update Ball Collision with du equaling the Square-Well depth of that pair

The function responsible for assigning balls new trajectories in the case of a Hard Sphere or Square-Well collision is called Update Ball Collision. It takes as a paramter, the collision object, along with a change in energy du, which is 0 if the collision is a Hard Sphere collision. In the Square-Well case, this function is also responsible for determining whether or not the balls are entering the Square-Well, or exiting. The following information can be determined from the collision object: mass, radius, position and velocity m_A, r_1, $\bar{x}_1$, $\bar{v}_1$ and m_B, r_2, $\bar{x}_2$, $\bar{v}_2$, for balls 1 and 2.

Update Ball Collision

The first thing that needs to be accounted for is the fact that although the centers of both balls are now in the primary container, they may not necessarily be touching because of the periodic boundary conditions. For this reason, there is a function called Adjust Position, which returns their actual distance accounting for the periodic boundaries. So we can proceed by calculating the unit vector $\hat{a}$, which points from ball 1 to ball 2.

$$\hat{a} = \frac{\bar{x}_2 - \bar{x}_1}{\|\bar{x}_2 - \bar{x}_1\|}$$

From here we can compute the magnitudes of the vectors pointing in the direction of the collision.

$$v_A^i = \hat{a} \cdot \bar{v}_1 \qquad v_B^i = \hat{a} \cdot \bar{v}_2$$

Next we determine the normal vectors.

$$\bar{v}_A^N = \bar{v}_1 - v_A^i \hat{a} \qquad \bar{v}_B^N = \bar{v}_2 - v_B^i \hat{a}$$

IF $du = 0$

$$v_B^f = \frac{(m_B - m_A)\, v_B^i + 2m_A v_A^i}{m_A + m_B}$$
$$v_A^f = \frac{(m_A - m_B)\, v_A^i + 2m_B v_B^i}{m_A + m_B}$$

ELSE IF $v_A^i > v_B^i$ (ENTERING Square-Well)

$$v_A^f = \frac{m_A\left(m_A v_A^i + m_B v_B^i\right) + \sqrt{m_A^2 m_B^2 \left(v_A^i - v_B^i\right)^2 + 2m_A m_B \left(m_A + m_B\right) du}}{m_A\left(m_A + m_B\right)}$$
$$v_B^f = \frac{m_B\left(m_A v_A^i + m_B v_B^i\right) - \sqrt{m_A^2 m_B^2 \left(v_A^i - v_B^i\right)^2 + 2m_A m_B \left(m_A + m_B\right) du}}{m_B\left(m_A + m_B\right)}$$

ELSE IF $v_A^i < v_B^i$ (EXITING Square-Well)

IF $du \leq \frac{m_A m_B \left(v_A^i - v_B^i\right)^2}{2(m_A + m_B)}$ (the balls have enough energy to leave Square-Well)

$$v_A^f = \frac{m_A\left(m_A v_A^i + m_B v_B^i\right) - \sqrt{m_A^2 m_B^2 \left(v_A^i - v_B^i\right)^2 - 2m_A m_B \left(m_A + m_B\right) du}}{m_A\left(m_A + m_B\right)}$$
$$v_B^f = \frac{m_B\left(m_A v_A^i + m_B v_B^i\right) + \sqrt{m_A^2 m_B^2 \left(v_A^i - v_B^i\right)^2 - 2m_A m_B \left(m_A + m_B\right) du}}{m_B\left(m_A + m_B\right)}$$

ELSE

$$v_A^f = \frac{(m_A - m_B)\, v_A^i + 2m_B v_B^i}{m_A + m_B}$$
$$v_B^f = \frac{(m_B - m_A)\, v_B^i + 2m_A v_A^i}{m_A + m_B}$$

END

END

Now we can set the velocities of the balls to

$$\bar{v}_1^f = v_A^f \hat{a} + \bar{v}_A^N$$
$$\bar{v}_2^f = v_B^f \hat{a} + \bar{v}_B^N$$

This function returns the type of collision that took place. This is useful when montioring the potential energy of the system.

4.2.2 Update CollisionMatrix

The update function takes the last collision as a parameter. This provides it with the ball(s), and thereby the rows/columns which need to be updated. Initially, the balls are looked up in a map which stores a one-to-one matching of ball objects with their corresponding indicies in the matrix. These indicies are then passed to the PopulateRow and PopulateColumn functions which populate the appropriate rows and columns with new collisions. The more complicated part is determining what parts of the Minimum Time Array need to be updated. There are two functions that perform this task: FindMinColumn, which scans an entire column and finds a minimum, and FindMinRow which performs scans each row. The former is trivial, however FindMinRow is not.

FindMindRow

FOR EACH entry in the row

IF the entry's time is less than the time already in the time array

Set entry in the Minimum Time Array to point to current entry in the Matrix

ELSE IF the entry in the Minimum Time Array is pointing to the current entry in the matrix AND (the current entry in the matrix is null OR the current entry in the matrix has a collision time greater than the entry in the Minimum Time Array)

Call FindMinColumn for that column

END

END

The repopulating of the elements in the collision matrix is $O(N)$, and so is FindMinColumn. However, FindMinRow varies. In the best case it is $O(N)$, in the worst it is $O\left(\frac{N(N-1)}{2}\right)$, that depends entirely on the state of the system.

4.2.3 Return Next Collision

Once CollisionMatrix is updated along with the Minimum Time array, the next collision time is the smallest collision time in the Minimum Time Array.

Chapter 5

Radial Distribution Functions

In order to determine how the balls are arranged inside the computational domain, we use a Radial Distribution Function. Consider Figure 5.1, by finding the number of hard-spheres contained within each concentric shell, and dividing by the number that we would expect to see, given a uniform distribution, we are measuring the hard-sphere densities as a function of radius. This process is performed for every hard-sphere in the container. Formally, this can be written

$$g(r) = \frac{\langle N(r, \Delta r) \rangle}{N_f V(r, \Delta r)}$$

where $N(r, \Delta r)$ is the number of hard-spheres in the shell of radius r, with thickness Δr, N_f is the normalizing factor, and $V(r, \Delta r)$, the volume of a shell of radius r and thickness Δr. [2] The value of N_f depends on what hard-sphere relationships one is examining. If it is of two different hard-sphere types then N_f equals the product of the number of both types, if one is examining one type against it self then $N_f = \frac{N(N-1)}{2}$, where N is the number of that type. The pseudo-code for calculating $g(r)$ is as follows. Assuming r_1, N_1 and r_2, N_2 are the radius and numbers of Balls 1 and 2, respectively.

dist_array = the pair distances of all the balls, if both are of the same type then there are $\frac{N_1(N_1-1)}{2}$ elements, otherwise there are $N_1 N_2$

IF the balls are of the same type

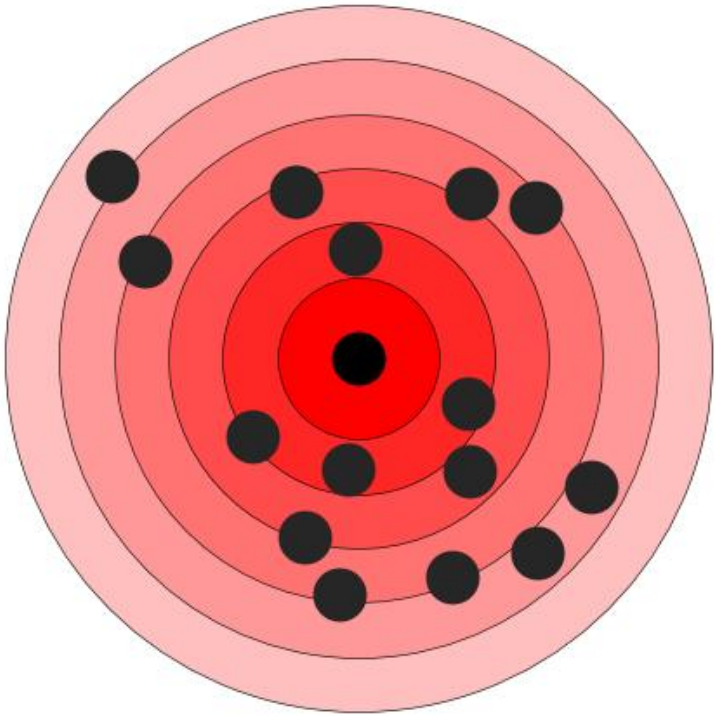


Figure 5-1: Radial Distribution Function

TOTAL_NUM_PAIRS (Normalization Factor) = $\frac{N_1(N_1-1)}{2}$

ELSE

TOTAL_NUM_PAIRS (Normalization Factor) = $N_1 N_2$

END

The BIN Fraction is a parameter that controls how finely we are bining the pair distances. If it is too large, the RDF will not have enough resolution. If it is too small, there will be too much noise to gain any meaning from it.

dr = BIN Fraction * (r_1 + r_2)

The Radial Distribution function is only computed for values of the radius from zero to half the box size, this is to avoid any posibility of errors generated by the periodic boundaries.[1]

FOR r = 0 to 0.5, in increments of dr

$Number\_in\_BIN = 0$

WHILE current element of dist_array is LESS THAN r+dr

$Number\_in\_BIN = Number\_in\_BIN + 1$

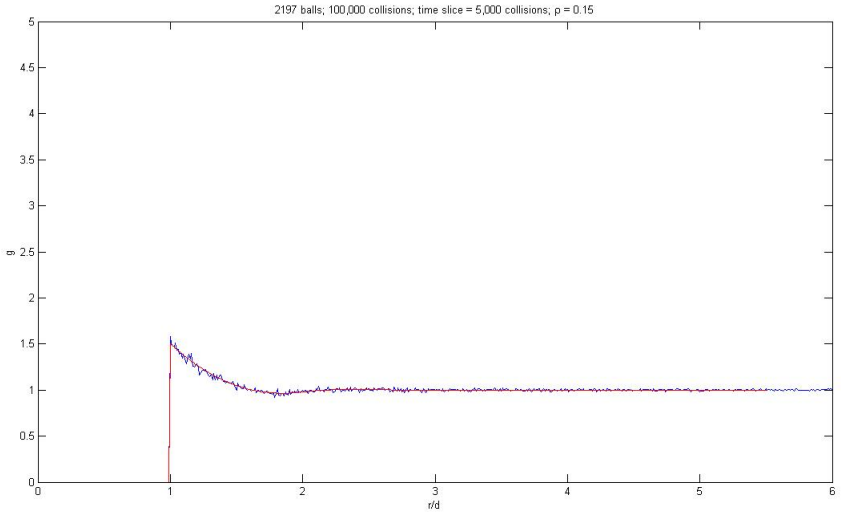
END

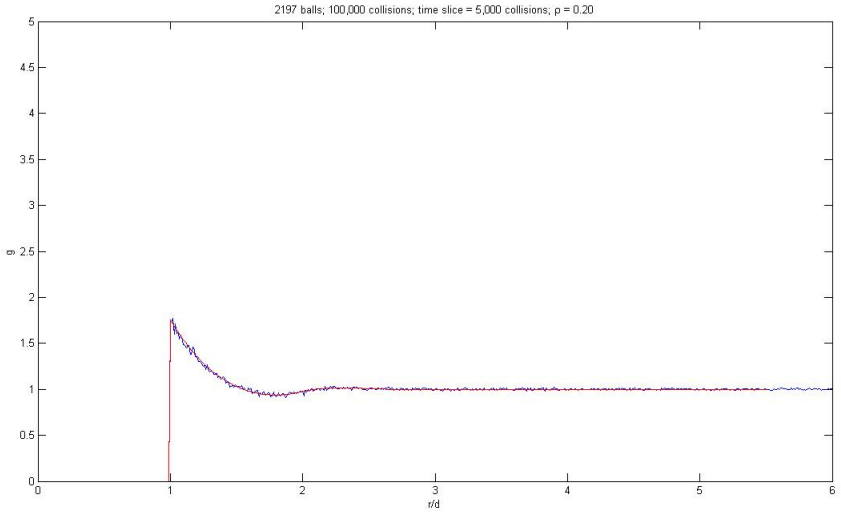
$$g(r) = \frac{Number\_in\_BIN}{TOTAL\_NUM\_PAIRS} \frac{1}{\frac{4}{3}\pi((r+dr)^3 - r^3)}$$

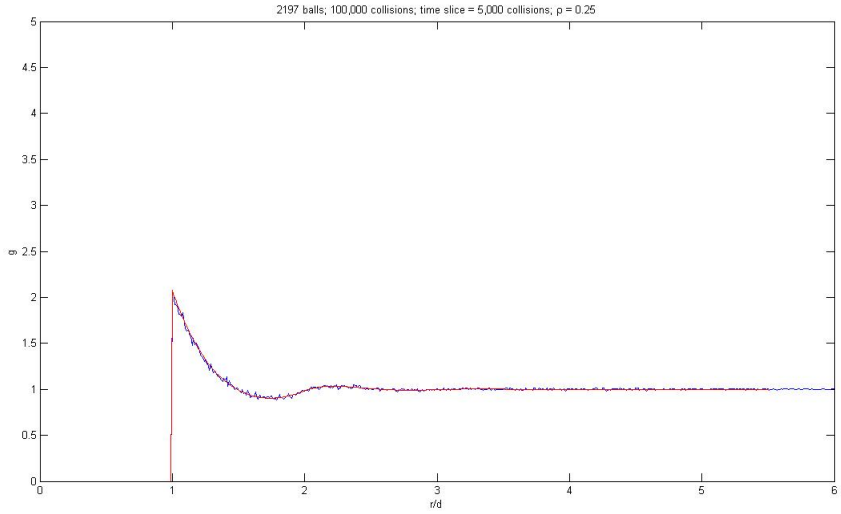
END

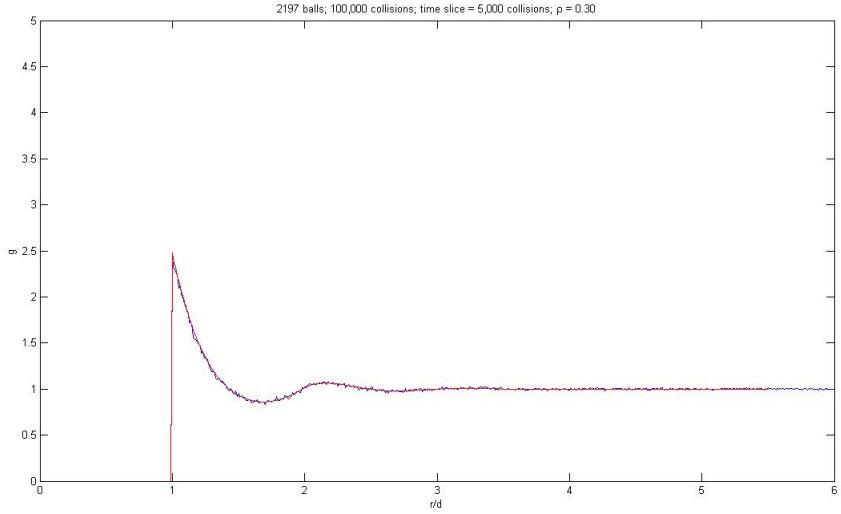
Results

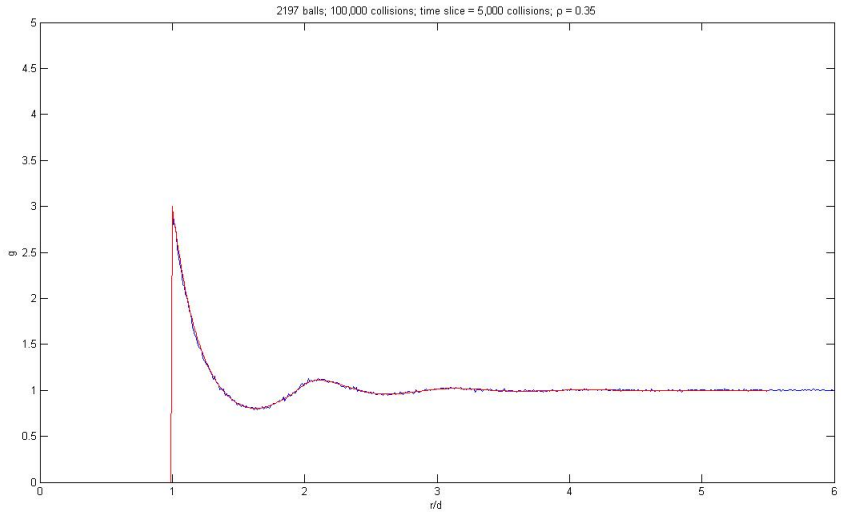
The following are radial distribution functions calculated from the hard-sphere simulation. The red line illustrates the simulation result using monte-carlo techniques, in blue we have the result generated by the molecular dynamics simulation.

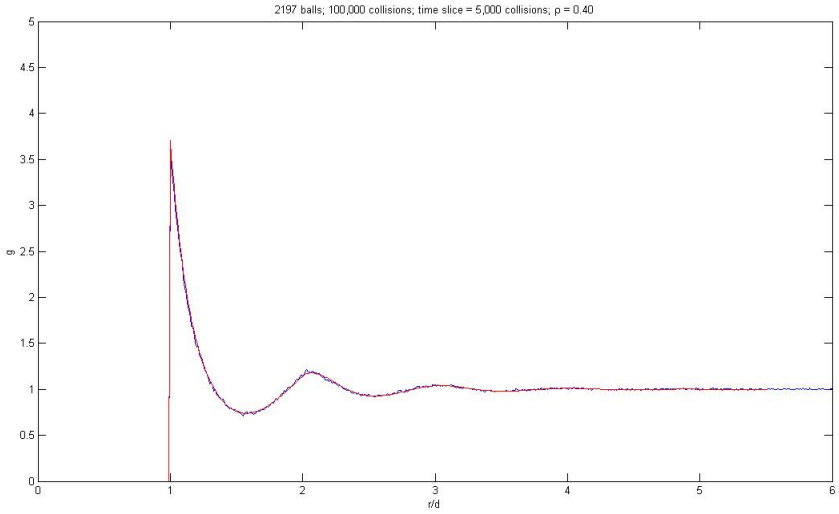


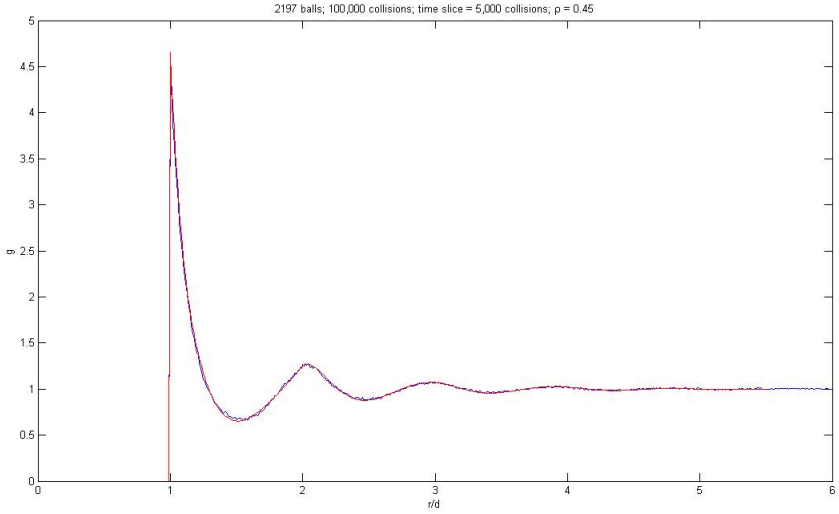



2197 balls; 100,000 collisions; time slice = 5,000 collisions; ρ = 0.25
g
r/d

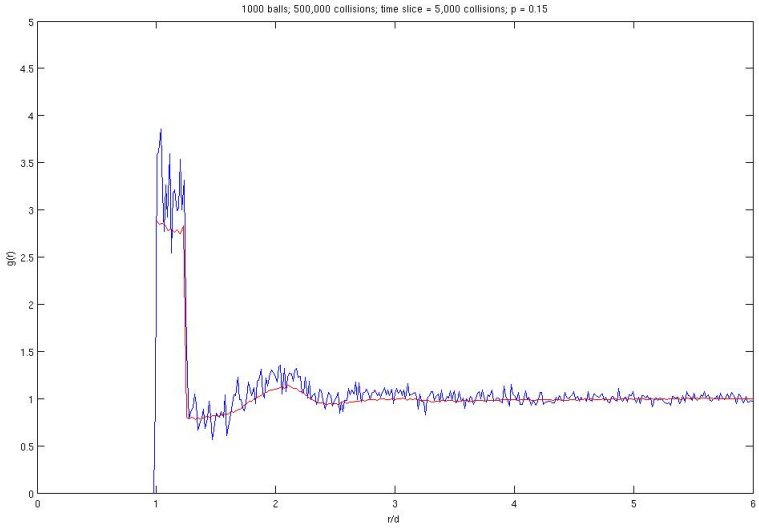

2197 balls; 100,000 collisions; time slice = 5,000 collisions; ρ = 0.30
g
r/d

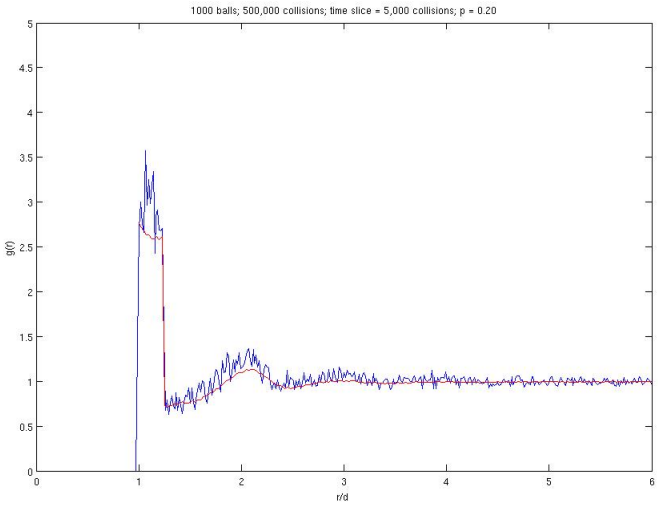

2197 balls; 100,000 collisions; time slice = 5,000 collisions; ρ = 0.35
g
r/d

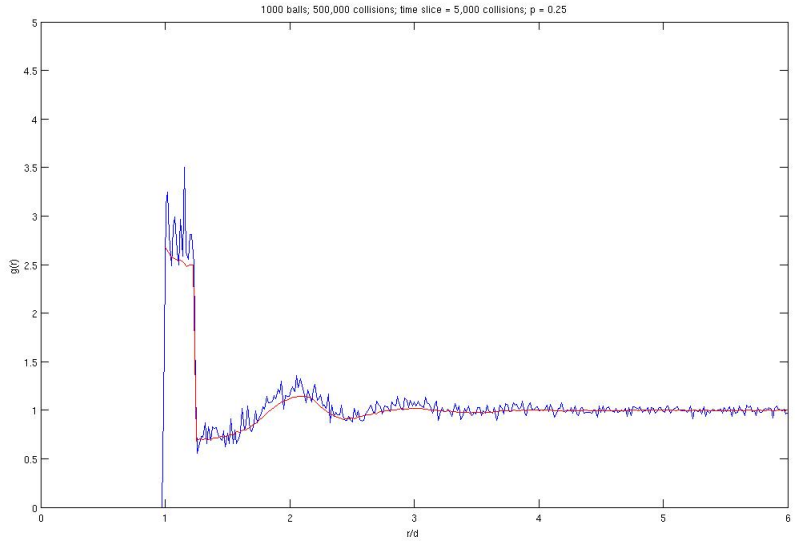

2197 balls; 100,000 collisions; time slice = 5,000 collisions; ρ = 0.40
g
r/d

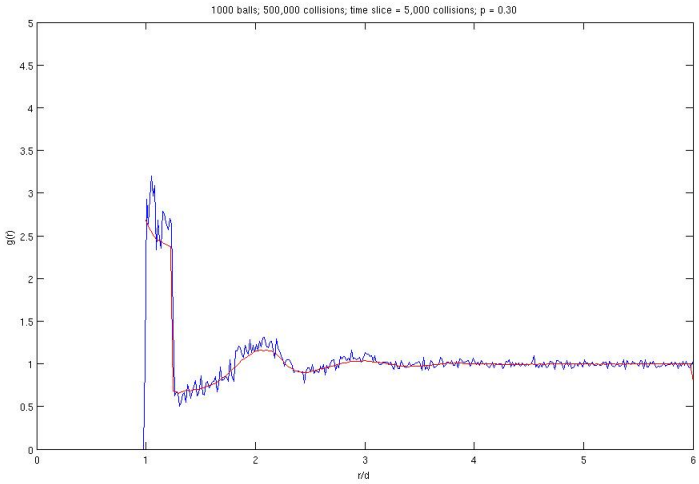

2197 balls; 100,000 collisions; time slice = 5,000 collisions; ρ = 0.45
g
r/d

These are some radial distribution functions calculated from the square-well simulation. The square-well width and depth are 1.25 and 1.267, respectively.






1000 balls; 500,000 collisions; time slice = 5,000 collisions; p = 0.25
g(r)
r/d


1000 balls; 500,000 collisions; time slice = 5,000 collisions; p = 0.30
g(r)
r/d

Chapter 6

Future Directions & Conclusion

In this thesis we have designed and implemented a highly flexible square-well molecular dynamics simulation that enables us to further examine the therodynamic properties of protein mixtures. Though the hard-sphere component of the simulation works correctly, as far as we can tell, the square-well is not completely functional yet. In its current state, there remain ongoing issues with the way the square-well simulation is equilibrating. Due to an as of yet undiscovered reason, the peaks of the radial distribution functions are simply too high. This check comes directly out of statistical mechanics literature, which states that at low concentration ($< 5\%$), the peak at around one diameter, should go to the Boltzmann factor of the potential. In the runs that we have carried out, it seems that his peak is roughly three times higher than we would expect. Before any further work can be done with this simulation, this issue needs to be resolved.

Once we can say with confidence that the behavior of the simulation is within what the theory describes, we can begin to study the interactions of mixtures of varying sized proteins. A significant component of this will be the study of the chemical potential, which due to the complexities of the problem, we were not able to address in this thesis.

The other direction in which this simulation can be explanded has to do with the particles we are simulating. Square-well particles, though they possess several useful characteristics, are not very accurate representations of the proteins themselves.

There exist many other much more complicated models that would give us a better insight into the behavior of protein mixtures.

Bibliography

1. D.C. Rapaport, *The Art of Molecular Dynamics Simulation.* New York, The Cambridge Press, 2004

2. J.M. Haile, *Molecular Dynamics Simulation: Elementary Methods.* New York, Wiley, 1997

3. Daan Frenkel & Berend Smit, *Understanding Molecular Simulation: From Algorithms to Applications Second Edition.* San Diego, Academic Press, 2002

Appendix

Hard Sphere Collision Equations

Consider the initial velocities v_A^i, v_B^i and masses m_A, m_B of balls A and B, respectively. The collision is perfectly elastic, meaning that the energy and momentum are conserved.

$$\frac{1}{2}m_A {v_A^i}^2 + \frac{1}{2}m_B {v_B^i}^2 = \frac{1}{2}m_A {v_A^f}^2 + \frac{1}{2}m_B {v_B^f}^2$$
$$m_A v_A^i + m_B v_B^i = m_A v_A^f + m_B v_B^f$$

Rearrange the above system to get the following.

$$m_A \left({v_A^i}^2 - {v_A^f}^2 \right) = m_B \left({v_B^f}^2 - {v_B^i}^2 \right) \tag{6.1}$$
$$m_A \left(v_A^i - v_A^f \right) = m_B \left(v_B^f - v_B^i \right) \tag{6.2}$$

Divide (1.1) by (1.2).

$$v_A^i + v_A^f = v_B^f + v_B^i \tag{6.3}$$

Multiply (1.3) by m_A and add it to (1.2).

$$2m_A v_A^i = (m_A + m_B)\, v_B^f + (m_A - m_B)\, v_B^i$$

Therefore,

$$\boxed{v_B^f = \frac{(m_B - m_A)\, v_B^i + 2m_A v_A^i}{m_A + m_B}} \tag{6.4}$$

Multiply (1.3) by m_B and subtract it from (1.2).

$$(m_A - m_B)\, v_A^i - (m_A + m_B)\, v_A^f = -2m_B v_B^i$$

Therefore,

$$\boxed{v_A^f = \frac{(m_A - m_B)\, v_A^i + 2m_B v_B^i}{m_A + m_B}} \tag{6.5}$$

Square-Well Collision Equations

Consider the initial velocities v_A^i and v_B^i of balls A and B, respectively. Assuming they have masses m_A and m_B, and change in kinetic energy Δu, then momentum and energy are conserved.

$$\frac{1}{2}m_A {v_A^i}^2 + \frac{1}{2}m_B {v_B^i}^2 = \frac{1}{2}m_A {v_A^f}^2 + \frac{1}{2}m_B {v_B^f}^2 + \Delta u$$
$$m_A v_A^i + m_B v_B^i = m_A v_A^f + m_B v_B^f$$

Rearrange the above system to get the following.

$$m_A \left({v_A^i}^2 - {v_A^f}^2\right) = m_B \left({v_B^f}^2 - {v_B^i}^2\right) + 2\Delta u \tag{6.6}$$
$$m_A \left(v_A^i - v_A^f\right) = m_B \left(v_B^f - v_B^i\right) \tag{6.7}$$

Divide (2.1) by (2.2).

$$v_A^i + v_A^f = v_B^f + v_B^i + \frac{2\Delta u}{m_B \left(v_B^f - v_B^i\right)} \tag{6.8}$$

Multiply (2.3) by m_A and add it to (2.2).

$$2m_A v_A^i = (m_A + m_B)\, v_B^f + (m_A - m_B)\, v_B^i + \frac{2m_A \Delta u}{m_B \left(v_B^f - v_B^i\right)}$$

Simplifying this further we get the following quadratic.

$$m_B\,(m_A + m_B)\, {v_B^f}^2 - 2m_B \left(m_A v_A^i + m_B v_B^i\right) v_B^f - m_B\,(m_A - m_B)\, {v_B^i}^2 + 2m_A m_B v_A^i v_B^i + 2m_A \Delta u = 0$$

Therefore,

$$\boxed{v_B^f = \frac{m_B\,(m_A v_A^i + m_B v_B^i) \pm \sqrt{m_A^2 m_B^2\,(v_A^i - v_B^i)^2 - 2m_A m_B\,(m_A + m_B)\,\Delta \mathrm{u}}}{m_B\,(m_A + m_B)}} \quad (6.9)$$

Multiply (2.3) by m_B and subtract it from (2.2).

$$(m_A - m_B)\, v_A^i - (m_A + m_B)\, v_A^f = -2m_B v_B^i - \frac{2m_B \Delta u}{m_A \left(v_A^i - v_A^f\right)}$$

Simplifying this further we get the following quadratic.

$$m_A\,(m_A + m_B)\, {v_A^f}^2 - 2m_A \left(m_A v_A^i + m_B v_B^i\right) v_A^f + m_A\,(m_A - m_B)\, {v_A^i}^2 + 2m_A m_B v_A^i v_B^i + 2m_B \Delta u = 0$$

Therefore,

$$\boxed{v_A^f = \frac{m_A\,(m_A v_A^i + m_B v_B^i) \pm \sqrt{m_A^2 m_B^2\,(v_A^i - v_B^i)^2 - 2m_A m_B\,(m_A + m_B)\,\Delta \mathrm{u}}}{m_A\,(m_A + m_B)}} \quad (6.10)$$