AN ABSTRACT OF THE THESIS OF

Lee Collins for the degree of Honors Baccalaureate of Science in Computational Physics

presented on May 31, 2011.

Title: Monte Carlo Simulations of Structure and Melting Transition of Small Ag Clusters

Abstract Approved: ________________________________

Guenter Schneider

Abstract

The behavior of small atomics cluster largely depends on their geometry, due to the high ratio of surface atoms to interior atoms. One interesting aspect to study clusters centers around characterizing the transition behavior between finite and bulk materials, where the ratio of surface atoms to interior atoms is very small.

Metropolis Monte-Carlo methods were used to simulate the melting of small silver clusters (3 to 56 atoms) using the Gupta potential. Each cluster was simulated from 200-1200 K in 40 K increments, allowing the cluster to transition from a liquid-like state to a solid state. The specific heat was calculated at each temperature value. A separate simulation calculated the binding energy per atom of each cooled cluster. For each cluster, the specific heat vs. temperature showed a slight peak at the phase transition from solid to liquid, corresponding to the latent heat required for the phase change. However, several cluster sizes had much higher peaks than their neighbors, identifying these sizes as clusters with a particularly stable geometry. The icosahedral 'magic number' clusters were 13 and 55 atoms, and showed full 5-fold symmetry in their final cooled state. Additional semi-magic number clusters, 19 and 25 atoms, were also identified using the binding energy per atom. These had partial 5-fold symmetry and were not as stable as the icosahedral magic number clusters.

Key Words: Magic Number Cluster, Monte Carlo, Computational, Melting, Structure

Corresponding e-mail address: leeecollins@comcast.net

Monte Carlo Simulations of Structure and Melting Transition of Small Ag Clusters

Lee Collins

5-31-2011

A PROJECT

submitted to

Oregon State University

University Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computational Physics (Honors Scholar)

Presented May 31, 2011

Commencement June 2011

Honors Baccalaureate of Science in Computational Physics project of Lee Collins presented on May 31, 2011.

APPROVED:

Mentor, representing Computational Physics

Committee Member, representing Computational Physics

Committee Member, representing Computational Physics

Dean, University Honors College

Lee Collins, Author

Contents

List of Figures

1 Introduction

The melting behavior of bulk materials ($\sim 10^{23}$ atoms) is well understood, with the material transitioning from solid to liquid at one specific temperature. Large clusters with more than ~1,000-10,000 atoms are also well understood: the material melts according to the droplet model, with a liquid surface and a solid core. The melting behavior of small atomic clusters (fewer than ~1,000 atoms) is less understood, and is difficult to study experimentally. Complexities arise when studying small clusters in the lab, either due to surface interactions between a supported cluster and the substrate, or difficulty manipulating the clusters in gaseous form. Therefore, the purpose of this project is to analyze the melting behavior of small atomic cluesters using computer models, and to determine a threshold cluster size at which the bulk model for melting breaks down and becomes the cluster model.

Small clusters of atoms differ from bulk materials in many aspects, one of the most interesting being the transition between the solid and liquid phase. Because the ratio of the number of surface atoms to the total number of atoms is very large in a small cluster, the geometry of the cluster plays a very large role in the melting behavior. In particular, the melting transitions of the 'magic number' clusters are very interesting due to the compact geometry of the cluster (Figure 1.1). For example, icosahedral clusters are those with complete 5-fold symmetry, and form a cluster with a completely closed outer shell of atoms (13,55,147... See Eq. 1.1 [2]). Other semi-magic number clusters exist as well, corresponding to different geometric and structural motifs.

$$N = \frac{10}{3}K^3 - 5K^2 + \frac{11}{3}K - 1 \qquad K = 1, 2, 3... \tag{1.1}$$

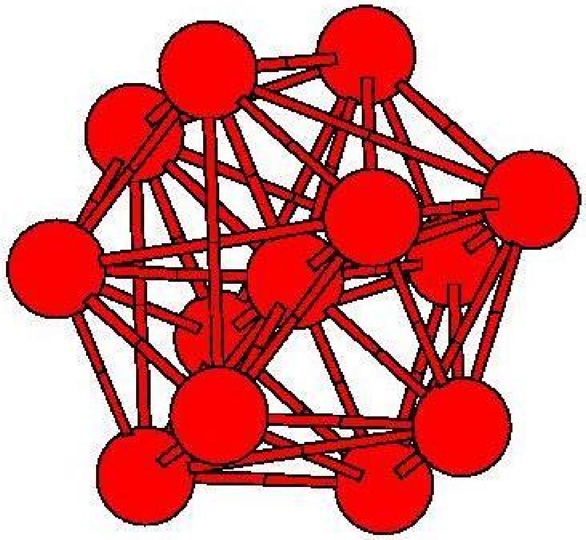

Figure 1.1: *The geometry of a 13-atom 'magic number' cluster. This unique icosahedral geometry causes interesting melting behavior that differs greatly from that of bulk materials, specifically an increase in the specific heat of the material corresponding to the latent heat required to melt it.*

Bulk materials melt at one specific temperature, unlike small clusters which melt over a range of temperatures. This range can be as large as 1000 Kelvin. Not much research has been devoted to determining the point at which melting behavior transitions from the cluster model to the bulk material model. Estimates for this range are between 1000 to 5000 atoms.

Computer simulations offer a direct approach to understanding the behavior of small clusters, and offer the ability to alter parameters easily. In a computer simulation, the accuracy of the results is limited by the model used to describe the atomic interactions. If the model used to describe the atoms is incorrect, the simulations cannot be directly compared to experimental observations. However, the model may still give useful qualitative information about the behavior of the cluster.

A simple, yet widely used model to describe the interaction between neutral atoms is the Lennard-Jones potential (Eq. 1.2). A detailed explanation of this equation is given in the Methods section (Section 2). This model is described by a pairwise potential: the total potential of the system is simply the sum of the potentials between each pair of atoms. The potential has two terms that, together, model inter-atomic behavior accurately. The attractive term dominates at large distances, simulating the van der Waals forces. The repulsive term dominates at small distances, simulating the repulsion of atomic nuclei (Figure 1.2). This potential models noble gas atoms well, and is computationally very simple, making it an ideal choice for testing the program and verifying the correct qualitative behavior of the simulation.

$$V_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] \tag{1.2}$$

The Gupta potential is a more accurate model of metallic clusters than the Lennard-Jones potential. The Gupta potential correctly describes the surface contraction that is generally observed in metals [8], and therefore was used to generate the results of this project. The Gupta potential has the same qualitative behavior as the Lennard-Jones potential: the potential approaches infinity as the distance between atoms becomes small, and approaches zero as the distance between atoms becomes large. However, the bonding term in the Gupta potential is much more complex (Eq 1.3), and therefore produces a different type of equation than the Lennard-Jones potential. The Gupta potential is more computationally intensive, and therefore requires

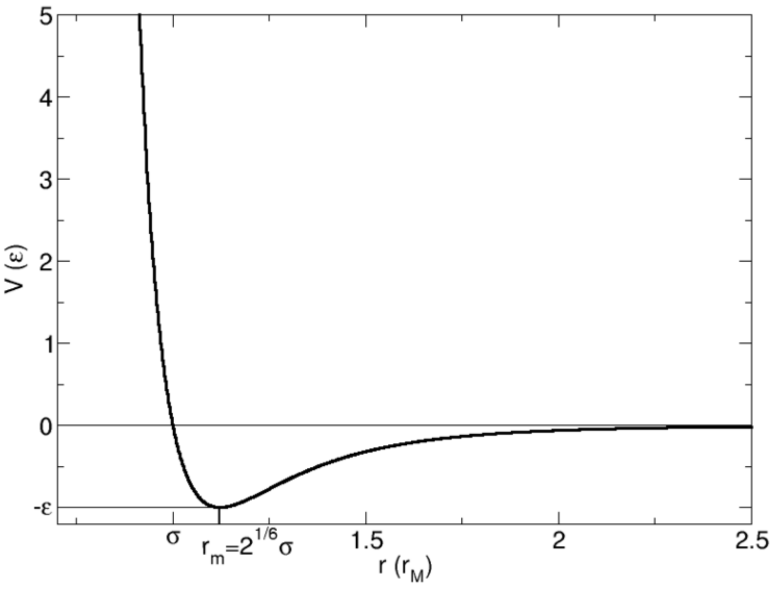

Figure 1.2: *Graphical depiction of the Lennard-Jones potential as a function of the distance between two neutral atoms. The potential is repulsive at short distances and attractive at large distances, converging to zero as r approaches infinity.*

more computing time to evaluate.

$$V(r) = \sum_{i}^{N} \left[\sum_{j \neq i}^{N} (A e^{-p r_{i,j}}) - \sqrt{\sum_{j \neq i}^{N} \xi^2 e^{-2 q r_{i,j}}} \right] \tag{1.3}$$

Computer simulations of atomic clusters come in two varieties, molecular dynamics and Monte-Carlo, each with advantages and disadvantages. Molecular dynamics simulations are 'real' in that they use Newton's Laws to allow the atoms to move according to classical mechanics, and therefore observable quantities are measured as a time-average. The gradient of the potential function determines the force on each atom, and the net force is used to 'move' each atom by a small amount. Then, the forces are re-calculated and the atoms are moved again. This process repeats, and accurately models how the atoms in a real cluster would move. This method is very comprehensive, and using molecular dynamics, some properties can be determined that cannot be determined using a Monte-Carlo simulation, such as the self-diffusion of atoms inside a cluster. Depending on the simulation type, molecular dynamics simulations can take a long time. Each atom must sample the force acting on it from all other atoms, and therefore processing time greatly increases as the cluster size increases. In addition, a molecular dynamics simulation using Newton's Laws under the influence of only internal forces conserves energy. This is a desirable quality in many instances, however energy cannot be conserved in an isothermal simulation. For an isothermal simulation, the system must be connected to a 'heat bath' to maintain the temperature, which requires more programming and further complicates the program.

Monte-Carlo simulations are much simpler and faster than molecular dynamics simulations at constant temperature, although they do not simulate 'real' dynamics at all, instead using random perturbations to generate new configurations of atoms. Averages in a Monte Carlo simulation are ensemble averages over many states of the system, as opposed to time averages in a molecular dynamics simulation. In a Metropolis Monte Carlo simulation, all atoms are displaced by a random amount, and the energy of the resulting configuration is compared to the energy of the previous configuration. If the new energy is lower than the previous energy, the new configuration is accepted. If the new energy is higher, the new configuration is accepted with a given probability (Eq. 1.4).

$$\mu = e^{-\frac{\Delta E}{k_B T}} \tag{1.4}$$

This probability depends on the temperature (T) of the simulation, as well as the energy difference between the old and new configurations (ΔE). Thus, as the change in energy becomes large, or the temperature decreases, the probability of accepting a new configuration becomes smaller. Conversely, with a large temperature or a small change in energy, it becomes very likely that the new configuration is accepted. This probability forces the system to create a Boltzmann distribution of energies, which is required for the Monte-Carlo simulation to function correctly.

Because Monte-Carlo simulations calculate observable quantities by taking an ensemble average over many states, it is crucial that the system samples many different states. Experience shows that scaling the maximum displacement to allow for approximately a 40% - 60% acceptance rate allows the system to sample enough states efficiently. This acceptance rate is a balance between making many small changes to the system and few large changes to the system.

Transitions from solid to liquid for one cluster are determined using the specific heat capacity of the cluster, a measurable property of a material that shows how much heat is required to raise the temperature by a given amount. At the melting transition, the specific heat of a material shows a peak corresponding to the latent heat required to cause the phase change (Figure 1.3). Pictures of the cluster in the phase-change region can also be used to identify the temperatures at which the cluster transitions from solid to liquid (Fig. 1.4).

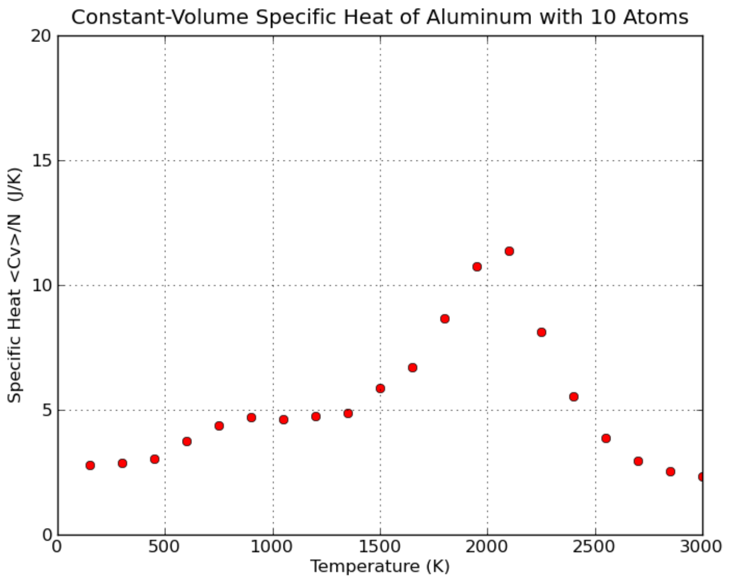

Figure 1.3: *Plot of the constant-volume specific heat of a 10-atom Lennard-Jones cluster of aluminum atoms. The peak from T=1500-2500 K shows melting over this temperature range. The cluster exists as a solid for T < 1500 K and as a liquid for T > 2500 K.*

2 Methods

The goal of this project was to examine the melting transition in small atomic clusters using computer models, and identify magic number clusters. This was accomplished using Monte-Carlo simulations to determine changes in the specific heat at the melting transition, as well as simulations to calculate the binding energy per atom.

The potential energy function used to test the interaction between atoms in this project is the Lennard-Jones (LJ) potential. This is a simple potential that describes neutral atoms very well, and is pairwise additive (the total potential can be computed by simply adding up the potential between each pair of atoms, Eq. 2.1). Aluminum atoms have well-documented Lennard-Jones parameters, and therefore were used to test the algorithm.

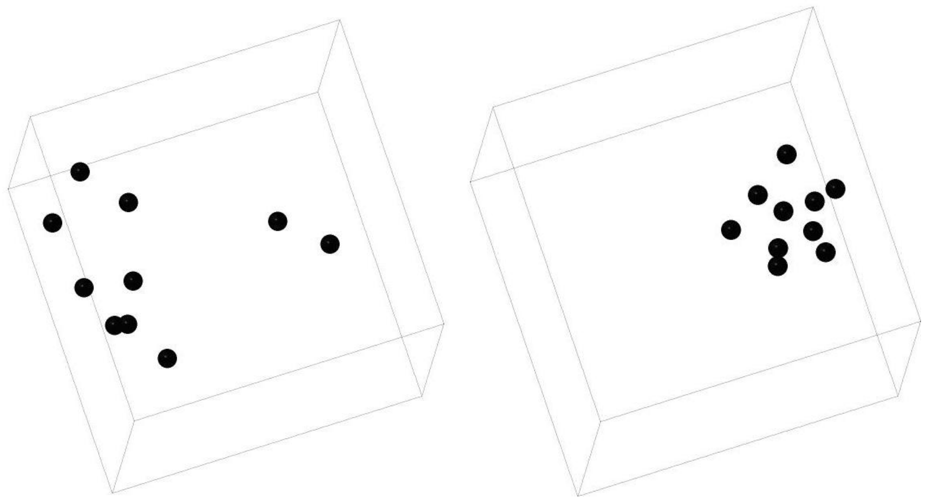

Figure 1.4: *Comparison of the 10-atom cluster depicted in Figure 1.3 immediately before the melting transition (left, T = 2500 K) and the cluster immediately after the melting transition (right, T = 1500 K). The dispersed, liquid-like state (left) is clearly distinguishable from the compact, solid state (right). This picture was generated using Jmol [4].*

$$V_{total} = \sum_{i,j}^{N} V_{LJ}(r_{ij}) \tag{2.1}$$

There are two terms in the pairwise potential function V_{LJ}, an attractive term that dominates at large distances, and a repulsive term that dominates at very small distances (Eq. 2.2). ϵ represents the depth of the potential well, σ is the distance at which the potential is zero, and r is the distance between the two atoms. Aluminum has LJ-parameters: $\epsilon = 4551$ K$\cdot k_B$, and $\sigma = 2.62$ Å[5]. Under the LJ potential, a pair of atoms will reach a minimum potential energy of -ϵ at a distance $r_m = 2^{1/6}\sigma$. For a graphical depiction of the Lennard-Jones potential between two neutral atoms, refer to Figure 1.2.

$$V_{LJ} = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.2}$$

Results were generated for the Silver clusters using the Gupta potential, similar to the Lennard-Jones potential but slightly more complicated, and therefore more computationally intensive (Eq. 2.3). The Gupta potential describes metallic interactions very well, including the surface contraction observed in metals [8]. The equation for the Gupta potential is given in Equation 2.3. The Gupta parameters for Silver are: A = 0.1028 eV (1192.94 K·k_B), ξ = 1.092 eV (12670.107 K·k_B), p = 10.928, and q = 3.139. In all simulations, k_B was set to 1 for simplicity.

$$V(r) = \sum_{i}^{N}\left[\sum_{j\neq i}^{N}(Ae^{-pr_{ij}}) - \sqrt{\sum_{j\neq i}^{N}\xi^2 e^{-2qr_{ij}}}\right] \tag{2.3}$$

The specific heat of a cluster is one common observable used to determine the melting point of the cluster. The specific heat is determined from the fluctuations of the potential energy. Eq. 2.4 gives the formula used to calculate the specific heat, where brackets $\langle\rangle$ denote the thermodynamic average, V represents the potential energy of the cluster, and T is the temperature [6]. The $\frac{3}{2}$ term is added to account for the three translational degrees of freedom that each contribute $\frac{1}{2}k_BT$ to the specific heat [7]. For all simulations, k_B was set to 1 for simplicity.

$$\frac{C_v}{Nk_B} = \frac{3}{2} + \frac{\langle V^2\rangle - \langle V\rangle^2}{Nk_B^2T^2} \tag{2.4}$$

Metropolis Monte-Carlo simulations were run for clusters of 3-56 atoms, and the values of the specific heat were calculated for 25 temperature values as the cluster cooled from 1200 K to 200 K, with ΔT = 40 K. The cluster was allowed to equilibrate for 10^6 steps after the temperature was reduced, then the cluster was sampled for 10^7

steps to obtain the specific heat. The temperature range of 1200 K – 200 K was chosen to allow the atoms in the cluster to sufficiently randomize, then cool down slowly as the melting transition is examined. Examples of the starting and ending configurations for a 13-atom simulation are given in Figure 2.1.

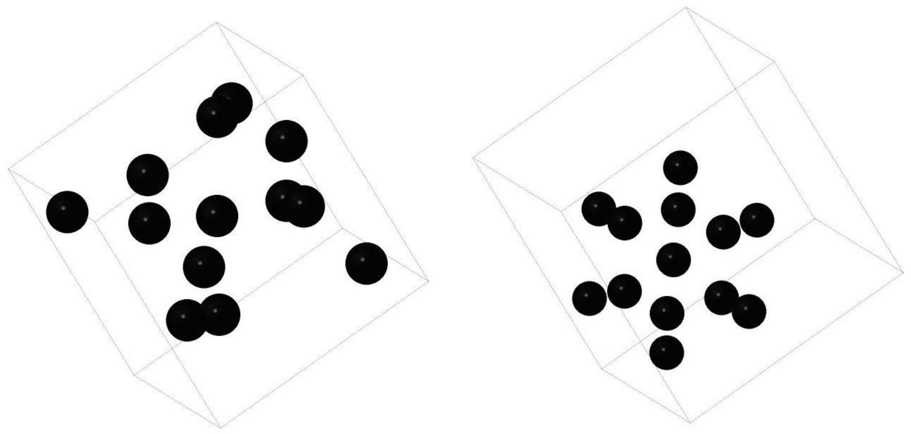

Figure 2.1: *Example of atomic configuration in a 13-atom simulation at the beginning of the simulation (left) and after the cluster has been cooled to its lowest-energy state (right). This picture was generated using Jmol [4].*

Figure 2.2 shows the steps in a Metropolis Monte-Carlo simulation for a given temperature. To begin the simulation, atoms are placed randomly inside a box and the total potential energy of the system is computed. Then, for each step in the simulation, the atoms are randomly displaced to a new configuration, and the new potential energy is computed, as well as the change in energy between the two configurations (ΔE). If ΔE is less than zero, the new configuration is kept because the new configuration represents a lower, more stable energy state. If ΔE is greater than zero, the probability that the new configuration is accepted is given by Eq. 2.5. This probability depends on the temperature of the simulation, as well as the energy difference between the old and new configurations. Thus, as the change in energy becomes large, or the temperature decreases, the probability of accepting a new configuration becomes smaller. Conversely, with a large temperature or a small change in energy, acceptance becomes very likely [1]. This probability creates a Boltzmann distribution of energies in the system.

$$\mu = e^{-\frac{\Delta E}{T}} \tag{2.5}$$

To determine the small displacement of the atoms at each step, each atom is moved by a random distance, the maximum of which is proportional to the square root of the temperature. The proportionality constant λ is chosen to give an acceptance rate of between 40% and 60%.

$$maximum\ displacement\ \delta x = \pm \lambda \sqrt{T} \tag{2.6}$$

The simulation carries out 10^6 steps per temperature value, and for every step either accepts a change in configuration or rejects the change based on the above criteria, also demonstrated in Figure 2.2. An optimal Monte-Carlo simulation accepts between 40% and 60% of the total random changes to the system. A smaller acceptance rate indicates an insufficient number of states being sampled during the simulation. A larger acceptance rate indicates that the displacement of particles is too small, and the simulation method is inefficient. Therefore, the coefficient λ was changed for each simulation to achieve an efficient acceptance rate. Depending on the number of atoms in the simulation and the starting temperature, the optimal value of λ ranged between 10^{-3} and 5×10^{-5}. During the initialization of the program, the

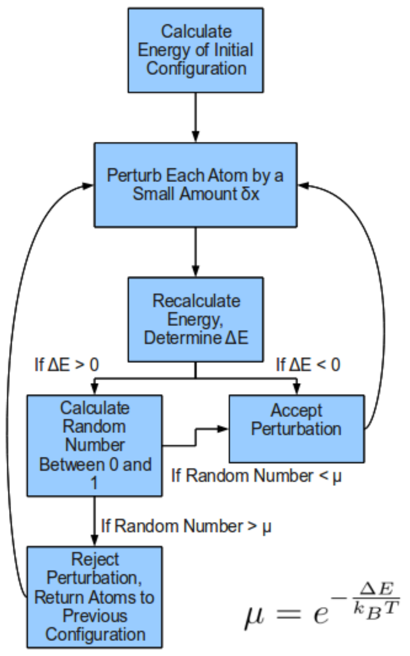
Calculate Energy of Initial Configuration
Perturb Each Atom by a Small Amount δx
Recalculate Energy, Determine ΔE
If ΔE > 0
If ΔE < 0
Calculate Random Number Between 0 and 1
Accept Perturbation
If Random Number < μ
If Random Number > μ
Reject Perturbation, Return Atoms to Previous Configuration
$\mu = e^{-\frac{\Delta E}{k_B T}}$

Figure 2.2: *Flowchart of the Metropolis Monte-Carlo algorithm. New configurations are accepted or rejected based upon the change in energy between the current and previous state.*

value of λ was adjusted to achieve an acceptance rate of approximately 50%. Once the optimal acceptance rate was achieved, the value of λ was held constant through the duration of the simulation.

The atoms in the simulation were constrained inside a box using hard-walled boundary conditions, with perfectly elastic collisions against the walls. Special care was used to determine the size of the constraining box. In a box that is too large, the atoms will be too far spread out, and will take too long to coalesce. In a box that is too small, the final cluster shape will be influenced by pressure effects from the walls and will not accurately reflect the cluster geometry. Examples of poor box sizes are given in Figure 2.3.

Through experimentation, an optimal box size was determined to allow maximal

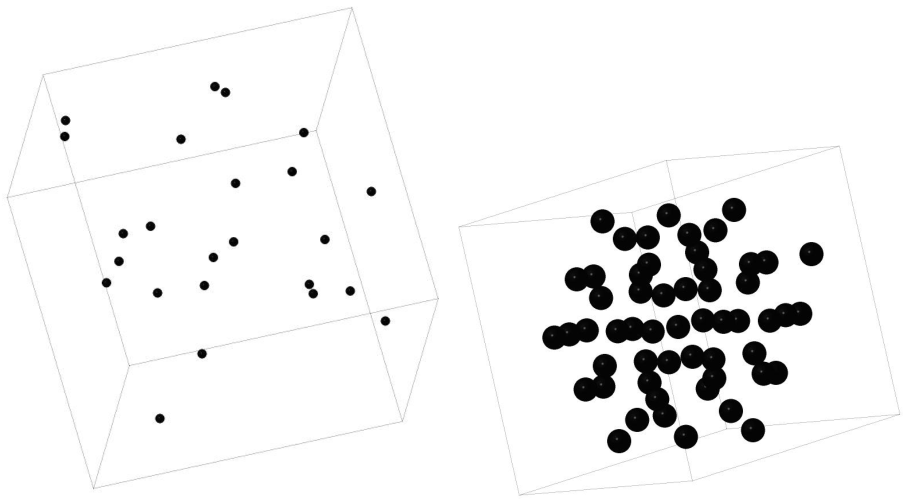

Figure 2.3: *Examples of too large (left) and too small (right) boxes. The left box contains 24 atoms and the right box contains 55 atoms. The left box will take far too long for the atoms to coalesce, and the right box will be influcenced too much by pressure effects from the walls. This picture was generated using Jmol [4].*

amount of freedom for the atoms to move without spacing them too far apart. This box has a side length that is approximately 6 times the average inter-atomic distance (Figure 2.4). This determination agrees with literature findings [6].

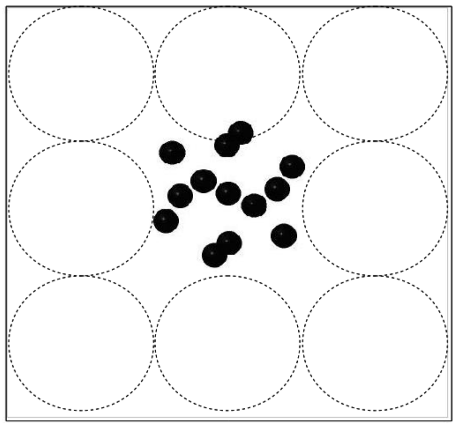

Figure 2.4: *Example of a perfectly-sized box, with a side length that is 6 times the average distance between atoms. This example is a 13-atom cluster. This picture was generated using Jmol [4].*

An additional set of simulations were run to calculate the minimal energy per atom of the cluster at its final, cooled state. Magic number clusters show a local minumum in the energy per atom, and therefore this data can identify magic number clusters as well. For these simulations, clusters of 3-56 atoms were cooled from 2000 K to 0 K in 100 steps, allowing 10^5 steps of equilibration between temperature changes. This equilibration rate allowed the simulations to be done quickly without compromising accuracy. These simulations were finished after 24 hours of computing.

3 Implementation

Originally, the program used in this project was written to perform molecular dynamics simulations of the clusters. However, the program was unable to model constant-temperature interactions without the use of thermostat algorithms to keep the simulated atoms in a Boltzmann energy distribution. Therefore, the program was expanded to allow both molecular dynamics and Monte-Carlo simulations to be run. Eventually, because of their simplicity and speed, Monte-Carlo simulations were exclusively used for modeling the behavior of the clusters, although the program still has the capability to run molecular dynamics simulations as well.

The computations in this project were performed using a combination of Python code and C code, and each language was used to optimize its strengths and avoid its weaknesses. Python is a relatively slow language, and therefore it is used as the master program to initialize the simulation. Python also allows for simple writing to external files, and has a graphics package that is very easy to use. C, on the other hand, is much faster than Python. Depending on the type, a C program can perform 5-10 times the number of tasks a Python program can in the same amount of time. Therefore, C is used in this project for the brute-force computations to determine the total potential energy, which takes N^2 operations, where N is the number of atoms in the cluster, and can become very time-consuming for large clusters.

Although Python code and C code work very well at these tasks independently, interfacing them to work together is difficult. The ability to interface Python and C code, and allow the two programs to talk and exchange data, is not natively built into either language. Extra wrapper files needed to be included when compiling and running the program to ensure compatibility. SWIG (Simplified Wrapper and Interface Generator) was used to generate the wrapper code necessary to allow the Python program to call C functions. Full documentation of all code, and a simple Bash script to run SWIG, is available in the Appendix.

Simulation times using this program ranged from several minutes for a 3-atom cluster to approximately 24 hours for a 55-atoms cluster. Because of the significant time and processor commitment to run these programs, the jobs were submitted on the Quipu cluster, located in Weniger Hall. Running the programs on an external server not only allowed for continuous runtime, but the fast processor speed of the Quipu server computers allowed the simulations to be completed faster as well. Simulations of 3-56 atoms using 5 separate nodes on the Quipu cluster required approximately 3 days to complete.

In order to ensure accurate data in the Silver clusters, the simulation was first tested with Argon clusters and the results were compared to published literature [3]. Agon has LJ-parameters of $\epsilon = 119.4$ K, and $\sigma = 3.405$ Å[3]. Cluster sizes with unique specific heat behavior were chosen for comparison, such as the 55-atom cluster, which has an unusually high and sharp peak at the melting temperature (Fig. 2.3). The validity of this program can be verified if the simulations match the published results. For the 55 atoms cluster, both the height and the location of the peak on the specific heat graph were compared (Fig. 2.5). In all instances, the graphs were nearly identical, confirming the validity of the results obtained from this program.

The positions of the atoms were also recorded throughout the simulation in XYZ

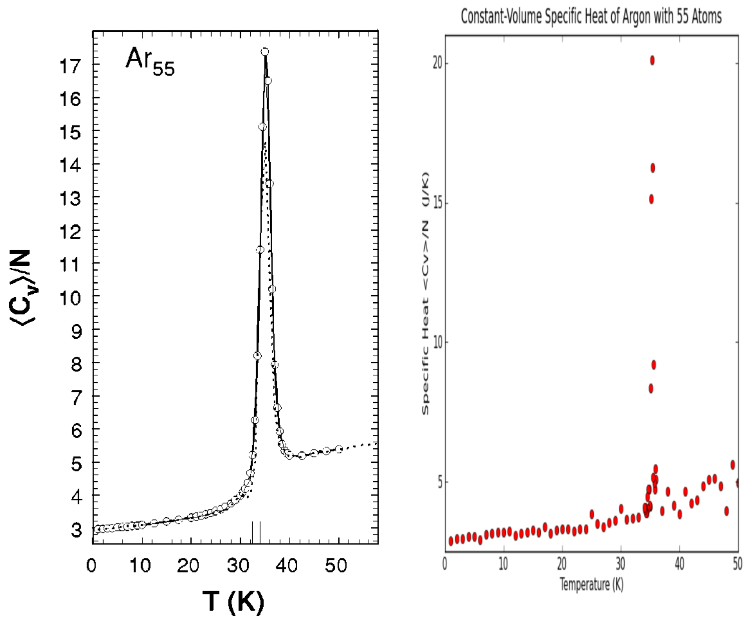

Figure 3.1: *Comparison of specific heat graphs of a 55 atom Argon cluster from literature [3] (left) and computed (right). Comparison shows similar qualitative behavior, as well as agreement in peak temperature (35 K) and peak height (20 J/K).*

format to later be read by Jmol, a program that creates pictures of the cluster given the x, y, and z coordinates of each atom [4]. This technique was another way of ensuring accuracy by viewing the pictures of the cluster at the beginning and the end of the melting transition. At temperatures higher than the melting transition, the cluster is disordered and widely dispersed. At temperatures lower than the melting transition, the cluster is ordered and the atoms are closely packed together.

4 Results

Two sets of simulations were run to determine the specific heat curves of 3-56 atom silver clusters. The first set of simulations were done over a wide temperature range, from 0 - 2000 K, taking 25 equally-spaced data points. This wide range established the general temperature at which the melting transition occurred, although the resolution was low due to the large range. Analysis of this wide range simulation generated information which allowed another set of simulations to be run at a narrower temperature range, 200-1200 K, to show the peaks with greater resolution. Figure 4.1 shows an example wide temperature-range graph of the specific heat for the 13 atoms cluster.

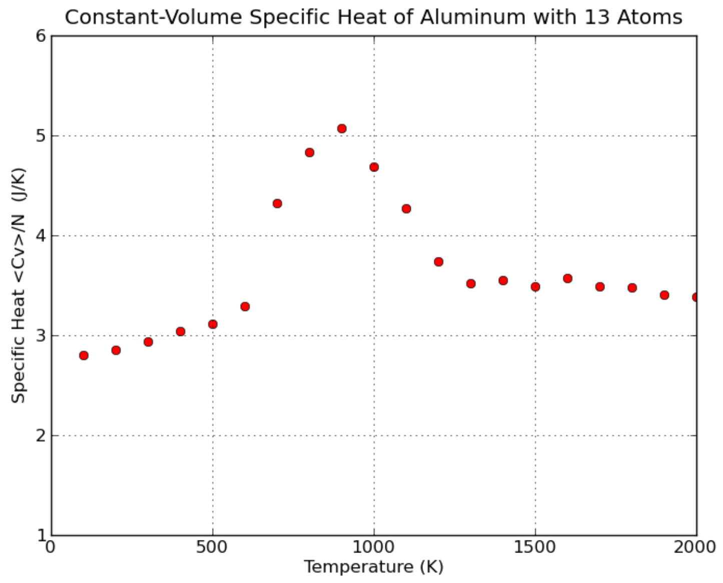

Figure 4.1: *Specific heat curve for a 13-atom silver cluster. The melting peak is broad but well defined. It is clear that the transition occurs between 200 and 1200 K, allowing for another simulation between these temperatures to achieve even greater accuracy.*

Additionally, along with narrowing the temperature range, the number of Monte

Carlo steps per temperature value was doubled for the narrow-range simulations to improve the accuracy of the peaks further. Figure 4.2 shows the 13-atom silver cluster simulated using the narrowed temperature range. Comparison of Figure 2 to the wide temperature results shows agreement at the endpoints (200 K and 1200 K), as well as a more defined melting curve.

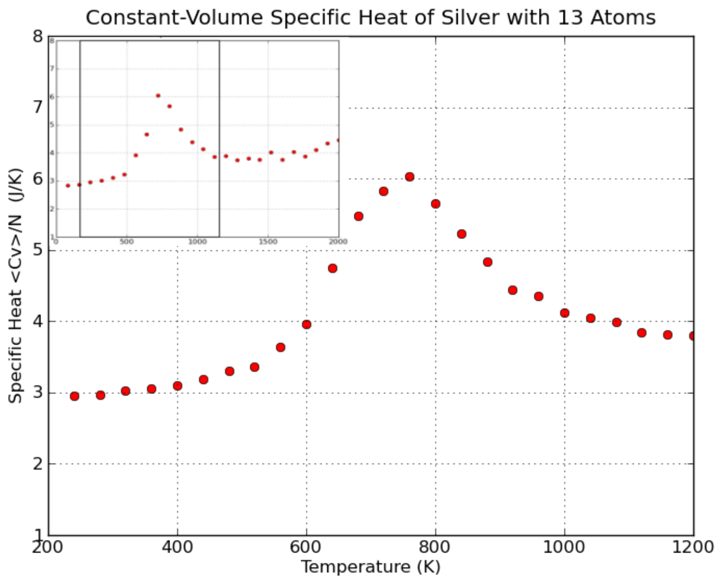

Figure 4.2: *Specific heat curve of 13-atom silver cluster under narrowed temperature range. The inset image shows the wide temperature results, and the box indicates the area 'zoomed in on' by the narrowed temperature.*

Of particular interest in these simulations is the behavior of the 'magic number' clusters, those clusters with symmetry in their final state configurations. Because of the symmetry and compact geometry, magic number clusters require a large latent heat to cause a phase change. Calculating the latent heat per atom by integrating the specific heat (Eq. 4.1) would provide an accurate technique to find magic clusters. However, this integral is difficult to compute numerically, and therefore the peak height is used as an analog to the latent heat to identify the magic number clusters.

$$L = \int_{T_i}^{T_f} C_v dT \tag{4.1}$$

The icosaheral magic number clusters are those with 13 and 55 atoms, and these clusters show a particularly large melting peak. Figure 4.3 shows a superposition of melting curves for a 12-atom, 13-atom, and 14 atom silver cluster over the narrowed temperature range. It is clear that the 13-atom cluster has a much higher melting peak than either the 12-atom or 14-atom cluster, identifying it as an icoshaderal magic number cluster. Figure 4.4 also shows a superposition of the melting curves for a 54-atom, 55-atom, and 56-atom cluster. Although not as clearly defined as figure 4.3, the peak of the 55-atom cluster is still identifiable as the largest peak on the graph, showing that the 55-atom cluster is also an icosahedral magic number cluster.

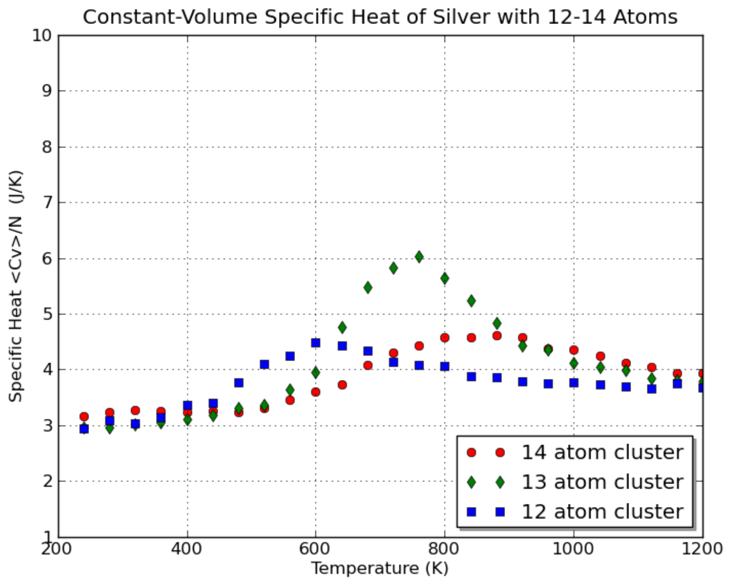

Figure 4.3: *Superposition of 12, 13 and 14-atom cluster melting graphs. Clearly the 13-atom cluster is identified as a magic number cluster due to the local maximum of its melting peak.*

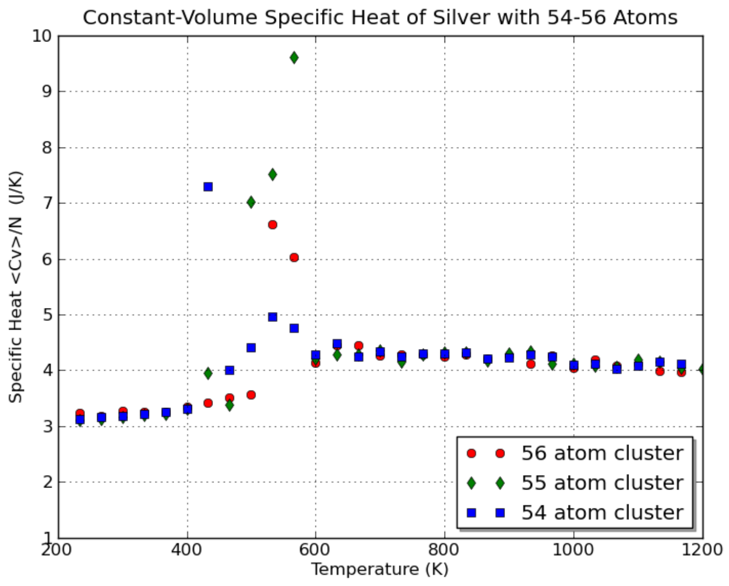

Figure 4.4: *Superposition of 54, 55 and 56-atom cluster melting graphs. The 55-atom cluster is identified as a magic number cluster due to the local maximum of its melting peak.*

The 13 and 55-atom clusters have high melting peaks due to their icosahedral 5-fold symmetry. Rotating the cluster one-fifth of a full turn (72°) along any of the 6 main axes produces the same figure again. Though the 13 and 55-atom clusters show the most stability with 5-fold symmetry, there are other semi-magic number clusters with high degrees of stability. Figure 4.5 shows the peak heights of the specific heat curves for clusters between 5 and 56 atoms. Clearly the N = 13 and N = 55 clusters stand out with the highest local peaks, however the 19 atom cluster also shows a peak.

Results from the energy minimization simulations identify one more semi-magic number cluster, along with confirming the magic number clusters found using the specific heat data. Figure 4.6 shows the minimum energy per atom versus cluster size. The general trend of the graph shows a decrease in the energy per atom as

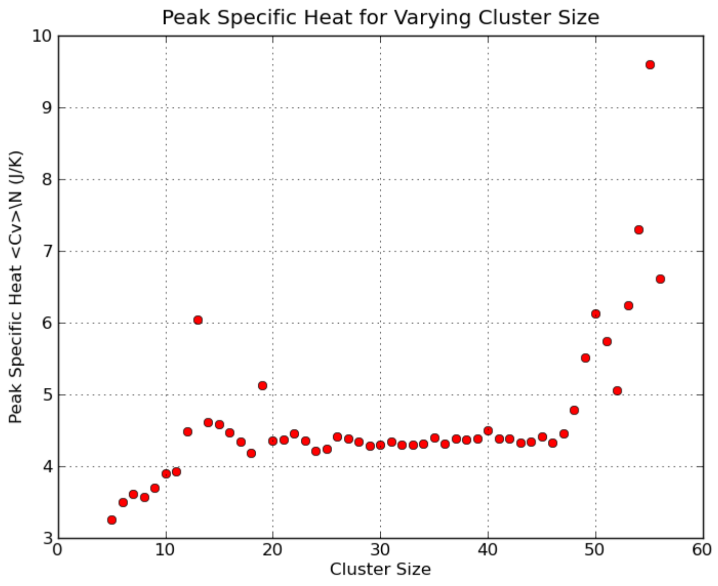

Figure 4.5: *The peak heights of the specific heat curve for varying cluster sizes. Local maxima indicate 'magic number' cluster sizes in which the cluster is particularly stable, and has a symmetrical geometry.*

cluster size increases. However, local minima in this graph indicate stable geometric structures where the decrease in energy is slightly higher than neighboring clusters, and therefore indicate magic numbers. Clearly the 13, 19, and 55 atom clusters show local minima, confirming the results from the specific heat simulations.

Another way to visualize this data is to graph the energy difference between clusters with N and N-1 atoms. Particularly stable clusters will exhibit a large decrease in energy by adding one more atom, and thus will be local minima on a graph of energy difference vs. cluster size (Figure 4.7). The 13, 19, and 55 atom clusters are the most prominent minima, however N=25 also shows a distinct minimum, indicating that it is also a semi-magic number cluster. Other minima of the graph are within the error of the experiment, and cannot conclusively be proven to be semi-magic numbers.

The geometry of the 13 atom cluster is obviously compact, with a single shell of

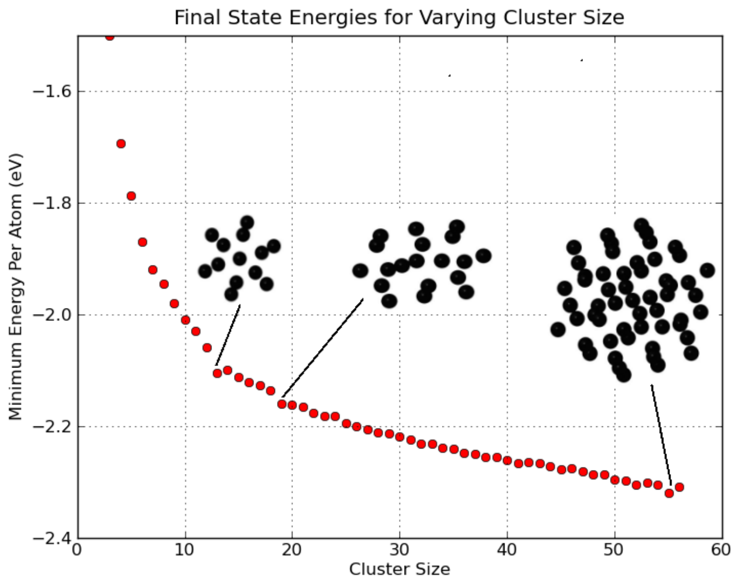

Figure 4.6: *Minimum energy per atom for clusters of 3-55 atoms. Local minima in this graph indicate the presence of magic number clusters, and the 13, 19, and 55 atom cluster are highlighted with pictures of the stable geometry.*

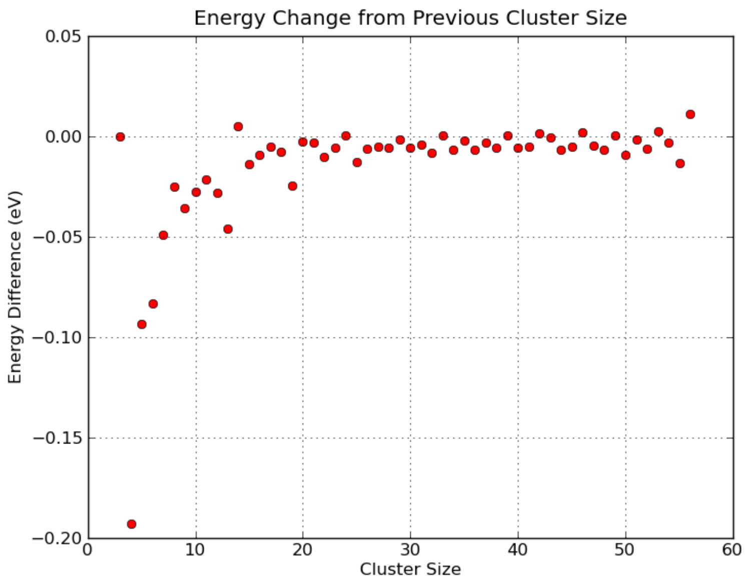

Figure 4.7: *Energy difference between the N atom and N-1 atom clusters, indicating the decrease in total energy from adding one atom. Local minima indicate magic numbers with stable geometry, N = 13, 19, 25, 55.*

atoms surrounding one atom in the middle. However, the geometry of the 19, 25 and 55 atom clusters are not quite as obvious, and require some explanation to identify the symmetry.

The 19 atom cluster is simply a 13 atom cluster with another 'half-shell' added to one end. Figure 4.8 shows the 19 atom cluster, with the original 13 atoms highlighted in red, and the 'added' six in black. This form has 5-fold symmetry only along the long axis, and 2-fold symmetry along the 5 short axes. Therefore, the cluster is not a true icosahedral magic number cluster, although it still has compact, symmetric geometry.

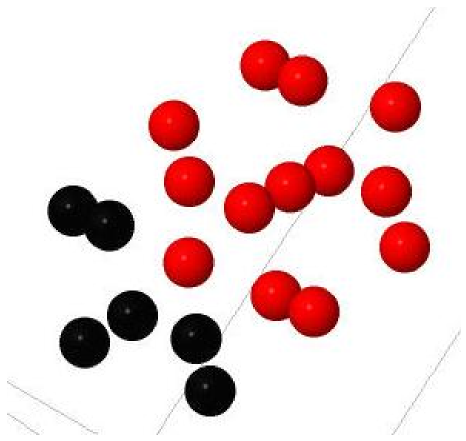

Figure 4.8: *Final energy state of the 19 atom cluster, with the 13 atom cluster highlighted in red to emphasize the symmetry of the cluster. This picture was generated using Jmol [4]*

Interestingly, the lowest energy state of the 25 atom cluster is not another 6 atom 'half shell' added to the end of the 19 atom cluster. The shape is entirely different, with the half shell added to the middle of the cluster. Because the 6 atoms are added to the middle of the 19 atom 'tube', the cluster maintains two axes with 2-fold symmetry, and therefore is a semi-magic number cluster as well.

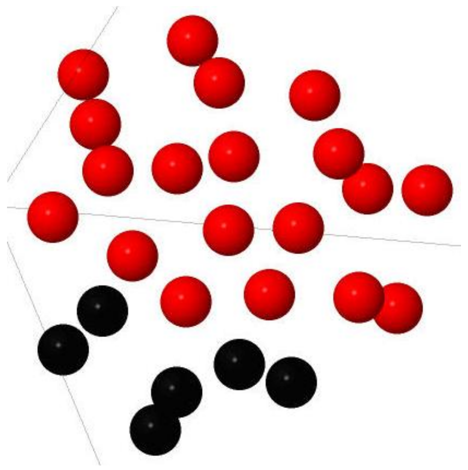

Figure 4.9: *Final energy state of the 25 atom cluster, with the 19 atom cluster highlighted in red. The black atoms, the added half shell, are positioned in the middle of the 19 atom 'tube', creating two planes of symmetry. This picture was generated using Jmol [4]*

The next icosahedral magic number cluster after N=13 is the 55 atom cluster. Figure 4.10 shows the 55 atom cluster as simply another shell of 42 atoms surrounding the 13 atom cluster. Like the 13 atom cluster, this also has complete 5-fold symmetry, and is a perfect icosahedron.

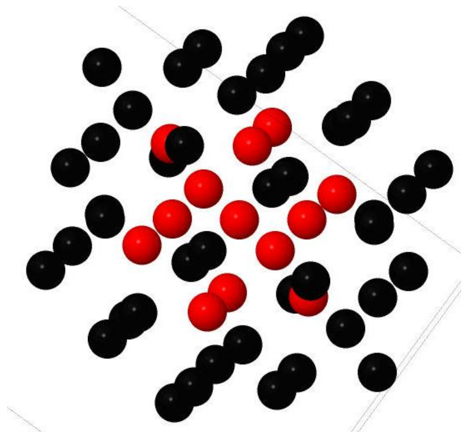

Figure 4.10: *Final energy state of the 55 atom cluster, with the 13 atom cluster highlighted in red. This cluster is formed by added 42 atoms to the exterior of the 13 atom cluster, creating another complete shell with identical symmetry. This picture was generated using Jmol [4]*

5 Conclusion

The 13 and 55 atom clusters were identified as magic number clusters from both the specific heat simulations and the binding energy per atom. Additionally, the specific heat simulations identified a 19-atom semi-magic number cluster, and the binding energy results identified a 25-atom semi-magic number cluster.

The 25 atom semi-magic number cluster found using the binding energy per atom does not show a heightened peak in the graph of specific heat vs. temperature. Although it has a small amount of stability in the final, cooled state, perhaps this stabilty is not enough to translate to a melting peak. Also, perhaps the resolution of the melting simulation is not fine enough to capture this melting peak, or perhaps the simulation was not run for long enough to resolve this peak.

For further results, incorporation of parallel programming into this project is essential. Parallel programming interfaces many computers together to work collectively, and allows the simulations to be conducted faster, allowing for more data points to be taken, as well as longer equilibration and sample times to give cleaner data. Also, with more processing power, clusters of larger than 55 atoms can be studied in order to identify more magic number clusters.

Also, other parameters of the cluster could be studied in further projects to identify and confirm the existance of magic number clusters. The Berry parameter is commonly studied in small atomic clusters, and is a measure of the reordering of pairs of atoms in the cluster's lowest energy state [6]. This parameter shows a notable increase at the melting transition, and could possibly be used to identify magic number clusters. The Berry parameter was orginially intended to be used in this project, however time constraints did not allow this data to be collected.

Finally, this program could be adapted to simulate clusters of several thousand atoms, in an attempt to identify the size at which the transition between cluster melting behavior and bulk melting behavior occurs. This change would take a large amount of restructuring, and would require many approximations, as well as parallel programming, to decrease the processor load. Otherwise, simulations could take years for the largest clusters.

References

[1] Allen, M. P.; Tildesley, D.J. ”Computer Simulation of Liquids.” Oxford Unity Press, 1987. Print.

[2] Baletto F, Ferrando R. ”Structural properties of nanoclusters: Energetic, thermodynamic, and kinetic effects.” Rev Mod Phys, 2005, 77: 371423

[3] Frantz, D.D. ”Magic number behavior for heat capacities of medium-sized classical Lennard-Jones clusters.” Journal of Chemical Physics 115.13 (2001): 6136-57

[4] Jmol: an open-source Java viewer for chemical structures in 3D. http://www.jmol.org/

[5] Puri, P; Yang, V. ”Effect of Particle Size on Melting of at Nano Scales .” Journal of Chemical Physics 111 (2007): 11776-83

[6] Werner, R. ”Melting and evaporation transitions in small Al clusters: canonical Monte-Carlo simulations.” The European Physical Journal B. 43 (2005): 47-52

[7] Wolfle, P., et al. ”Lowering of Surface Melting Temperature in Atomic Clusters with a nearly Closed Shell Structure.” Physical Review B 81.7 (2010)Print.

[8] Gupta, Raju. ”Lattice Relaxation at a Metal Surface.” Physical Review B 23 (1981)

Appendices

ClusterSim_Interface.py

```
import random
from ClusterSim import *
import pickle
import time
import sys
import math
import random
import numpy as np
import matplotlib.pyplot as plt

# Main program used to control the simulation. ClusterSim.c
# provides and implements the functions that are called from
# this python program. This program is easliy modified to
# change the parameters of the simulation.

random.seed()

# Number of equilibration steps and number of sampling
# steps used in monte carlo simulation
n_eq = 1000000
n_sample = 20000000

num_atoms = int(sys.argv[1])

# Timestep given for MD simulation
dt = 0.0001

#Starting and ending temperatures, as well as number
# of steps to carry out, are given from command line
Tstart = float(sys.argv[2])
Tend = float(sys.argv[3])
steps = int(sys.argv[4])
h = (Tstart - Tend)/steps

mass = 13.0

prefactor = 0.000

# Potential parameters for both Lennard Jones and
# Gupta Potential are given
```

```
#Silver
# Gupta
A = 1192.94
xi = 12670.107
p = 10.928
q = 3.139
r0 = 2.892
#Lennard Jones
sig = 1
eps = 2000

"""

#Aluminum
#Gupta
A = 1416.91
xi = 15271.52
p = 8.612
q = 2.516
r0 = 2.864
#Lennard Jones
sig = 2.62
eps = 4551
"""
# Calculate box size from Gupta parameters
min_dist = r0*3

# Values for specific heat and temperature
# are stored in arrays, and appended at
# each step of simulation
Cv = []
Temps = []

# Writes the position of the cluster in
# xyz format to external file
def Write():
  f.write(str(num_atoms))
  f.write('\n\n')
  i = 0
  while(i<num_atoms):
    arg = "C  %s   %s   %s\n" % (getPx(i), getPy(i), getPz(i))
    f.write(arg)
    i += 1

# Checks to make sure atoms are not too close at start of MD program.
# Close atoms can cause errors in the program, as forces become
# very large.
```

```
def CheckClose(num):
  flag = True
  while(flag == True):
    flag = False
    i = 0
    while(i < num):
      j = 0
      while(j < i):
        dist = getDist((getPx(j)-getPx(i)),(getPy(j)-getPy(i)),(getPz(j)-getPz(i)))
        if(dist < min_dist / 2.0):
          setValues(i, mass)
          flag = True
        j += 1
      i += 1

# Defines the prefactor of the monte carlo simulation,
# adjusting the value until the acceptance rate is close to 50%.
def DefinePrefactor(T):
  global prefactor
  while(Monte_Carlo_Accept(500,num_atoms, T, prefactor) > 0.6):
    prefactor += 0.0001
    print prefactor, Monte_Carlo_Accept(500,num_atoms, T, prefactor)

  while(Monte_Carlo_Accept(500,num_atoms, T, prefactor) < 0.4):
    prefactor -= 0.0001
    print prefactor, Monte_Carlo_Accept(500,num_atoms, T, prefactor)


####################   MAIN   ############################

# Opens files for storing the positions in xyz format
# for viewing, as well as storing the specific heat data.
f = open('movieSilver' + str(num_atoms) + '.xyz', 'w')
g = open('SpecificHeatSilverGupta' + str(num_atoms) + '_' +
str(sys.argv[3]) + '-' + str(sys.argv[2]) + 'steps_' + str(sys.argv[4]), 'w')
# g = open('Energies' + str(num_atoms), 'w')

# Sets constants for simulation
setConstants(min_dist, dt, sig, eps, A, xi, p, q, r0)

# Gives each atoms an initial position and velocity
i = 0
while(i<num_atoms):
  setValues(i, mass)
  i += 1

# CheckClose(num_atoms)
# Defines the current temperature as the starting temperature
T = Tstart

DefinePrefactor(T)

# Writes the starting and ending temperature to the first
# values in the specific heat file. Used for plotting later.
g.write(("%f  \n%f  \n" % (Tstart, Tend)))

# For each step until the temperature reaches the ending temperature,
# simulate for given steps, then sample for given steps, finally
# write position and specific heat data to respective files
while(T > Tend):
  sim = Monte_Carlo(n_eq, num_atoms, T, prefactor)
  SH =  Monte_Carlo(n_sample, num_atoms, T, prefactor)
  g.write(("%f  \n%f  \n" % (T, SH)))
  Write()

# Redefine the prefactor to ensure approximately 50% acceptance rate
  DefinePrefactor(T)
  T = T - h
```

ClusterSim.c

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

//Set Variables for Simulation

double min;
double dt;

double sig;
double eps;
double A;
double xi;
double p;
double q;
double r0;

// Stores x, y, and z positions of each atom
double posx[100000];
double posy[100000];
double posz[100000];

// Stores x, y, and z velocities of each atom
double vx[100000];
double vy[100000];
double vz[100000];

//Forces on each atom at current step, and from previous step, using MD
double foldx[100000];
double foldy[100000];
double foldz[100000];
double Fx[100000];
double Fy[100000];
double Fz[100000];

//Stores mass of each atom
double mass[100000];

//Initialize constants

void setConstants(double mindist, double Dtime, double s, double e, double a, double x,
double P, double Q, double R0)
{
srand(time(0));  //Randomize seed
min = mindist; //Box size
dt = Dtime; //Time step for MD

//Lennard Jones Constants
sig = s;
eps = e;

//Gupta Constants
A = a;
xi = x;
p = P;
q = Q;
r0 = R0;
}

//Sets the initial position and velocity for each atom in MD simulation
void setValues(int index, double m)
{
posx[index] = ((double)rand() / (double)(RAND_MAX))*2*min - min;
posy[index] = ((double)rand() / (double)(RAND_MAX))*2*min - min;
posz[index] = ((double)rand() / (double)(RAND_MAX))*2*min - min;
vx[index] = 0;
vy[index] = 0;
vz[index] = 0;
foldx[index] = 0;
foldy[index] = 0;
foldz[index] = 0;
mass[index] = m;
}

//Returns a random variable with a gaussian distribution depending on
//given mean and STDEV
double Gauss(double mean, double stdev)
{
double x,y,r;
do {
x = 2.0*rand()/RAND_MAX - 1;
y = 2.0*rand()/RAND_MAX - 1;
r = x*x + y*y;
} while(r==0.0 || r > 1);
double d = sqrt(-2.0*log(r)/r);
double n1 = x*d;

double result = n1*stdev + mean;

return result;
}
```

```
//Readjusts velocities of atoms into a gaussian distribution. Anderson Thermostat
void Thermostat(int index, int numatoms, double temp)
{
vx[index] = Gauss(0, sqrt(temp/mass[index]));
vy[index] = Gauss(0, sqrt(temp/mass[index]));
vz[index] = Gauss(0, sqrt(temp/mass[index]));
}

//Returns the distance between two atoms given their
//differences in x, y, and z position
double getDist(double dx, double dy, double dz)
{
return sqrt(dx*dx + dy*dy + dz*dz);
}

//Calculates force between atoms in x direction
double forcex(double dx, double dist)
{
double fx;
fx = 4*eps*pow(sig,6)*(6.0*dx)/(pow(dist,8)) -
4*eps*pow(sig,12)*(12.0*dx)/(pow(dist,14)); //Force calculated from LJ potential
return fx;
}

//Calculates force between atoms in y direction
double forcey(double dy, double dist)
{
double fy;
fy = 4*eps*pow(sig,6)*(6.0*dy)/(pow(dist,8)) -
4*eps*pow(sig,12)*(12.0*dy)/(pow(dist,14)); //Force calculated from LJ potential
return fy;
}

//Calculates force between atoms in z direction
double forcez(double dz, double dist)
{
double fz;
fz = 4*eps*pow(sig,6)*(6.0*dz)/(pow(dist,8)) -
4*eps*pow(sig,12)*(12.0*dz)/(pow(dist,14)); //Force calculated from LJ potential
return fz;
}

//Returns x, y, and z positions of atoms with given index

double getPx(int index)
{
```

```
return posx[index];
}

double getPy(int index)
{
return posy[index];
}

double getPz(int index)
{
return posz[index];
}


//Returns x, y, and z velocities of atoms with given index

double getVx(int index)
{
return vx[index];
}

double getVy(int index)
{
return vy[index];
}

double getVz(int index)
{
return vz[index];
}


//Calculates the force on each atom prior to the simulation starting.
//This initial force is used to step each atom forward one timestep before
//the simulation begins, providing two force values for the iteration algorithm
void initialForce(int numatoms)
{
int count1 = 0;
for(count1=0;count1<numatoms;count1++) {
  int count2 = 0;
  foldx[count1] = 0;
  foldy[count1] = 0;
  foldz[count1] = 0;
  for(count2=0;count2<numatoms;count2++) {
    if(count1 != count2) {
        double dist = getDist((posx[count2]-posx[count1]),(posy[count2]-posy[count1]),
        (posz[count2]-posz[count1]));
        foldx[count1] = foldx[count1] + forcex((posx[count2]-posx[count1]),dist);
        foldy[count1] = foldy[count1] + forcey((posy[count2]-posy[count1]),dist);
        foldz[count1] = foldz[count1] + forcez((posz[count2]-posz[count1]),dist);
        }
  }
}
}

//Check if any atom is outside the simulation box, and reverse its
//direction if it is outside the box
void checkBounds(int count)
{
{
        if (posx[count] <= -min)
                 {
                vx[count] = -1.0*(vx[count]);
                 }
        else if (posx[count] > min)
                 {
                vx[count] = -1.0*(vx[count]);
                }
        else if (posy[count] <= -min)
                {
                vy[count] = -1.0*(vy[count]);
                 }
        else if        (posy[count] > min)
                 {
                vy[count] = -1.0*(vy[count]);
                 }
        else if (posz[count] <= -min)
                {
                vz[count] = -1.0*(vz[count]);
                }
        else if (posz[count] >  min)
                {
                vz[count] = -1.0*(vz[count]);
                }
        else
                {
                }
}
}

//Prints the position and velocity of each atom. For debugging purposes.
void print_data(num) {
int i;
for(i=0;i<num;i++) {
  printf("%s  %i  %f  %f   %f  \n" , "atom num pos", i, posx[i], posy[i], posz[i]);
  printf("%s  %i  %f  %f   %f  \n" , "atom num vel", i, vx[i], vy[i], vz[i]);
  }
}

//Returns the total potential energy of the cluster. Both potentials are given,
//commenting out the one that is not used.
double get_PE(num) {

//Gupta

double PE = 0;
int i;
int j;
for(i=0;i<num;i++) {
  double fac1 = 0;
  double fac2 = 0;
  for(j=0;j<num;j++) {
    if(i!=j) {
      double dist = (getDist((posx[j]-posx[i]),(posy[j]-posy[i]),
      (posz[j]-posz[i])))/r0 - 1;
      fac1 += A*exp(-p*dist);
      fac2 += xi*xi*exp(-2*q*dist);
      }
    }
  PE += fac1 - sqrt(fac2);
  }

return PE;

// LENNARD JONES
/*
double PE = 0;
int i;
int j;
for(i=0;i<num;i++) {
  for(j=0;j<i;j++) {
    if(i!=j) {
      double dist = getDist((posx[j]-posx[i]),(posy[j]-posy[i]),
      (posz[j]-posz[i]));
      double tw = pow((sig/fabs(dist)),12);
      double sx = pow((sig/fabs(dist)),6);
      PE = PE + 4*eps*(tw - sx);
      }
    }
  }
return PE;
```

```
*/
}

//Returns the kinetic energy. Used in MD simulation.
double get_KE(num) {
double KE = 0;
int i;
for(i=0;i<num;i++) {
  KE = KE + 0.5*mass[i]*(getVx(i)*getVx(i)+getVy(i)*getVy(i)+getVz(i)*getVz(i));
}

return KE;

}

//Modifies velocities. Used to 'cool' cluster by simple velocity rescaling.
void update_vel(int num, double rate) {
int i;
for(i=0;i<num;i++) {
  vx[i] = vx[i] * rate;
  vy[i] = vy[i] * rate;
  vz[i] = vz[i] * rate;
}
}

//Writes the starting configuration to an xyz file.
void generate_xyz_start(int num) {
FILE *file;
file = fopen("start.xyz","w");
fprintf(file,"%i \n\n", num);
int i;
for(i=0;i<num;i++) {
  fprintf(file, "%s  %f  %f  %f  \n", "C", posx[i], posy[i], posz[i]);
}
fclose(file);
}

//Writes the ending configuration to an xyz file.
void generate_xyz_end(int num) {
FILE *file;
file = fopen("end.xyz","w");
fprintf(file,"%i \n\n", num);
int i;
for(i=0;i<num;i++) {
  fprintf(file, "%s  %f  %f  %f  \n", "C", posx[i], posy[i], posz[i]);
}
fclose(file);
```

```
}

//Main MD simulation loop. Iterates for a given number of steps at a temperature T,
//constantly rescaling velocities using the thermostat algorithm.
void iterate(int steps, int numatoms, double T)
{
int i, count1, count2, j;

for(i=0;i<steps;i++)
{
  for(j=0;j<numatoms;j++)
  {
  Fx[j] = 0;
  Fy[j] = 0;
  Fz[j] = 0;
  }

  for(count1=0;count1<numatoms;count1++)
  {
        //Main velocity-verlet algorithm. Updates positions of atoms
        //based on velocities and previous forces.
        posx[count1] = posx[count1] + vx[count1]*dt/mass[count1] -
        dt*dt*foldx[count1]/2.0;
        posy[count1] = posy[count1] + vy[count1]*dt/mass[count1] -
        dt*dt*foldy[count1]/2.0;
        posz[count1] = posz[count1] + vz[count1]*dt/mass[count1] -
        dt*dt*foldz[count1]/2.0;

  for(count2=count1;count2<numatoms;count2++)
  {
  if(count1 != count2)
        {
        //Recalculate forces between each pair of atoms.
        double dist = getDist((posx[count2]-posx[count1]),
        (posy[count2]-posy[count1]),(posz[count2]-posz[count1]));
        if(dist < 6.0*sig) {
        Fx[count1] += forcex((posx[count2]-posx[count1]),dist);
        Fx[count2] -= Fx[count1];
        Fy[count1] += forcey((posy[count2]-posy[count1]),dist);
        Fy[count2] -= Fy[count1];
        Fz[count1] += forcez((posz[count2]-posz[count1]),dist);
        Fz[count2] -= Fz[count1];
        }
        }
  }
        //Make sure boundary conditions are enforced
        checkBounds(count1);
        //Recalculate velocities using previous velocity as well as
        //current and previous force, averaged.
        vx[count1] = vx[count1] + (dt*((Fx[count1]+foldx[count1])/2.0));
        vy[count1] = vy[count1] + (dt*((Fy[count1]+foldy[count1])/2.0));
        vz[count1] = vz[count1] + (dt*((Fz[count1]+foldz[count1])/2.0));

        //Current force becomes previous force for next iteration.
        foldx[count1] = Fx[count1];
        foldy[count1] = Fy[count1];
        foldz[count1] = Fz[count1];

        //Rescale velocities using thermostat to keep at constant temperature
        double randomnum = 1.0*rand()/RAND_MAX;
        if(randomnum < 0.00005) {
          Thermostat(count1, numatoms, T);
        }
}

}

}


//Algorithm to determine the acceptance rate of the monte carlo simulation.
//Identical to real monte carlo algorithm, except for returning the acceptace rate
//instead of measurable data. Used to adjust 'prefactor' for a ~50% acceptance rate.
double Monte_Carlo_Accept(int steps, int numatoms, double T, double prefactor)
{

int i, count;
double totalsteps = 0;
double changes = 0;

double stepsize = prefactor * sqrt(T);

double px_prime[numatoms];
double py_prime[numatoms];
double pz_prime[numatoms];

for(i=0;i<steps;i++)
{
  double PE_old = get_PE(numatoms);
  for(count=0;count<numatoms;count++)
  {
    px_prime[count] = posx[count];
    py_prime[count] = posy[count];
    pz_prime[count] = posz[count];
    posx[count] = posx[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;
    posy[count] = posy[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;
    posz[count] = posz[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;
/// Hard Boundary Conditions ///
    if(posx[count] > min)
      posx[count] = min;
    else if(posx[count] < -min)
      posx[count] = -min;
    if(posy[count] > min)
      posy[count] = min;
    else if(posy[count] < -min)
      posy[count] = -min;
    if(posz[count] > min)
      posz[count] = min;
    else if(posz[count] < -min)
      posz[count] = -min;
  }
  double PE_new = get_PE(numatoms);
  if(PE_new < PE_old)
    changes += 1;
  else
  {
    double ran = 1.0*rand()/RAND_MAX;
    if(exp(-1.0*(PE_new - PE_old)/T) > ran) {
      changes += 1;
    }
    else
    {
      for(count=0;count<numatoms;count++)
      {
        posx[count] = px_prime[count];
        posy[count] = py_prime[count];
        posz[count] = pz_prime[count];
      }
    }
  }
  totalsteps += 1;
}

return changes/totalsteps;

}

//Main monte carlo algorithm. Carries out given number of steps at given temperature T.
//Returns the specific heat of the cluster, using energy averages
//stored over all iterations.
```

```
double Monte_Carlo(int steps, int numatoms, double T, double prefactor)
{

int i, count;
double PE = 0;
double E_Avg = 0;
double E2_Avg = 0;

double stepsize = prefactor * sqrt(T);

double px_prime[numatoms];
double py_prime[numatoms];
double pz_prime[numatoms];

for(i=0;i<steps;i++)
{
  double PE_old = get_PE(numatoms);
  for(count=0;count<numatoms;count++)
  {
    //Store old positions of atoms, move each atom a small amount
    //proportional to T and the prefactor.
    px_prime[count] = posx[count];
    py_prime[count] = posy[count];
    pz_prime[count] = posz[count];
    posx[count] = posx[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;
    posy[count] = posy[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;
    posz[count] = posz[count] + (2.0*rand()/RAND_MAX - 1.0)*stepsize;


/// Hard Boundary Conditions ///
    if(posx[count] > min)
      posx[count] = min;
    else if(posx[count] < -min)
      posx[count] = -min;
    if(posy[count] > min)
      posy[count] = min;
    else if(posy[count] < -min)
      posy[count] = -min;
    if(posz[count] > min)
      posz[count] = min;
    else if(posz[count] < -min)
      posz[count] = -min;
  }

  //Recalculate potential energy. Depending on new value, accept
  //or reject the perturbation.
  double PE_new = get_PE(numatoms);
  if(PE_new > PE_old)
  {
    double ran = 1.0*rand()/RAND_MAX;
    if(exp(-1.0*(PE_new - PE_old)/T) < ran)
    {
      //Perturbation is rejected. Restore atoms to original positions.
      for(count=0;count<numatoms;count++)
      {
        posx[count] = px_prime[count];
        posy[count] = py_prime[count];
        posz[count] = pz_prime[count];
      }
      E_Avg += PE_old;
      E2_Avg += PE_old*PE_old;
    }
    else {
      //Perturbation is accepted. Keep configuration and use the
      //new energy to calculate observables.
      E_Avg += PE_new;
      E2_Avg += PE_new*PE_new;
    }
  }
  else {
    E_Avg += PE_new;
    E2_Avg += PE_new*PE_new;
  }
}

//Determine averages of E and E^2
E_Avg = E_Avg / steps;
E2_Avg = E2_Avg / steps;

//Return Cv
return (1.0/(numatoms*T*T))*(E2_Avg - (E_Avg*E_Avg)) + 1.5;

}
```

ClusterSim.h

```
#ifndef CLUSTERSIM_H
#define CLUSTERSIM_H

extern void   setValues(int index, double m);
extern double getDist(double dx, double dy, double dz);
extern double getPx(int index);
extern double getPy(int index);
extern double getPz(int index);
extern double getVx(int index);
extern double getVy(int index);
extern double getVz(int index);
extern void   iterate(int steps, int numatoms, double T);
extern double forcex(double dx, double dist);
extern double forcey(double dy, double dist);
extern double forcez(double dz, double dist);
extern void   initialForce(int numatoms);
extern void   checkBounds(int numatoms);
extern double get_PE(int num);
extern double get_KE(int num);
extern void generate_xyz_start(int num);
extern void generate_xyz_end(int num);
extern void print_data(int num);
extern void setConstants(double mindist, double Dtime, double s, double e, double a, dou
extern void update_vel(int num, double rate);
extern double Gauss(double mean, double stdev);
extern void Thermostat(int index, int numatoms, double temp);
extern double Monte_Carlo_Accept(int steps, int numatoms, double T, double prefactor);
extern double Monte_Carlo(int steps, int numatoms, double T, double prefactor);

#endif
```

ClusterSim.i

```
%module ClusterSim
%{
#include "ClusterSim.h"
%}

double forcex(double dx, double dist);
double forcey(double dy, double dist);
double forcez(double dz, double dist);
double getDist(double dx, double dy, double dz);
double getPx(int index);
double getPy(int index);
double getPz(int index);
double getVx(int index);
double getVy(int index);
double getVz(int index);
void   initialForce(int numatoms);
void iterate(int steps, int numatoms, double T);
void setValues(int index, double m);
void checkBounds(int numatoms);
double get_PE(int num);
double get_KE(int num);
void generate_xyz_start(int num);
void generate_xyz_end(int num);
void print_data(int num);
void setConstants(double mindist, double Dtime, double s, double e, double a, double x,
double P, double Q, double R0);
void update_vel(int num, double rate);
double Gauss(double mean, double stdev);
void Thermostat(int index, int numatoms, double temp);
double Monte_Carlo_Accept(int steps, int numatoms, double T, double prefactor);
double Monte_Carlo(int steps, int numatoms, double T, double prefactor);
```

runClusterSim

runClusterSim

```
#!/bin/bash

# Small bash script to compile and submit simulations. The first three commands
# create the wrapper code that is necessary to interface the Python program with
# the C code. The last command simply runs the Python program with the given parameters.
# The first parameter declares the number of atoms, the second declares the
# starting temperature, the third declares the final temperature, and the
# last parameter declares the number of temperature values to sample.

swig -python ClusterSim.i
gcc -fpic -I/usr/include/python2.6 -c ClusterSim.c ClusterSim_wrap.c
gcc -shared -o _ClusterSim.so ClusterSim.o ClusterSim_wrap.o

python ClusterSim_Interface.py 13 1200 200 25
```