

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART  
ALBERT NERKEN SCHOOL OF ENGINEERING

# **Simulated Annealing of Molecular Clusters in Python**

By

Armaan Thapar

A thesis submitted in partial fulfillment of the requirements for the degree of  
Master of Engineering

Advisor

Professor Robert Q. Topper

THE COOPER UNION FOR THE ADVANCEMENT OF SCIENCE AND ART  
ALBERT NERKEN SCHOOL OF ENGINEERING

This thesis was prepared under the direction of the Candidate's Thesis Advisor and has received approval. It was submitted to the Dean of the School of Engineering and the full Faculty, and was approved as partial fulfillment of the requirements for the degree of Master of Engineering.

---

Barry L. Shoop, Ph.D., P.E. - Date  
Dean, Albert Nerken School of Engineering

---

Professor Robert Q Topper  
Candidate's Thesis Advisor

## **Acknowledgements**

First and foremost, I would like to thank Professor Topper for his patient guidance throughout the process of this thesis. I could not have completed this project without his help and the knowledge I have gained through all the courses I have taken with him. I would also like to thank my parents and sister for all their support as I pursued my studies as well as my friends at the Cooper Union for creating a great community and learning environment. Last, but not the least, I would like to thank all the faculty in the Albert Nerken School of Engineering for providing me with a great education and a foundation that will help me in my future endeavors.

## Abstract

This paper describes an object-oriented Python programming project designed to perform simulated annealing Monte Carlo simulations of molecular clusters and thereby find the lowest energy geometry of each cluster. Clusters studied include homogeneous systems consisting of a single species and heterogeneous systems consisting of multiple species. Example studies were performed on water clusters  $[(\text{H}_2\text{O})_n, n = 2, \dots, 5]$  and clusters of ammonium chloride  $[(\text{NH}_4\text{Cl})_n, n = 1, 2 \text{ and } 4]$ . Simulations are performed using mag-walking to ensure ergodic sampling and molecules are treated as rigid bodies. A comparison with previous studies shows that the cluster geometries and conformer energies obtained from the program agree well with literature values. This project will ultimately result in the public release of the first simulated annealing Monte Carlo program which includes mag-walking.

## Table of Contents

Title Page .....	unnumbered
Signature Page .....	unnumbered
Acknowledgements .....	i
Abstract .....	ii
List of Figures .....	iv
Introduction .....	1
Theory .....	6
Modified Metropolis Algorithm .....	14
Potential Energy Calculation .....	17
Python Environment .....	21
Other Programs .....	24
Annealing .....	28
Program Structure .....	31
Initial Configuration .....	33
Simulation Program .....	39
Annealing Program .....	41
Directions For Use of Program .....	42
Model Testing .....	43
Conclusions and Recommendations .....	55
Appendix: Python Files .....	56
References .....	102

## List of Figures

Figure 1: Example Water Clusters of Different Sizes .....	2
Figure 2: Example Heterogeneous System of Ammonium Chloride .....	3
Figure 3: Function with Multiple Minima .....	4
Figure 4: Plot of Points from MC Integration.....	8
Figure 5: Flowchart Outlining the Steps of the Metropolis Algorithm .....	14
Figure 6: Example of an xyz file for carbon dioxide .....	23
Figure 7: Exponential Cooling from 1000K where $\alpha = 0.75$ .....	29
Figure 8: Example of Sawtooth Annealing Schedule with 4 Teeth ( $\alpha = 0.8$ ).....	30
Figure 9: General Workflow of Program.....	31
Figure 10: Initial Water Configurations of 10 Molecules.....	38
Figure 11: Example Possible Configurations for N <sub>2</sub> , SO <sub>2</sub> , NH <sub>3</sub> , and NH <sub>4</sub> Cl Systems .....	38
Figure 12: Cumulative $\langle U \rangle / k_B$ for a simulation with 20000 Trials .....	43
Figure 13: Cumulative C <sub>v</sub> for the Simulation in Figure 12.....	44
Figure 14: Plot of System Size vs. Computing Time.....	45
Figure 15: Plot of Trials vs. Computing Time.....	46
Figure 16: Molecular Geometries (Reprinted from Reference 55).....	47
Figure 17: Accepted Energies for N = 3 System .....	48
Figure 18: Accepted Energies for N = 4 System .....	49
Figure 19: Final Structures for N=3, 4 Water Clusters.....	49
Figure 20: Annealing Schedule for System of 5 TIP3P Molecules .....	50
Figure 21: Graph of Accepted Energies for N=5 System.....	51
Figure 22: Running Average of Accepted Energies of System .....	51
Figure 23: Evolution of Water Clusters for N=5 .....	52

Figure 24: Heterogenous Cluster Results .....	53
Figure 25: Optimal Ammonium Chloride Geometries .....	53
Figure 26: Local Minimum Structure for $(\text{NH}_4\text{Cl})_4$ .....	54

## Introduction

Computational molecular modeling is an important field in physical chemistry and chemical engineering that uses computer simulations of systems at the atomic level to predict properties of the system of interest. This is especially important when experimental data requires additional interpretation or is simply not available. The kinds of systems studied by simulation vary from individual atoms to molecules, clusters, nanoparticles, biomolecules, solutions, materials, and cell membranes. Simulating these systems is often intensive and quantum mechanical simulations often result in very large calculation times as the system size increases. One way to deal with this issue is to instead use classical dynamics to model atomic motions, informed by the Correspondence Principle as to where this approximation can be used appropriately. This is combined with the use of empirical models for intermolecular forces which are adjusted to match experimental results. The use of classical mechanics implies solving Newton's equations of motion and periodically readjusting the velocities so that thermal equilibrium can be reached, which is referred to as "molecular dynamics." In this work we emphasize an alternative approach in which Monte Carlo simulations are used. These simulations use the Metropolis-Hastings algorithm, in which a system of particles or molecules are simulated by sampling over a large range of possible configurations and averaged in terms of probability to determine a certain property.<sup>6</sup> This method can be used either for the purpose of computing thermodynamic properties or for locating the lowest energy configurations of the system, which are the most probable ones for experimental observation.

In the present study molecular clusters of two different types were studied. Homogeneous clusters consisting of a single type of species and heterogeneous systems containing two



different types of species were studied. The aim was to find the lowest energy configuration of each cluster. Figures 1 and 2 show two randomly selected starting positions of systems that can be analyzed by the Monte Carlo program developed and presented in this thesis. Molecules in these systems are interacting with one another via potential forces, which include Coulombic and non-Coulombic interatomic forces. In this work, all molecules are treated as rigid bodies, which can move translationally and can rotate freely, and the approach is limited to systems that do not exhibit dynamic proton or electron transfer between molecules. All such transfers are assumed to have happened before the simulation begins.

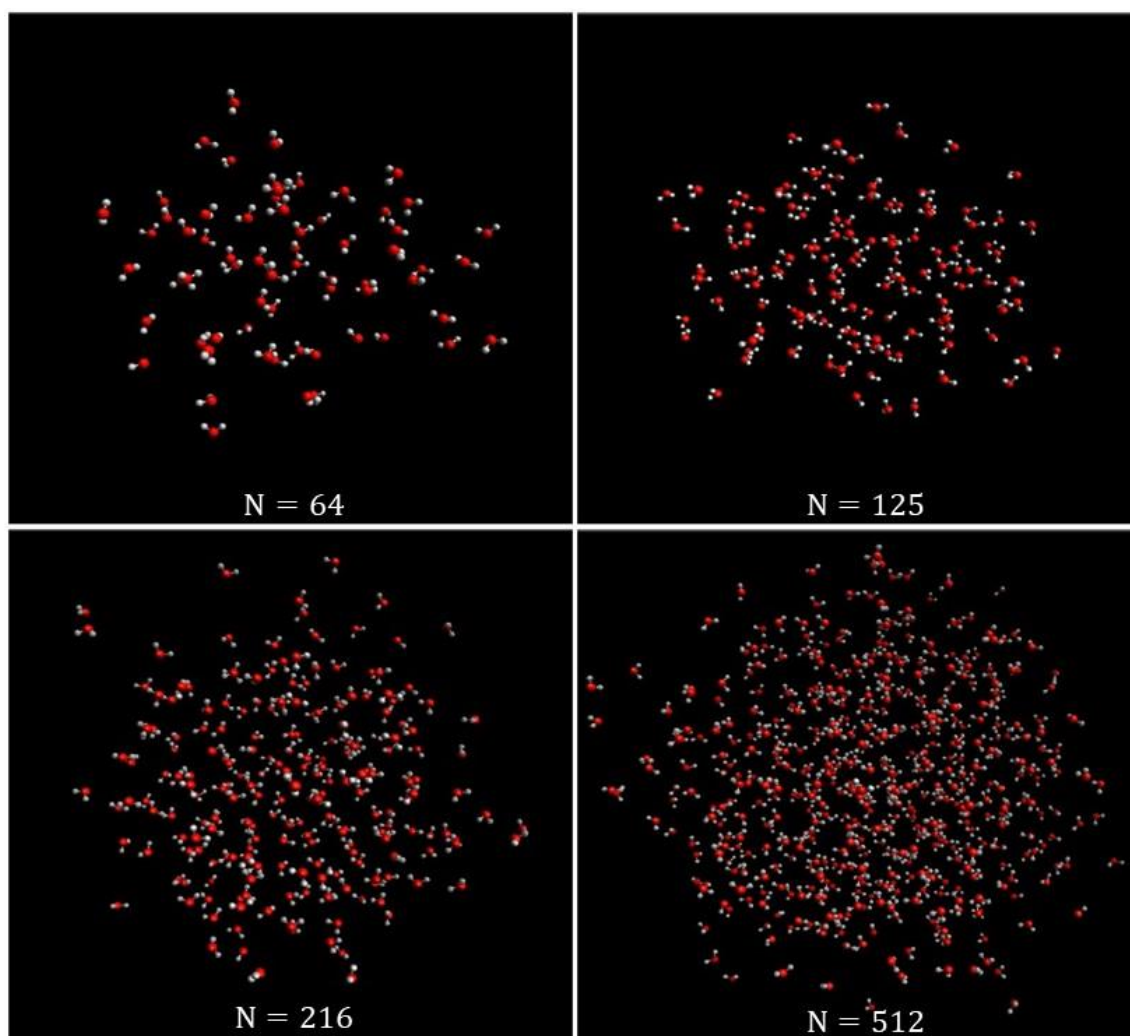


Figure 1: Example Water Clusters of Different Sizes

An important aspect of such simulations is that many states are less probable and thus less likely to contribute to the average of a particular property of the system. In molecular systems, the probability is directly tied to the energy of the state. Lower energy states are more probable than higher energy states. Thus, these configurations are more important to determining properties such as the energy of the system itself as well as the heat capacity and free energy. In order to find more probable states, Monte Carlo methods must be able to sample the possible states of a system ergodically.<sup>10</sup> Ergodic sampling implies that given enough time, a simulation could sample all points of the system. This is very important for systems with multiple local minima. Once a system reaches a potential well of lower energy, it may need to accept some increases in its energy to move and reach another minimum. This is also an issue when performing a search for the lowest energy structures, as this search cannot be performed without efficiently sampling all possible configurations. A one-dimensional example is given in Figure 3. For the system to move from a local minimum to another minimum, the system would have to first increase in energy.

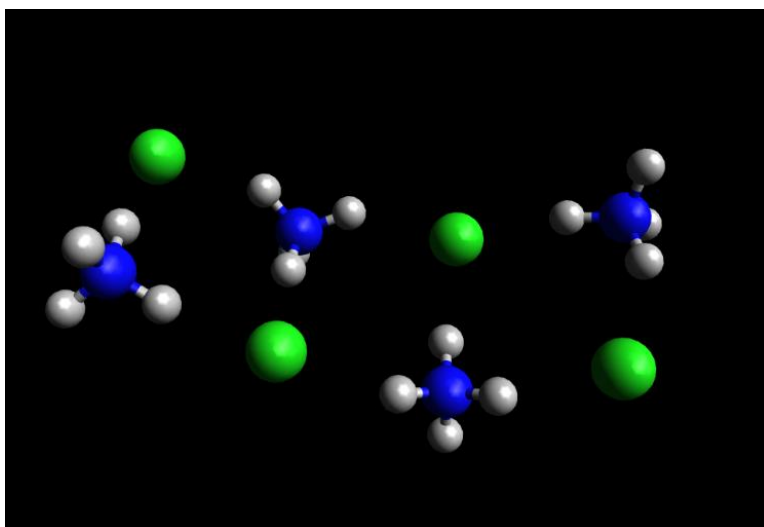


Figure 2: Example Heterogeneous System of Ammonium Chloride (with blue Nitrogen atoms and green Chloride ions)

Monte Carlo methods account for the multiple minima problem by allowing some higher energy moves. In molecular simulations, the Boltzmann factor is used as the probability of accepting a higher energy move.

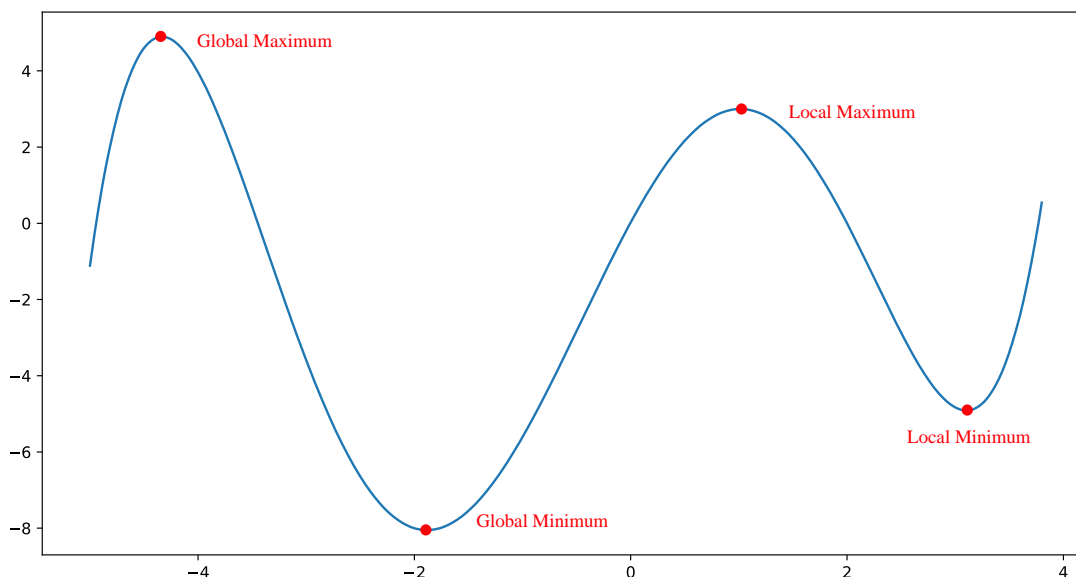


Figure 3: Function with Multiple Minima

Though this measure helps sample higher energy states, this may still prevent a system from breaking out of a local minimum during a simulation. Since the number of trial moves is finite and chosen by the user, sampling may end up not being ergodic as the system is not able to sample all the states. This is especially true if the local maximum or hill between two minima is very large. Two ways to account for this are used in the algorithm for this paper. Both were used in previous work by Topper et al. for the simulation of ammonium chloride clusters.<sup>1</sup>

The first method is called “mag-walking.”<sup>1</sup> In this method, the size of steps is occasionally magnified during the simulation to allow the system to possibly leap to another stable state that may have otherwise required several uphill moves to reach. This helps with ergodic sampling as a mag-step may be able to move between two minima with a high energy well between them without needing to accept too many high energy moves. Another feature of

the program presented here that helps to find low-energy structures is the use of “simulated annealing.”<sup>2</sup> In this method the system is first simulated at a very high temperature and slowly cooled down to absolute zero. The initial elevated temperatures allow for the system to sample nearly all configurations and the slow drop in temperature causes the system to reach a low energy configuration.<sup>3</sup>

## Theory

Monte Carlo simulations are rooted in the idea of using statistical sampling rather than direct calculation to solve a problem more efficiently. An early attempt was made by Enrico Fermi to simulate neutron diffusion on the FERMIAC but he did not publish any of these studies.<sup>4</sup> The Markov Chain Monte Carlo method was developed by von Neumann and Ulam during their work at Los Alamos Laboratory, and received the name “Monte Carlo” because of its use of random numbers in calculations.<sup>5,6</sup> The method uses the generation of pseudorandom numbers from a computer to perform stochastic sampling. This method can be used to estimate definite integrals of the form:

$$F = \int_{x1}^{x2} f(x)dx \quad (1)$$

Such an integral can be rewritten as,

$$F = \int_{x1}^{x2} \left( \frac{f(x)}{\rho(x)} \right) \rho(x)dx \quad (2)$$

Approximating the integral by a sum leads to a simple average over trials:

$$F \approx \frac{\sum_{\tau=1}^{\tau_m} \frac{f(\xi_{\tau})}{\rho(\xi_{\tau})}}{\tau_m} \quad (3)$$

Here,  $\tau_m$  is the total number of trials and  $\xi_{\tau}$  is a random sample from  $\rho$ , where the subscript  $\tau$  signifies that a new sample is taken for each trial. If the probability distribution is uniform,

$$\rho(x) = x2 - x1 \quad (4)$$

Then,

$$F \approx \frac{x_2 - x_1}{\tau_m} \sum_{\tau=1}^{\tau_m} f(\xi_\tau) \quad (5)$$

### Preliminary Example: MC Integration of a Circle

Equation 5 yields a result that can be used on a computer to solve simple problems to a reasonable accuracy. A simple example using random points for integration was performed using 10,000 points to estimate  $\pi$ . In the example a circle of radius  $r = 1$  and diameter  $D = 2$  is used so that the area is  $\pi$ :

$$F = \int_0^2 f(x) dx = A_{circle} \quad (6)$$

$$A_{circle} = \pi r^2 = \pi \quad (7)$$

Here,  $y$  or  $f(x)$  is simply the equation of a circle given by:

$$x^2 + y^2 = 1 \quad (8)$$

$$f(x) = \sqrt{r^2 - x^2} \text{ OR } -\sqrt{r^2 - x^2} \quad (9)$$

The study can be performed by randomly generating  $\tau_m = 10,000$  points from a uniform distribution on a square grid between 0 and 2, which serve as the  $\xi_\tau$ . The coordinates of the generated point can then be tested to see if it matches the circle equation. As the probability of choosing each point is uniform along both axes, the number of points falling in any given area is also uniform. That is,

$$\frac{A_{circle}}{n_{circle}} = \frac{A_{square}}{n_{square}} \quad (10)$$

Here,  $n_{circle}$  is the number of point within the circle. Since  $r = 1$  for a circle of diameter of 2, this equation can be simplified to:

$$A_{circle} = \frac{n_{circle}}{n_{square}} \cdot A_{square} = 4 \frac{n_{circle}}{n_{square}} \quad (11)$$

Solving the problem gave an approximation of  $\pi$  as 3.1472. Using a higher number of points leads to increased accuracy but the convergence rate is known to be rather slow with this method. A plot of the points from the example study is given in Figure 4.

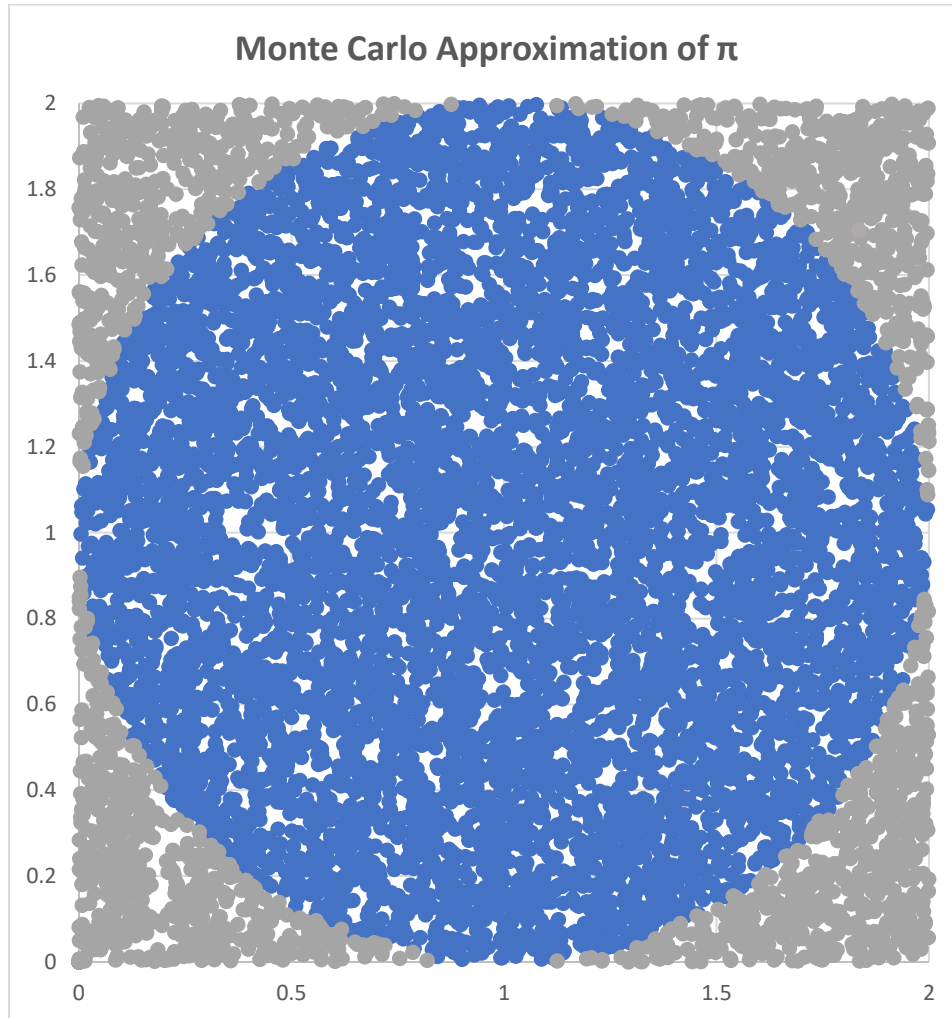


Figure 4: Plot of Points from MC Integration

A filtering process used in Microsoft Excel moved all points outside the circle to the origin, which are shown as a single blue point in the plot.

### Relation to Statistical Mechanics

Monte Carlo simulation in many ways is the statistical mechanics counterpart to molecular dynamics. This is because the principles used in Monte Carlo simulations to estimate properties are very similar to those of statistical mechanics.<sup>7</sup> Gibbs proposed the idea of ensembles, which are a very large collection of systems, each a replica of the system of interest.<sup>8</sup> Assuming the principle of equal a priori probabilities, the average over the entire ensemble will yield the average property for the system. In the canonical ensemble, which accounts for isothermal systems, with a constant number of particles, and a constant volume, the ensemble average is given by:

$$\bar{E} = \frac{\sum_j E_j \exp(-\beta E_j)}{\sum_j \exp(-\beta E_j)} \quad (12)$$

Since the canonical ensemble follows the Boltzmann distribution, we can rewrite this as:<sup>8</sup>

$$\bar{E} = \frac{\int U(\mathbf{x}) \exp(-\beta U(\mathbf{x})) d\mathbf{x}}{\int \exp(-\beta U(\mathbf{x})) d\mathbf{x}} \quad (13)$$

This can be generalized for other properties such that:

$$\langle f \rangle = \frac{\int f(\mathbf{x}) \exp(-\beta U(\mathbf{x})) d\mathbf{x}}{\int \exp(-\beta U(\mathbf{x})) d\mathbf{x}} \quad (14)$$

where  $f(\mathbf{x})$  is some property of interest, and  $U(\mathbf{x})$  is the potential energy of the system. Finding the average of property  $f$  will be a difficult task because  $\mathbf{x}$  will have  $3N$  coordinates, where  $N$  is the number of atoms, over which to integrate. One method, which has become relevant with the



advent of computers is to sample different states of the system to estimate the integrals. The most basic method of sampling is randomly sampling so that states are produced with uniform probability. The average of a property  $f$  is given by:

$$\langle f \rangle = \frac{\sum_{i=1}^k f(\mathbf{x}_i) \cdot \exp(-\beta U(\mathbf{x}_i))}{\sum_{i=1}^k \exp(-\beta U(\mathbf{x}_i))} \quad (15)$$

Here, the upper limit on the sum is the total number of trials, or the number of times the system is sampled. Using a uniform distribution, however, can be inefficient as improbable states are just as likely to be picked as probable states, and the method will require many samples to estimate the value of the property. A move towards a high energy state results in very low Boltzmann factors which does not affect the value of  $\langle f \rangle$ , as both the numerator and denominator are summations.

$$\exp(-\beta U(\mathbf{x}_i)) \approx 0 \text{ when } U(\mathbf{x}) \text{ is large} \quad (16)$$

A method that samples points with a higher probability from more probable points is called importance sampling. This is important in the context of molecular simulations, as lower energy states are more probable than higher energy states. Thus, the state of a system is more likely to be in one of its lower energy states, and a method of sampling that reflects this would converge faster than one that does not. One method which was developed to sample from the probability distribution for the canonical ensemble,  $\rho_{NVT}$ , is the Metropolis algorithm, which was developed by a team of researchers led by Nicholas Metropolis at Los Alamos consisting of Arianna Rosenbluth, Marshall Rosenbluth, Augusta Teller and Edward Teller.<sup>6</sup>

In order to solve the problem of sampling from  $\rho_{NVT}$ , the Metropolis Monte Carlo algorithm constructs a Markov chain of states that converges to  $\rho_{NVT}$ . In a Markov chain, the next state of a system is entirely dependent on the previous state. There is also a transition

probability for every possible state that the system can go to that make up the state space.

Metropolis et al. in their study constructed a method to approximate the probability distribution by sampling. The probability from moving from  $\mathbf{x}$  to  $\mathbf{x}'$  is given by  $P(\mathbf{x} \rightarrow \mathbf{x}')$ :<sup>10</sup>

$$P(\mathbf{x} \rightarrow \mathbf{x}') = K(\mathbf{x} \rightarrow \mathbf{x}') \rho(\mathbf{x}) \quad (17)$$

This probability is a product of the system being in state  $\mathbf{x}$ , and  $K(\mathbf{x} \rightarrow \mathbf{x}')$ , the conditional probability that state  $\mathbf{x}$  will move to  $\mathbf{x}'$  in a move. For systems to evolve towards a unique limiting distribution, the condition of microscopic reversibility on average is held, such that the system is as likely to move from  $\mathbf{x}$  to  $\mathbf{x}'$  as it is  $\mathbf{x}'$  to  $\mathbf{x}$ .<sup>10</sup> This is given by:

$$P(\mathbf{x} \rightarrow \mathbf{x}') = P(\mathbf{x}' \rightarrow \mathbf{x}) \quad (18)$$

and,

$$K(\mathbf{x} \rightarrow \mathbf{x}') \rho(\mathbf{x}) = K(\mathbf{x}' \rightarrow \mathbf{x}) \rho(\mathbf{x}') \quad (19)$$

The conditional probability of move to  $\mathbf{x}'$  occurring is a product of the probability that the state  $\mathbf{x}'$  will be generated by the random trial ( $\Pi$ ) and the probability that this trial move is accepted ( $A$ ).

$$K(\mathbf{x} \rightarrow \mathbf{x}') = \Pi(\mathbf{x} \rightarrow \mathbf{x}') A(\mathbf{x} \rightarrow \mathbf{x}') \quad (20)$$

The choices for  $\Pi$  are quite flexible in the Monte Carlo algorithm and many implementations exist including the original Metropolis method and the mag-walking method used in this work.

The acceptance probability is chosen so that the condition of detailed balance is satisfied.

Rewriting equation 19, we see:

$$\Pi(\mathbf{x} \rightarrow \mathbf{x}') A(\mathbf{x} \rightarrow \mathbf{x}') \rho(\mathbf{x}) = \Pi(\mathbf{x}' \rightarrow \mathbf{x}) A(\mathbf{x}' \rightarrow \mathbf{x}) \rho(\mathbf{x}') \quad (21)$$

and,

$$\frac{A(\mathbf{x} \rightarrow \mathbf{x}')}{A(\mathbf{x}' \rightarrow \mathbf{x})} = \frac{\Pi(\mathbf{x}' \rightarrow \mathbf{x}) \rho(\mathbf{x}')}{\Pi(\mathbf{x} \rightarrow \mathbf{x}') \rho(\mathbf{x})} \quad (22)$$

The trial probabilities in the Metropolis method and in this work are uniform so that every path from  $\mathbf{q}$  to  $\mathbf{q}'$  can be travelled in both directions, which results in the simplification:

$$\frac{A(\mathbf{x} \rightarrow \mathbf{x}')}{A(\mathbf{x}' \rightarrow \mathbf{x})} = \frac{\rho(\mathbf{x}')}{\rho(\mathbf{x})} \quad (23)$$

A rejection method can be determined to define A:

$$A(\mathbf{x} \rightarrow \mathbf{x}') = \begin{cases} 1 & \text{if } \rho(\mathbf{x}') \geq \rho(\mathbf{x}) \\ \frac{\rho(\mathbf{x}')}{\rho(\mathbf{x})} & \text{if } \rho(\mathbf{x}) \geq \rho(\mathbf{x}') \end{cases} \quad (24)$$

If  $\rho(\mathbf{q}') \geq \rho(\mathbf{q})$  and thus  $\mathbf{q}'$  is the more probable state, the acceptance probability will be 1.

Otherwise, the probability of accepting the move will be less than 1. If we let:

$$\alpha(\mathbf{x} \rightarrow \mathbf{x}') = \Pi(\mathbf{x} \rightarrow \mathbf{x}') \rho(\mathbf{x}) \quad (25)$$

Then the Metropolis solution for the system is then given by:<sup>5</sup>

$$P(\mathbf{x} \rightarrow \mathbf{x}') = \begin{cases} \alpha(\mathbf{x} \rightarrow \mathbf{x}') & \text{if } \rho(\mathbf{x}') \geq \rho(\mathbf{x}) \\ \alpha(\mathbf{x} \rightarrow \mathbf{x}') \cdot \frac{\rho(\mathbf{x}')}{\rho(\mathbf{x})} & \text{if } \rho(\mathbf{x}) \geq \rho(\mathbf{x}') \end{cases} \quad (26)$$

This asymmetrical solution where the probability of transitioning is lower when the new state is less probable is an important aspect of this Markov chain method. The method was generalized for use with arbitrary distributions by Hastings in 1970, and as a result the method is now widely referred to as the “Metropolis-Hastings algorithm.”<sup>9</sup> For systems in the Canonical ensemble,  $\rho(\mathbf{x})$  is given by:<sup>10</sup>

$$\rho_{NVT}(\mathbf{x}) = \frac{\exp[-\beta U(\mathbf{x})] J(\mathbf{x})}{Z_{NVT}} \quad (27)$$

where,

$$Z_{NVT} = \int d\mathbf{x} \exp[-\beta U(\mathbf{x})] J(\mathbf{x}) \quad (28)$$

where  $J(\mathbf{x})$  is the determinant of the Jacobian needed for statistical mechanics calculations when non-Cartesian coordinates are used. Using this equation for  $\rho(\mathbf{x})$ ,

$$\frac{A(\mathbf{x} \rightarrow \mathbf{x}')}{A(\mathbf{x}' \rightarrow \mathbf{x})} = \frac{\Pi(\mathbf{x}' \rightarrow \mathbf{x})}{\Pi(\mathbf{x} \rightarrow \mathbf{x}')} \exp[-\beta \Delta U] \frac{J(\mathbf{x}')}{J(\mathbf{x})} \quad (29)$$

where  $\Delta U = U(\mathbf{x}') - U(\mathbf{x})$  and  $\exp[-\beta \Delta U]$  is the Boltzmann factor. As Cartesian coordinates are used in this work, the Jacobian terms will both be 1. The acceptance probabilities will then change to:

$$A(\mathbf{x} \rightarrow \mathbf{x}') = \begin{cases} 1 & \text{if } \rho(\mathbf{x}') \geq \rho(\mathbf{x}) \\ \exp(-\beta \Delta U) & \text{if } \rho(\mathbf{x}) \geq \rho(\mathbf{x}') \end{cases} \quad (30)$$

and,

$$P(\mathbf{x} \rightarrow \mathbf{x}') = \begin{cases} \alpha(\mathbf{x} \rightarrow \mathbf{x}') & \text{if } \rho(\mathbf{x}') \geq \rho(\mathbf{x}) \\ \alpha(\mathbf{x} \rightarrow \mathbf{x}') \cdot \exp(-\beta \Delta U) & \text{if } \rho(\mathbf{x}) \geq \rho(\mathbf{x}') \end{cases} \quad (31)$$

This final equation shows how importance sampling is applied in this work. The probability of a state being generated will depend on  $\alpha$ , yet the probability of a move being accepted by the algorithm will depend on the condition of how probable that state is. If  $\mathbf{x}'$  is lower in energy, the probability of moving to it from  $\mathbf{x}$  will only depend on  $\alpha$ . If the state  $\mathbf{x}'$  is higher in energy and thus less probable, the probability of accepting the move will be a product of  $\alpha$  and  $\exp(-\beta \Delta U)$ , which will lead to decrease as  $\Delta U$  is always positive for uphill moves. Therefore, importance sampling is applied by suppressing the probability of higher energy moves, and the factor by which acceptance is suppressed depends on the magnitude of  $\Delta U$ .

## Modified Metropolis Algorithm

The original Metropolis algorithm is altered in this paper to allow for “mag-walking.” A useful flowchart was obtained from a paper by Topper et al. and outlines the steps for sampling the canonical ensemble.<sup>10</sup>

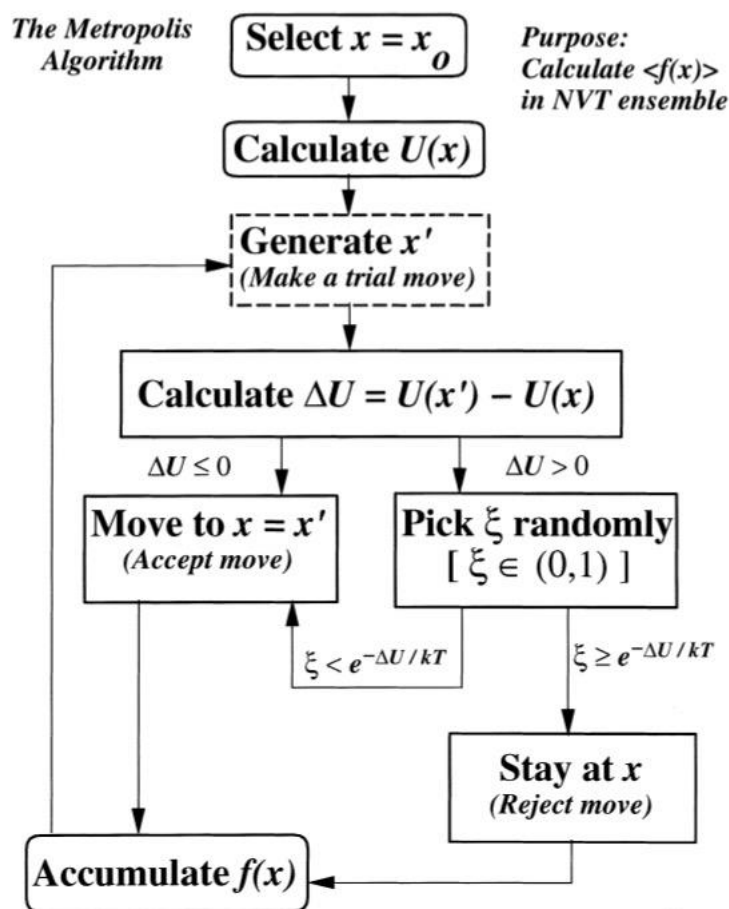


Figure 5: Flowchart Outlining the Steps of the Metropolis Algorithm (reprinted with permission from Reference 10)

The general flow of the program is to start with some initial state  $x_0$  and find the initial potential energy of the system. A trial move is then generated which could increase or decrease the energy. For the molecular systems studied in the program, trial moves would have translations

and rotations of molecules. Translational steps are chosen randomly between  $-\frac{L}{2}$  to  $\frac{L}{2}$ , where L is the size of the box within which the molecule can move. Rotation sampling based on the work of Barker and Watts<sup>11</sup> is used to rotate any rigid body along the origin. Rotations along the three axes are given in equations 32, 33, and 34.

$$\text{X rotation:} \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (32)$$

$$\text{Y rotation} \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (33)$$

$$\text{Z rotation} \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (34)$$

Since these equations are for rotations along the origin, coordinates need to be adjusted for molecules as they move away from the origin. This is accounted for in the program by performing rotations around the center of mass of each molecule.

The method of calculating the potential energy is discussed in the next section. Once the difference in energy between the two states is calculated, the algorithm does one of two things: accept the move if  $\Delta U \leq 0$  (non-positive moves are always accepted) or perform a test to determine whether to accept an uphill move. If the move is accepted, the trial move becomes the new state, and the algorithm is repeated. The latter condition for uphill moves is important towards sampling ergodically. The test performed in the program is to randomly generate a number between 0 and 1,  $\xi$ , and compare it to the Boltzmann factor.<sup>10</sup> The Boltzmann factor will always be less than 1 but greater than 0, for positive  $\Delta U$ .

$$\text{Boltzmann factor} = \exp\left(-\frac{\Delta U}{kT}\right) \quad (35)$$

The probability that an uphill move is accepted is higher if  $\Delta U$  is not large. If the move is accepted, it is saved like a downhill move and the trial state becomes the new state of the system. If the move is rejected, the system stays in the configuration prior to the trial.

The first major modification to this algorithm in this paper is during the generation of new trial moves. Prior to randomly perturbing a molecule, two random numbers are generated. The numbers are compared to the mag-step probabilities. In the program,  $P_{MT}$  and  $P_{RT}$  are the probabilities of accepting a translational and rotational magstep, respectively. If the translational mag-step condition is met, the box size will increase from  $L$  to  $\beta \cdot L$ . Similarly, if the rotational magstep condition is met, the molecule can be rotated by a larger angle. The translational and rotational mag-step conditions are tested separately, so that one, both, and neither condition could be accepted during a trial move. Once the move is generated, the algorithm behaves much like the original algorithm with the Boltzmann factor determining if an uphill move is accepted.

## Potential Energy Calculation

There are many forms of potential energy functions available in the literature including AMBER<sup>12</sup>, CHARMM, and OPLS.<sup>13</sup> These potential energy functions not only describe “non-bonding” interactions between molecules, but also “bonding” interactions, which describe molecular vibrations. Therefore, the total potential energy is,

$$U_{Total} = U_{bonding} + U_{non-bonding} \quad (36)$$

In this work, we will only consider non-bonding interactions. An example function is the modified version of the AMBER forcefield created Kollman et al. is given below:<sup>14</sup>

$$U_{Total} = \sum_{Bonds} K_r (r - r_{eq})^2 + \sum_{angles} K_\theta (\theta - \theta_{eq})^2 + \sum_{dihedrals} \frac{V_n}{2} [1 + \cos(n\phi - \gamma)] + \sum_{i < j} \left[ \frac{A_{ij}}{R_{ij}^{12}} - \frac{B_{ij}}{R_{ij}^6} + \frac{q_i q_j}{\epsilon R_{ij}} \right] \quad (37)$$

The last expression is the non-bonding contribution and is a simplified version of the Lennard-Jones potential and a Coulombic term. The terms  $A_{ij}$  and  $B_{ij}$  are given by:

$$A_{ij} = 4\epsilon \cdot \sigma^{12} \quad (38)$$

$$B_{ij} = 4\epsilon \cdot \sigma^6 \quad (39)$$

In more detail: the first term represents interactions between bonded atoms and the second term is the change in energy due to a change in the angle of atoms. The dihedral angle represents changes in energy due to rotations about a bond. Molecular dynamics potentials can be useful for Monte Carlo simulations, but the additional terms for bonding when the molecule is treated as a rigid body (as it is in this case) can be dropped. In this simulation, bond lengths, angles, and dihedral angles will remain constant. In the above expression, only the last term is relevant to the present study, but the values used for  $A_{ij}$  and  $B_{ij}$  will result in a different final result. Since



TIP3P is being studied as a homogenous system, partial charges and bond distances are needed.

These were found in the primary literature and are given in Table 1.<sup>15</sup>

Property	Value	Units
r(OH)	0.9572	Å
HOH Angle	104.52	deg
A	582.0	10 <sup>3</sup> kcal Å <sup>12</sup> /mol
B	595.0	kcal Å <sup>6</sup> /mol
q(O)	-0.834	e
q(H)	+0.417	e

Table 1: TIP3P Parameters

The potential used in the present study contains 3 terms. The first is the Lennard Jones 6-12 potential given in equation 40. This form of the equation is for two atoms separated by distance  $r$ .<sup>5</sup>

$$U_{LJ} = 4\varepsilon \left[ \left( \frac{\sigma}{r} \right)^{12} - \left( \frac{\sigma}{r} \right)^6 \right] \quad (40)$$

Here,  $\varepsilon$  and  $\sigma$  are parameters that depend on the molecule in question. The authors also assumed classical statistics as opposed to quantum statistics on the particle system to determine material properties.

For polar molecules and ions, the Coulombic force is important to account for. For two molecules composed of  $m$  and  $n$  atoms, respectively, the total Coulombic force will be the sum of the individual forces between the atoms of the two molecules:

$$U_{Coulomb} = \sum_{a=1}^m \sum_{b=1}^n \frac{k_c q_a q_b}{r_{ab}} \quad (41)$$

The third term in the potential is a constraint needed for simulations at higher temperatures. As the molecules being studied are in cluster form, it is possible for molecules to evaporate away at

higher temperatures (>100K), particularly for smaller clusters. To adjust for this, an additional term is added to the total potential in simulations:

$$U_{Constraint} = \left(\frac{r}{C}\right)^{2n} \text{ or } U_{Constraint} = \left(\frac{r-R_{COM}}{C}\right)^{2n}, n = 1, 2, 3... \quad (42)$$

Here,  $r$  is the distance from the origin of the molecule,  $R_{COM}$  is the distance from the center of mass, and  $C$  is a constant that will control when the constraint potential punishes moves. Higher order terms than 2 can be used for a stronger effect but was found not be needed due to the polar nature of the molecules studied. As molecules get further away from the center of the cluster, the constraint term increases making these moves less likely to be accepted.

The individual potentials can then be summed over all molecules to give the total energy from each source. The total potential from Lennard Jones interactions where a single atom is used for calculations is:

$$U_{LJ} = \sum_i \sum_{j>i} \left( 4\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right) \quad (43)$$

The values for parameters  $\sigma$  and  $\epsilon$  can be found in literature for pure species, and interatomic interactions can be estimated using the Lorentz-Bertholot rules or other mixing rules.<sup>16,17</sup> For systems where calculations need to be performed atom by atom:

$$U_{LJ} = \sum_i \sum_{j>i} \left( \sum_a \sum_b \left[ 4\epsilon_{eq} \left( \left( \frac{\sigma_{eq}}{\|r_i^a - r_j^b\|} \right)^{12} - \left( \frac{\sigma_{eq}}{\|r_i^a - r_j^b\|} \right)^6 \right) \right] \right) \quad (44)$$

Where,  $\sigma_{eq}$  and  $\epsilon_{eq}$  are calculated using mixing rules based on the two atoms.

$$U_{eq} = \left( \frac{\sigma_i^a + \sigma_j^b}{2} \right) \quad (45)$$

$$U_{eq} = \sqrt{\epsilon_i^a \cdot \epsilon_j^b} \quad (46)$$

The total potential energy from Coulombic interactions is given by:

$$U_{Coulombic} = \sum_i \sum_{j>i} \left( \sum_a \sum_b \left[ \frac{k_c q_i^a \cdot q_j^b}{\|r_i^a - r_j^b\|} \right] \right) \quad (47)$$

Lastly, the contribution from the constraint potential is given by:

$$U_{Constraint} = \sum_i \left( \frac{r_i}{C} \right)^{2n}, \quad n = 2, 4, 6, \dots \quad (48)$$

The total energy of the system at any state is then given by the sum:

$$V_{Total} = V_{LJ} + V_{Coulomb} + V_{Constraint} \quad (49)$$

The explicit form for the total energy is given by:

$$U_{Total} = \sum_i \sum_{j>i} \left( 4\epsilon \left[ \left( \frac{\sigma}{r_{ij}} \right)^{12} - \left( \frac{\sigma}{r_{ij}} \right)^6 \right] \right) + \sum_i \sum_{j>i} \left( \sum_a \sum_b \left[ \frac{k_c q_i^a \cdot q_j^b}{\|r_i^a - r_j^b\|} \right] \right) + \sum_i \left( \frac{r_i}{C} \right)^{2n} \quad (50)$$

or when calculating atomic interactions for Lennard Jones:

$$U_{Total} = \sum_i \sum_{j>i} \left( \sum_a \sum_b \left[ 4\epsilon_{eq} \left( \left( \frac{\sigma_{eq}}{\|r_i^a - r_j^b\|} \right)^{12} - \left( \frac{\sigma_{eq}}{\|r_i^a - r_j^b\|} \right)^6 \right) \right] \right) + \sum_i \sum_{j>i} \left( \sum_a \sum_b \left[ \frac{k_c q_i^a \cdot q_j^b}{\|r_i^a - r_j^b\|} \right] \right) + \sum_i \left( \frac{r_i}{C} \right)^{2n} \quad (51)$$

For mixed systems, a modified OPLS potential similar in structure to the AMBER potential is used with parameters from Topper et al. is currently used, which calculates the total potential atom by atom.<sup>18</sup> The values used for the parameters  $A, \alpha, C$  and  $D$  can be found in the original work.

$$U_{Total} = \sum_i \sum_{j>i} \left( A_{ij} \exp(-\alpha_{ij} r_{ij}) + \frac{q_i q_j}{r_{ij}} + \frac{D_{ij}}{r_{ij}^{12}} - \frac{C_{ij}}{r_{ij}^6} \right) + \sum_i \left( \frac{r_i}{C} \right)^{20} \quad (52)$$

The total energy will include for the system will include the kinetic energy for an ideal gas:

$$E = \frac{3}{2} N k_B T + U_{Total} \quad (53)$$

## Python Environment

Simulations were performed using **Python 3.6**<sup>19</sup>. More advanced libraries were avoided in the creation of the program to avoid the need to make future corrections for deprecated functions. Using simpler functions and basic syntax makes it less likely that future users will need to adjust the script. Several Python libraries were very important in creating this program. These are:

**Numpy**<sup>20</sup>: This library allows the user to create matrices and vectors, which are termed numpy arrays or ndarray within Python. The numpy can then use functions within the numpy library to perform other actions such as multiplying the matrices, finding norms, transposing, and reshaping the matrix. Numpy also has many operations for scalars including the generation of random numbers both uniformly (can be used to generate a random number between 0 and 1) and according to a particular distribution, such as a beta distribution.

**Matplotlib**<sup>21</sup>: This library allows a user to create graphs readily and works with the numpy library. The general method of use in this program is the simulation first creates a numpy array after performing the algorithm on the system. Arrays of the general properties can then be plotted and visualized using matplotlib. This library and in particular the pyplot collection of functions behave quite like MATLAB.<sup>22</sup>

**Pandas**<sup>23</sup> This library allows the user to organize data that would be in arrays more conveniently. Pandas allows the user to create data frames in which data is organized like cells in

a spreadsheet as compared to a term in a matrix, allowing for faster and more convenient analysis.

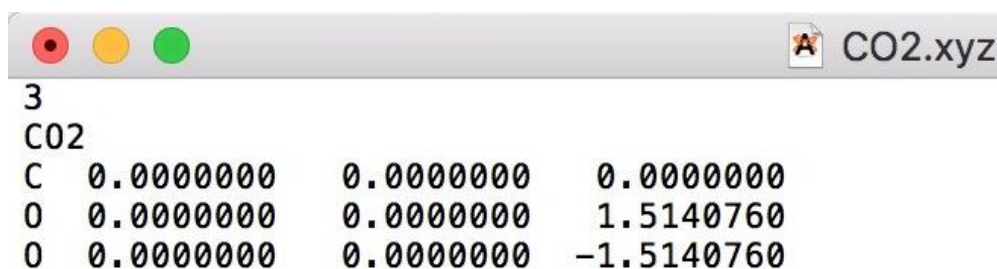
These three libraries are the most important in creating most of the Monte Carlo program, and need to be installed separately. There also some libraries that are preinstalled that are useful to know when dealing with common performance issues in simulations and storing data.

**Math:** Though one can perform simple arithmetic on python relatively easily without importing libraries, one does need to import the math library to use certain constants like pi, or for functions like trigonometric functions. Many of these functions and constants can be imported through numpy as well.

**Copy:** When a variable is assigned, Python does not create a separate space in memory. This can be problematic as change of one variable may alter another variable assigned the same place in memory by the program. This flaw of shallow copying is troublesome during simulations where there may already be many factors that could change the value of a variable like energy, making this a hard problem to figure out and better simply avoided. The `deepcopy()` function allows the user to store a copy in a separate place in the memory.

The Monte Carlo program performs most calculations on position matrices, and during the simulation, it can be difficult to visualize how molecules are moving. For this purpose, molecule editor and visualizing software can be very useful. During the simulations, molecule positions can be saved, providing snapshots of the evolution of the system. In this program, the

method used to visualize the system was to store atomic positions to an xyz file<sup>24</sup>, and use the program Avogadro<sup>25</sup> to visualize the files. An example of an xyz file is given in Figure 6. The first line of the program is the number of atoms, the next line is a comment, and from then on atomic symbols and cartesian coordinates of the atoms in the system are printed. The simple and convenient format of the file made it useful for visualizing. There are also other chemical file formats that are widely used for research including protein data bank or pdb files as well as the CHARMM file format.



```
3
CO2
C  0.0000000  0.0000000  0.0000000
O  0.0000000  0.0000000  1.5140760
O  0.0000000  0.0000000 -1.5140760
```

Figure 6: Example of an xyz file for carbon dioxide

## Other Programs

There are several programs already in the literature that are designed for the use of Monte Carlo calculations on atomic level systems such as BOSS, as well as programs that include Monte Carlo simulations as a feature. Others such as PSI4 can be used alongside the program in this work to further optimize a geometry.<sup>26</sup> Most programs that are commonly used are written in C, C++, or Fortran, as compiled programs yield the fastest run times, which is important when dealing with large numbers of iterations.<sup>27</sup> Many do have APIs in Python which call on these source scripts in the compiled, as for example with Cassandra. Increases in computer power over time have made it possible to use Python, which users may be more familiar with, to perform annealing calculations in a reasonable time frame and help get an improved understanding before reading source code in these other languages with differing features.

**BOSS**<sup>28,29,30</sup>: Developed by William L Jorgensen at Yale, BOSS is designed as a general-purpose Monte Carlo simulation program for simulating biochemical and organic systems. The program is written in Fortran and can perform Monte Carlo simulations consisting of molecules in a box with up to 25 solute particles with periodic boundary conditions, in solvent clusters, or dielectric medium. The program also offers energy minimization searches, and conformational searches using Quantum Mechanical methods given a starting structure. The program generally uses an isothermal-isobaric NPT ensemble for MC calculations, though the canonical NVT ensemble used in this work can also be employed. The program allows the user to perform annealing calculations under the OPTIMIZER file parameter, with additional parameters to control the annealing schedule and initial temperature.

**SPARTAN**<sup>31, 32</sup>: SPARTAN is software that is useful for molecular modeling as well as other computational chemistry applications. Like other programs such as CP2K<sup>33</sup> created for quantum mechanical calculations, SPARTAN has many options available for calculations in molecular mechanics, ab initio methods, DFT, Hartree Fock and post-Hartree Fock methods. SPARTAN can be used to create initial structures before simulations as well as to perform MC calculations within the program. SPARTAN uses dihedral space Monte-Carlo methods in order to find the lowest energy conformers for a given molecule.

**Cassandra**<sup>34,35</sup>: Cassandra is open source and was developed by the Maginn group at the University of Notre Dame in order to perform Monte Carlo simulations of molecules on an atomistic level and is mostly written in Fortran, though some user features have been added in python that call on the Fortran scripts. The program allows simulation in several ensembles including the Canonical (NVT) ensemble, Isothermal-Isobaric ensemble (NPT), Grand Canonical ensemble ( $\mu$ VT), constant volume Gibbs ensemble, and constant pressure Gibbs ensemble. Cassandra is designed to simulate bulk systems rather than clusters, with periodic boundary conditions and Ewald sums instead of a constraining potential as employed in the present work.

**TINKER**<sup>36</sup>: An open source molecular modeling package written in Fortran and developed by the Jay Ponder lab in Washington University, Saint Louis. The program is generally designed for molecular mechanics and dynamics calculations, though the program offers the option for Monte Carlo simulations as well as simulated annealing. Many potential functions are available including Amber ff94, CHARMM19, Dang, MM2, MM3, MMFF, OPLS, AMOEBA, and



HIPPO.<sup>37</sup> The program has separate functions for MONTE and ANNEAL, though the method of performing Monte Carlo moves is different and based on Harold Scheraga's group's work. Individual atoms are moved or single torsional angles are changed, rather than Barker Watts rotations of the entire rigid body. ANNEAL method allows the user to choose the cooling schedule including linear, exponential and sigmoidal cooling. The sawtooth method derived from Topper et al. is not available as an option.

**Pysimm**<sup>38</sup>: Pysimm is molecular simulation program written in Python and is developed by Michael E Fortunato and Coray M. Molina at the University of Florida. However, users do have the option to rely on third party software depending on the needs of the simulation. Additional API has been added that allows the user the ability to use Cassandra to better perform Monte Carlo simulations.<sup>39</sup> The program is designed to solve problems in the canonical (NVT), grand-canonical ( $\mu$ VT) and isobaric–isothermal (NPT) molecular ensembles. The program is separated into three modules: `pysimm.system`, `pysimm.forcefield`, and `pysimm.lmps`. The system module creates and defines the particles, bonds, angles, and dihedrals of a system. The forcefield function allows the user to choose or define a forcefield, or use a third party force field for simulations. The lmps module uses LAMMPS to perform data handling needed for simulations. The downside of pysimm is that it currently does not have a separate annealing feature.

**OpenMM**<sup>40,41</sup>: Open source Python program library developed at Stanford University with unique features. These include custom force parameters between atoms with the use of strings, and GPU Acceleration for faster calculations. It is important to note that OpenMM is not a program itself, and instead a collection of python programs, chains these together to perform a

simulation. Users can use scripts to perform the calculations flexibly, but the program is focused on molecular dynamics than Monte Carlo simulations. Simulated annealing is possible in the program; however, the user must manually create the cooling schedule.

## Annealing

An important aspect of the program is the use of simulated annealing to find the lowest energy configuration. As a probabilistic method, annealing is used to find the global minimum of a cost function.<sup>3</sup> It is conceptually related to the method historically used to give steel better material properties during forging. In sword making, steel would be repeatedly heated and allowed to cool slowly in a process called normalizing and annealing, to improve the final grain structure of the material.<sup>42</sup>

In a similar way, annealing is used in the mag-walking program to slowly cool a system starting at a very high temperature (1000K for example), and then cool it down to absolute zero and find the lowest energy configuration of the material. The aim of this method in the context of simulations is to allow the program to sample the space of configurations ergodically, as nearly all states are accessible at the high temperature, and so once the system is cooled, the system is capable of reaching a global minimum. Since the Boltzmann factor would result in an undefined number for uphill moves at absolute zero, the program is adjusted so that any moves that increase the energy of the system are automatically rejected at 0K.

There are several annealing schedules by which a system can be cooled. The most accessible and easy to implement is in the linear form, like the work of Dosso and Oldenurg in which they use a constant factor  $\epsilon$  similar to  $\alpha$  to linearly cool the system during annealing:<sup>43</sup>

$$T_{k+1} = \alpha \cdot T_k \quad (54)$$

The temperature can also be dropped exponentially. An example is in a paper by Kirkpatrick et al. outlining a connection between statistical mechanics studying large systems with a high degree of freedom in thermal equilibrium to combinatorial problems. The authors use exponential annealing in the form:<sup>44</sup>

$$T_n = \left(\frac{T_1}{T_0}\right)^n T_0 \quad (55)$$

where the ratio  $\frac{T_1}{T_0}$  is already known and is less than one. This can be rewritten in the form:

$$T_n = \alpha^n T_0 \quad (56)$$

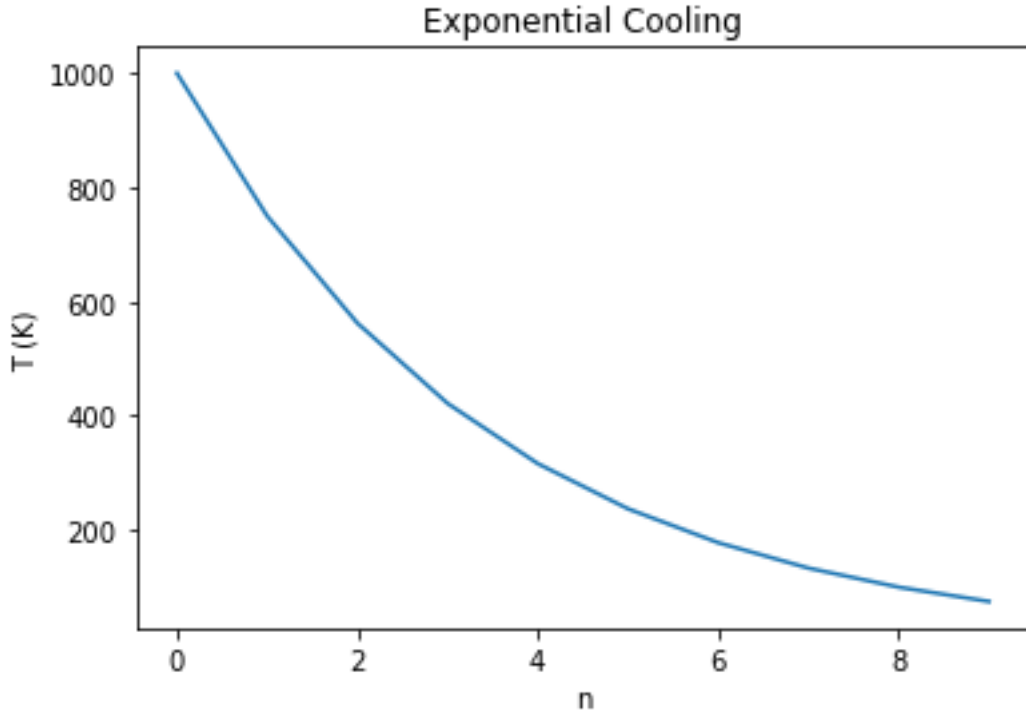


Figure 7: Exponential Cooling from 1000K where  $\alpha = 0.75$

The method of annealing used in the program is called sawtooth annealing.<sup>45</sup> In this form of annealing, the system is cooled linearly, but after reaching absolute zero, the temperature is raised once again and the process repeats. The new highest temperature is low than the previous highest. An example of an annealing schedule is given in Figure 8.

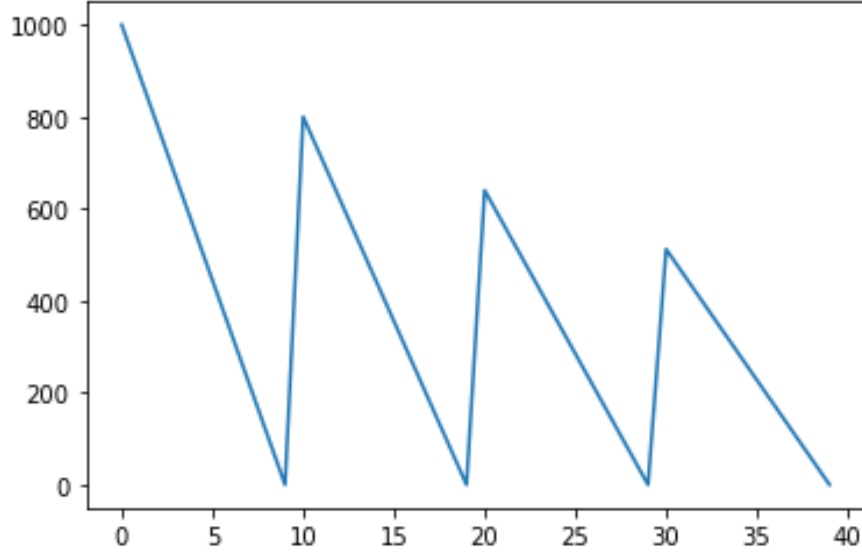


Figure 8: Example of Sawtooth Annealing Schedule with 4 Teeth ( $\alpha = 0.8$ )

In this form of annealing, the system is first simulated at the maximum temperature,  $T_{max}$ , which was set to 1000K in Figure 8. Then the system is cooled down to absolute zero. The number of temperatures it takes to get to absolute zero or the number of temperatures per tooth is dependent on the study. In this study:

$$N_{tooth} = 10 \quad (57)$$

Once the system reaches 0K, it is once again heated but not to  $T_{max}$ . It is instead heated to some smaller high temperature:

$$T = \alpha \cdot T_{max} \quad (58)$$

This process is repeated so that the length of each tooth is lower than the previous tooth as seen in Figure 8. Annealing runs were performed with  $\alpha = 0.8$ . This allows for the system to explore all possible configurations before being cooled down, making it possible to reach the global minimum.

## Program Structure

The general structure of the program is given in Figure 9. There are four files called on by the simulation, which will be called primary files. A table of file names is given in Table 2. The file *initial\_configuration.py* file serves two roles. It contains the code of the molecule class which allows the program to object oriented instances of molecules. It also contains a function to create a system of molecules, which the simulation file then modifies in each run.

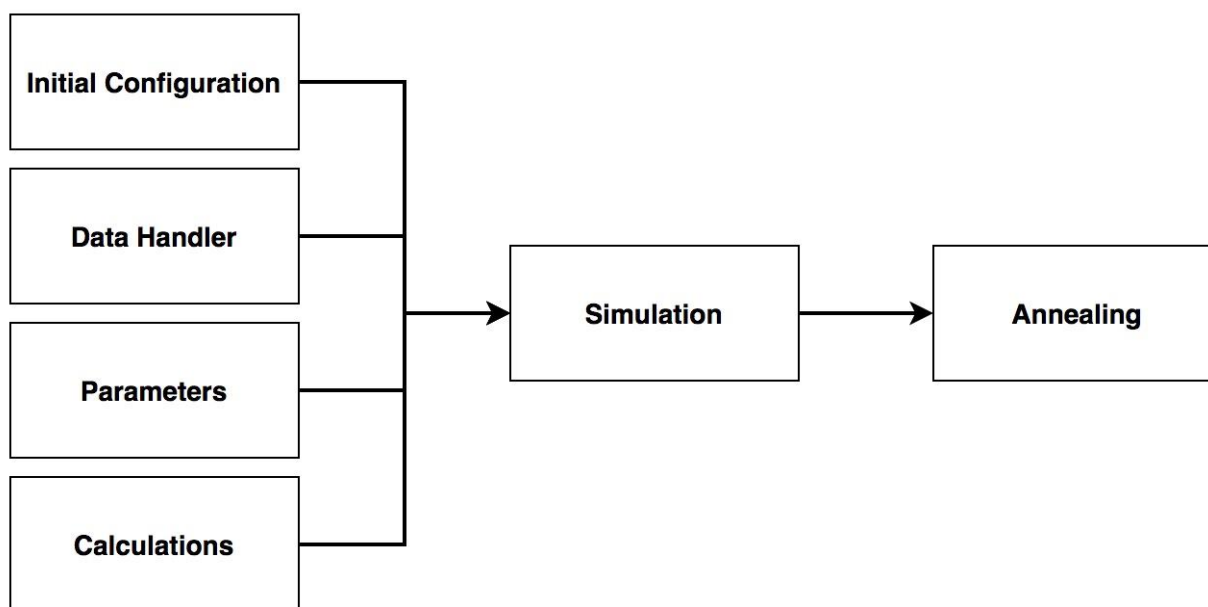


Figure 9: General Workflow of Program

The purpose of *data\_handler.py* is to deal with the collection of data during a run in the simulation, and to convert each molecule list into an *xyz* file. The number of times a configuration is saved can be controlled by the user in the simulation file. Parameters contain constants that may be needed for primary files, as well as constants that may be used for different types of simulations such as atomic masses and atomic radii.<sup>4647</sup> Calculations takes on the role of

performing important calculations including the potential energy, distance, etc. It also serves as a place where future additions to calculations could be made.

The *sim.py* program contains the general function to perform a Monte Carlo run at a particular temperature for a given system. It usually starts by generating a random initial state using the function from the *Initial Configuration* file and then performs the modified metropolis algorithm on the system. Molecules are chosen one at a time, and randomly perturbed based on a given step size, and the change in potential is calculated.

<b>Function</b>	<b>Filename</b>
Initial Configuration	<i>initial_configuration.py</i>
Data Handler	<i>data_handler.py</i>
Parameters	<i>parameters.py</i>
Calculations	<i>calculations.py</i>
Simulation	<i>sim.py</i>
Annealing	<i>anneal.py</i>

Table 2: Filenames of Programs Used

## Initial Configuration

The `initial_configuration.py` script serves as the backbone of the simulation as it creates the molecule objects and the first configuration of the system. The most important object-oriented portion of the code is created in this file and will be described here. A molecule can be created by calling on the *Molecule* class and assigning a variable to the object. An atom can similarly be created by using the *Atom* class.

When the class is called on it requires one input, which is the name of molecule being created. This input is taken by the special Python method `__init__`, which initializes a Python object. The `__init__` function does not always need an input, but in this case the name of the molecule is necessary to create the initial properties of the molecule object. The `__init__` function in this work assigns the molecule properties including its mass, starting position, center of mass, size, and collision sphere. Included default input molecules and atoms are given in Table 3 and Table 4. If a molecule type is not included by default, the user has the option to use the custom option. In this case the user would need to provide two inputs: “custom” and the path to an xyz file of the molecule.

Name	Accepted Inputs
Water	“H2O” or “Water”
Sulfur Dioxide	“SO2”
Nitrogen	“N2” or “Nitrogen”
Ammonia	“NH3” or “Ammonia”
Ammonium	“NH4” or “Ammonium”
Sulfuric Acid	“H2SO4”
Sulfur Tetrafluoride	“SF4”

Table 3: List of Default Inputs for *Molecule*



Name	Accepted Inputs
Fluorine	“F” or “Fluorine”
Chlorine	“Cl” or “Chlorine”
Bromine	“Br” or “Bromine”
Iodine	“I” or “Iodine”

Table 4: List of Default Inputs for *Atom*

The input case of the string is not important, and inputs are flexible in this regard. All strings are converted to upper case. Thus, an input such as “Water” is converted to “WATER” and “h2o” is converted to “H2O” before checking if it is a valid input and helps avoid unnecessary errors due to capitalization.

As an example,  $x = \text{molecule}(\text{“H2O”})$  will assign  $x$  as a molecule object. The variable  $x$  will have the default positions assigned to water molecules in the `__init__` function. Once the molecule is made these properties and variables can now be adjusted by the user directly or by a function. These values, or instance variables, are important as they allow the user to quickly learn the same type of information (mass, center of mass, etc.) for multiple molecules using the same syntax.

Object Variable	Explanation
name	Name of molecule as a string used by functions for storing data
size	Number of atoms in molecule
atom_names	List of atoms individually (this is used in calculations and to store data)
col_sphere	Collision distance sphere
mass	Mass of molecule
center	Center of mass coordinates
position	Position matrix of molecule
com_frame	Position matrix with respect to center of mass
previous_position	Previous position is saved for when a trial move is rejected

Table 5: Object Variables Associated with *Molecule* Objects

The way to access object variables has a particular syntax. If we create a molecule,  $x = \text{molecule}(\text{“water”})$ , then the method to access the position matrix or mass of  $x$ , are given by:

*x.position*

*x.mass*

Similarly, there also functions that are tied to molecule and atom objects, which are termed methods in Python, are also contained within the *Molecule* and *Atom* classes. There are eight key methods in the *Molecule* class and 3 in the *Atom* class. Each performs a calculation on a particular instance of the molecule. These are given in Table 6 and Table 7.

Method Name	Input	Purpose
<i>COM()</i>	None	Calculates center of mass of the molecule
<i>COM_FRAME()</i>	None	Finds position matrix with center of mass as origin (used for rotations)
<i>translate()</i>	delta (vector)	Translationally move molecule
<i>xrotation()</i>	theta (scalar)	Rotate molecule about the x axis
<i>yrotation()</i>	theta (scalar)	Rotate molecule about the y axis
<i>zrotation()</i>	theta (scalar)	Rotate molecule about the z axis
<i>update()</i>	None	Updates center of mass and COM frame position matrix after a move
<i>random_move()</i>	dist, theta	Randomly moves molecule, with maximum translation dist, and maximum rotation theta.

Table 6: List of Methods in *Molecule* Class

Method Name	Input	Purpose
<i>translate()</i>	delta (vector)	Translationally moves atom
<i>update()</i>	None	Updates center of mass
<i>random_move()</i>	dist	Randomly moves atom maximum distance <i>dist</i>

Table 7: List of Methods in *Atom* Class

An important detail to notice is that the syntax of methods is slightly different than functions.

Methods applied to an object are used by typing dot after the name of the object. An example of using these methods can be:

```
x = molecule("SO2")
x.translate([1,1,1])
x.update()
```

In this sequence, the variable *x* was assigned sulfur dioxide. The molecule is then translated along the *x*, *y*, and *z* axes. Lastly the center of mass and atom coordinates relative to it are updated.

The following is a more vectorized example and shows how multiple molecules as a system could be contained and manipulated.

```
M = []  
M.append(molecule("Water"))  
M.append(molecule("SO2"))  
M.append(molecule("Water"))  
for i in range(3):  
    M[i].random_move(dist=3, theta=90)  
    M[i].update()
```

In this longer sequence, the operations performed are very similar to those performed on *x* above. The difference here is that we first create an empty list *M* as a container for the molecules. Then one molecule is added at a time to the container *M*, to give a list of 3 molecules. Since a list is being used, to access the first molecule, one needs to use the list syntax to access elements, which is *M[0]* (Python indices start at 0).

These are 2 molecules of water and one of sulfur dioxide. Once the list has been made, it is important to note that all the molecules are centered at the origin by default and are currently overlapping. Thus, the molecules are each to be moved randomly by the for loop. The variable *i* is used to iterate over the 3 molecules, and randomly moves them and updates the molecule centers.

With the molecule class described, there is the remaining *Create\_System* class, containing two methods *homogeneous()* and *heterogeneous()*. Both perform a process very similar to the second example of code using methods. *homogeneous()* is used to create a system of molecules that are homogenous and contained within the variable *M*, similar to the example. It

can also store the position data of all molecules into an *xyz* file, which a user can then use to view the molecules with another program such as Avogadro.<sup>25</sup>

Input	Explanation
Npart	Number of atoms and molecules to create
Mol_Class	Class of object to create. Allows a modified class instead of <i>Molecule</i> to be used
moleculeName	Molecule name
create_xyz	Will create xyz file if True
save_path	Save location of xyz files

Table 8: Inputs for *homogeneous()* method

The function has one required input, and 4 optional inputs. The required input is number of molecules. Optional inputs in Python functions have a default value set and used within the function, which can be changed by the user. For this function the default molecule is water of the *Molecule* class and will automatically store the xyz file. The molecule class used is allowed to be varied in case the user wants to use a modified *Molecule* class, as well the option to create an xyz file, and the file save name. An example of a homogenous system of water molecules created by *homogeneous()* is given in Figure 10.

*heterogeneous()* works similarly to *homogeneous* except that it requires an input dictionary. This dictionary will contain the amounts of each molecule to be created. The molecule and atom classes are also inputs. A table of inputs is given in Table 9.

Variable	Explanation
species	dictionary of species
Mol_Class	Class of molecule ( <i>Molecule</i> by default)
Atom_Class	Class of atom ( <i>Atom</i> by default)
create_xyz	Boolean to determine if xyz file created
save_path	Location to save xyz file

Table 9: Inputs for *heterogeneous* method

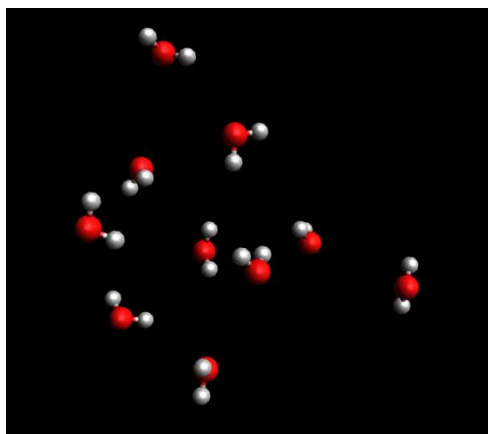


Figure 10: Initial Water Configurations of 10 Molecules

An example species dictionary would be:  $Species = \{ "NH_3": 1, "HCl": 1 \}$

This dictionary would create a mixed system of one ammonia and one hydrogen chloride system.

Some other example of systems that can be generated by the script are given in Figure 11

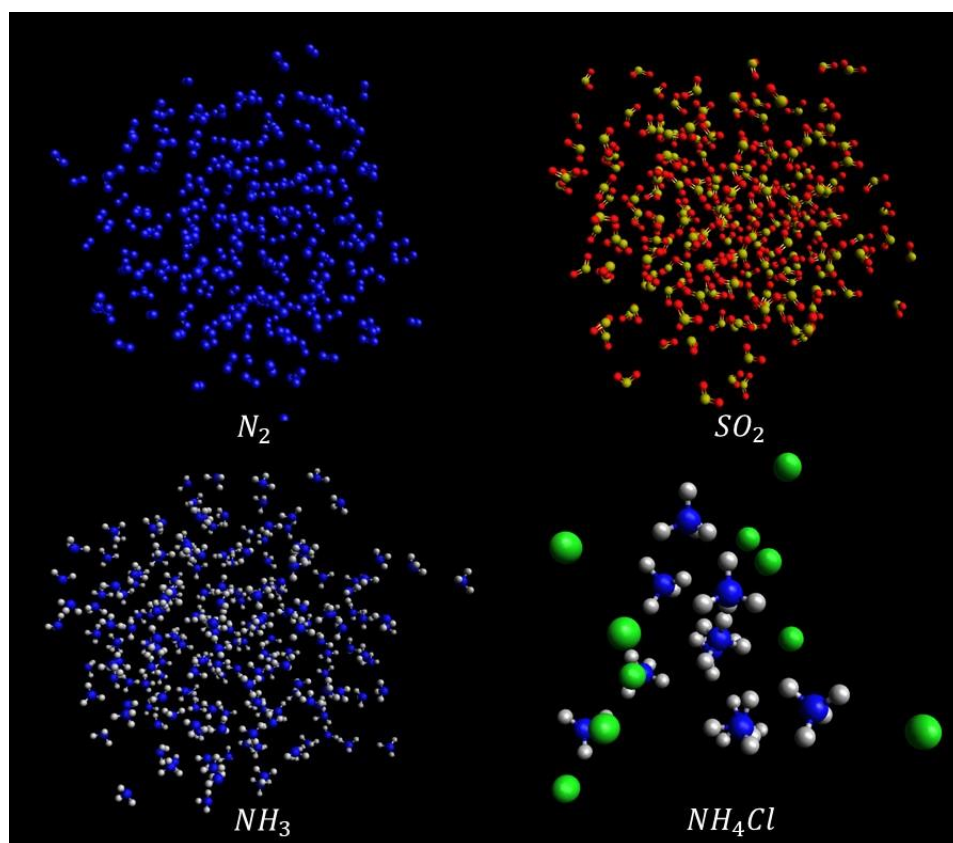


Figure 11: Example Possible Configurations for  $N_2$ ,  $SO_2$ ,  $NH_3$ , and  $NH_4Cl$  Systems

## Simulation Program

There is a single simulation file that handles both homogeneous systems and mixed simulations: `sim.py`. The simulation file contains the `Simulation` class which is able to perform a Monte Carlo simulation via the “`run()`” method within the class. It is important to note that the simulation must first be fed an initial molecular array to operate on before it can perform any trials. This initial system is generated by the `initial_configuration.py` file. A separate run file should be created for each particular simulation as the simulation parameters may need to be altered to suit the system being studied, and this may include changes to the *Molecule* class itself. In the study of TIP3P molecules the charges and the potential energy function parameters are based on the model, and therefore these aspects needed to be taken into account by the molecule class. However, Python has class inheritance which is useful in this context. A child class can be created which inherits functions and methods from its parent class, but also has different variables and methods. For example, a TIP3P class can be created, which takes into account the TIP3P charge distribution, but also inherits the mass and initial position from *Molecule*.

The `Simulation` function carries out the modified Metropolis algorithm by mag-walking at a given temperature as outlined earlier. The function has several inputs given in Table 10.

The outputs of the function are the number of accepted, rejected, and total moves, as well as the final configuration of the system, and arrays storing the energy of the system, which includes the potential energy from Lennard Jones potential, Coulombic force, and constraint potential. These can then be stored into a text file by the user if they choose by using the numpy library command `numpy.savetxt()`.

Input	Explanation
trials	Number of trials
n_eq	Equilibration period (default is 0)
temperature	Temperature (default is 20)
save	Iterations before saving data (default is 100)
step_size	Initial step size
step_check	Iterations before correcting step size
outputPath	Path to save location of file
seeding	Allows user to choose starting seed value
debug	Allows user to print log of moves to help debug code
potential_list	List of potential class names to be used
diffCenter	Use different point for Lennard Jones potential
LJCenter	Pick alternative center point by index

Table 10: Table of Inputs into the Simulation Program

The outputs from the run() function are stored into a dictionary. These can be accessed with the following keys:

Output	Explanation
Energy_Store	Stores all energies for each potential and total in a dataframe
Accepted_Energies	Stores all accepted energies for each potential and total in a dataframe
Acceptance	Stores into array accepted, rejected, and total moves in an array
M_fin	Final configuration array
step_size	Final step size
dataLog	(Optional) data log is printed if debugging

Table 11: Table of Keys for Output Dictionary

## Annealing Program

The conceptual basis for the Annealing program has been discussed in the annealing section earlier. The Annealing program is one the simpler programs and contains two nested loops. One over the number of temperatures per tooth and another over the teeth. Both of these parameters are set by the user. In each tooth, a simulation is performed at the highest temperature and the energy data is saved to text files in the folder “Data”, and xyz files are created based on the user input, but by default at the end of each individual simulation. The xyz files are saved into one folder “Movie”, where each file is saved separately but there is one movie file where each simulation appends the file by adding its final configuration.

The inputs required by the user are given in Table 12.

User Input	Explanation
T_max	Max temperature in first tooth
N_PT	Number of temperatures per tooth
N_T	Number of teeth
alpha	Factor by which T_max is reduced
DATA_LOC	Location to save text files of energy data
XYZ_LOC	Location to save xyz files
RUNS	Number of trials per simulation
NEQ	Equilibration period
SAVE	Number of iterations before saving
CHECK	Number of iterations before adjusting step size

Table 12: Required Input from User for Annealing Program



## Directions For Use of Program

General directions for use of the program are as follows:

1. It is best to start with the initial configuration and test creating systems and moving objects to understand how the object-oriented variables work firsthand.
2. Once the user understands the *Molecule* and *Atom* classes and manipulating an object list, *M*, the user is ready to move on to the simulation program.
3. The simulation program contains most default inputs and can be run easily, but the user may want to take time to understand how the code inside the *Simulation* class performs Monte Carlo simulations and uses the object classes simultaneously and its inputs and expected outputs should be reviewed.
4. The user will first need to assign *Simulation.M* (object array for the simulation) to an initial configuration using the *Create\_System* class when performing a single trial. When annealing, the first state needs to be assigned as well but the program will then automatically assign the output configuration of the previous simulation as the input for the next simulation. Once the initial state is created, the simulation can be performed by the *Simulation.run* method and passing through the input dictionary.
5. The debug option in the *Simulation.run* function will be useful as it allows the user to see trial moves, their energies, and why they were accepted or rejected. This should aid in understanding how the modified Monte Carlo method is applied.
6. The default input dictionary and its default values are a guide for what parameters to use for simulation. The user must keep in mind that Lennard Jones parameters are assumed to be constant when they may need to use mixing rules and create arrays or dictionaries for their choice of system.
7. The user can perform a single simulation using only the run file or move to simulated annealing to search for the lowest energy configuration of a cluster.
8. csv files are created to store the energies and accepted energies of the system automatically for simulated. Structures are saved as xyz based on the *save* variable for each trial, and the final structure of a simulations is added to the *Movie.xyz* file.

## Model Testing

It is important to note that a Monte Carlo simulation will not immediately optimally sample from the Boltzmann distribution and is only guaranteed to do so on the order of a large number of trials.<sup>10</sup> It is thus important to allow the simulation to equilibrate by going through a large number of moves before collecting data on the simulation. The number of simulations needed for equilibration can be found by plotting the running average of the energy and heat capacity against the number of trials performed and observing when the plot begins to stabilize. The case of two TIP3P water molecules interacting via a Lennard-Jones interaction and a Coulombic potential is given in Figure 12.

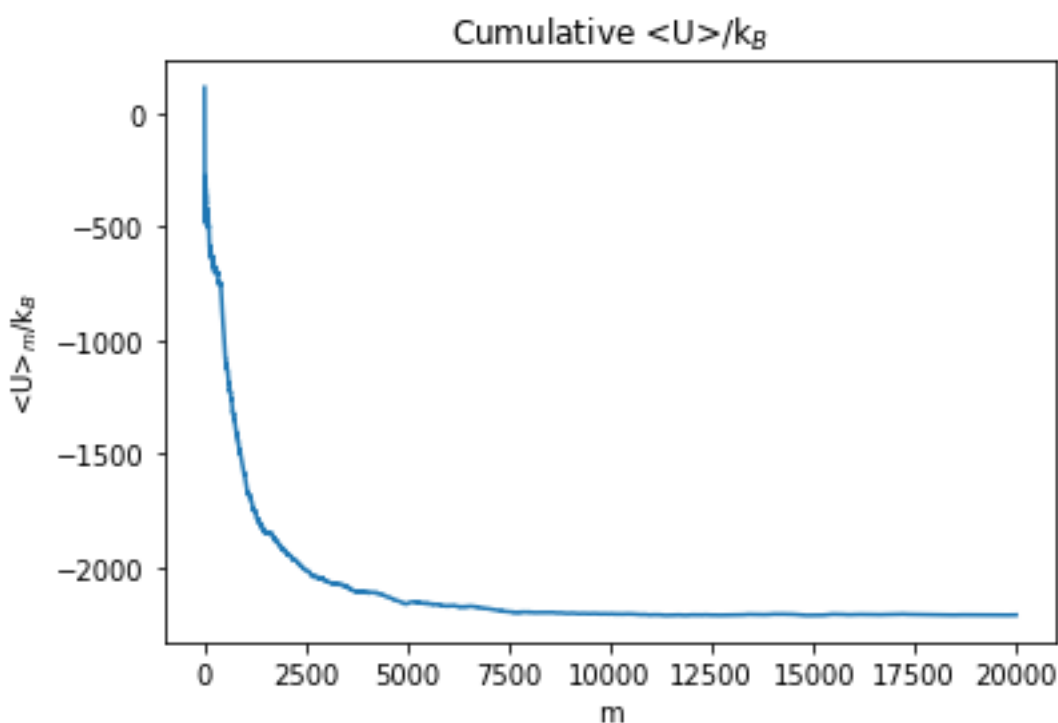


Figure 12: Cumulative  $\langle U \rangle / k_B$  for a simulation with 20000 Trials

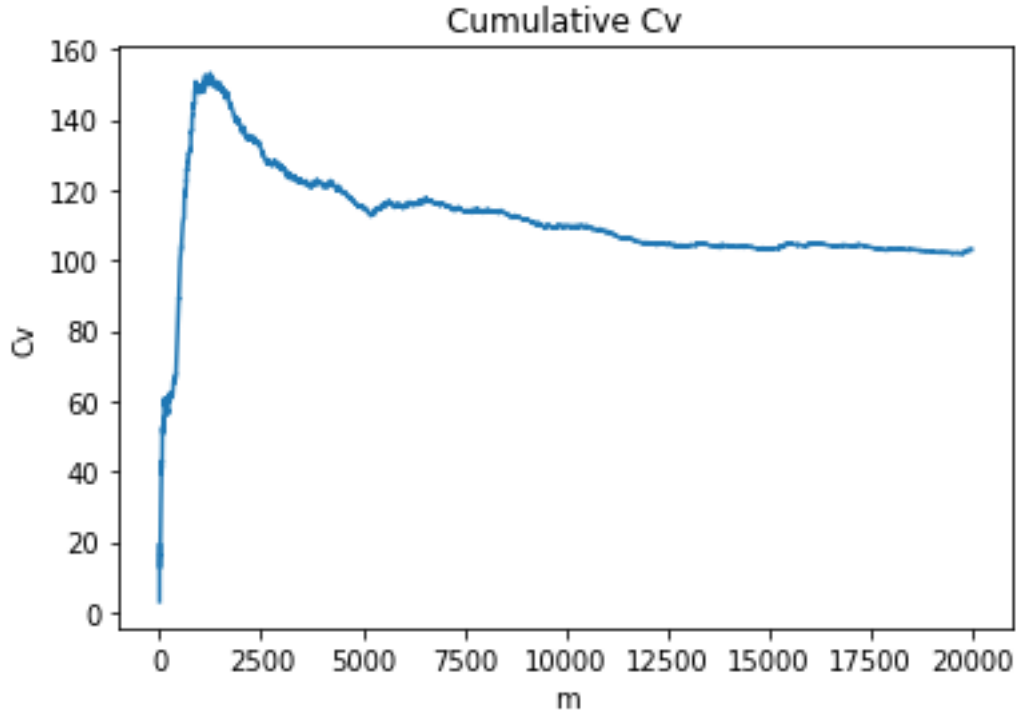


Figure 13: Cumulative  $C_v$  for the Simulation in Figure 12

Graphs were used to determine the number of simulations needed before simulation data could be collected. As the plot for  $C_v$  shows, the heat capacity takes longer to equilibrate than the average energy. Based on these plots, 10,000 trials need to be performed for equilibration. As computer resources were limited in this study, the total number of trials was made 50,000 total, out of which 10,000 trials would be used for equilibration.

N	Trials			
	1000	5000	10000	20000
2	1	4.729512	7.268109	12.18664
3	2.713746	7.040892	12.47766	23.56971
4	4.467054	11.82779	22.82199	41.30317
5	6.798613	17.78813	31.99771	59.51511

Table 13: Relative Runtime Table for varying Trial Count and System Size

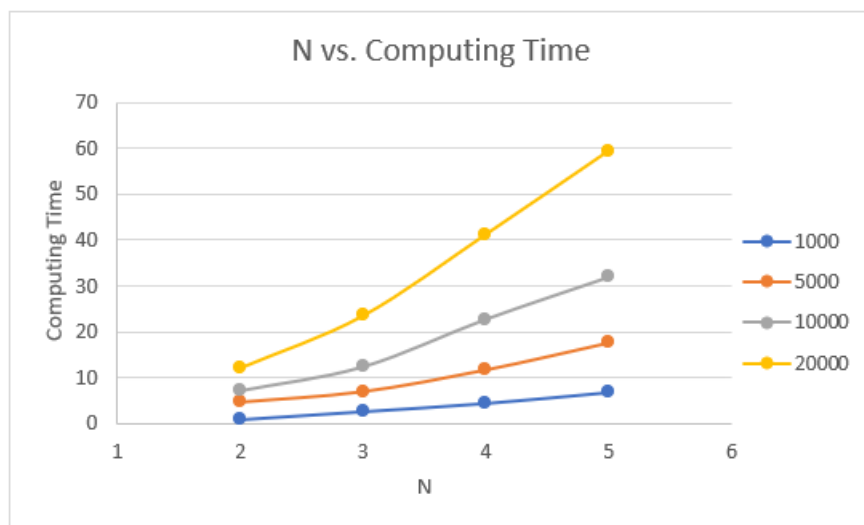


Figure 14: Plot of System Size vs. Computing Time

A runtime analysis of different trials of different system sizes was also performed for which the results are shown in Table 13. Figure 14 and Figure 15 show plots of runtime against the number of molecules and number of trials, respectively. The runtimes in the chart are unitless and calculated in relation to the calculation time for 1000 trials on a system of size  $N=2$ . From the figures one can see that runtime increases linearly with trials but non-linearly with system size. This is to be expected as each new molecule in the system adds an additional  $N - 1$  interactions with the other molecules to calculate. It is important to include in the considerations that run times will depend heavily on the machine being used to perform the calculations. Calculations in the present work were performed on a laptop, which yielded less than average results and made simulations with bigger system sizes infeasible.

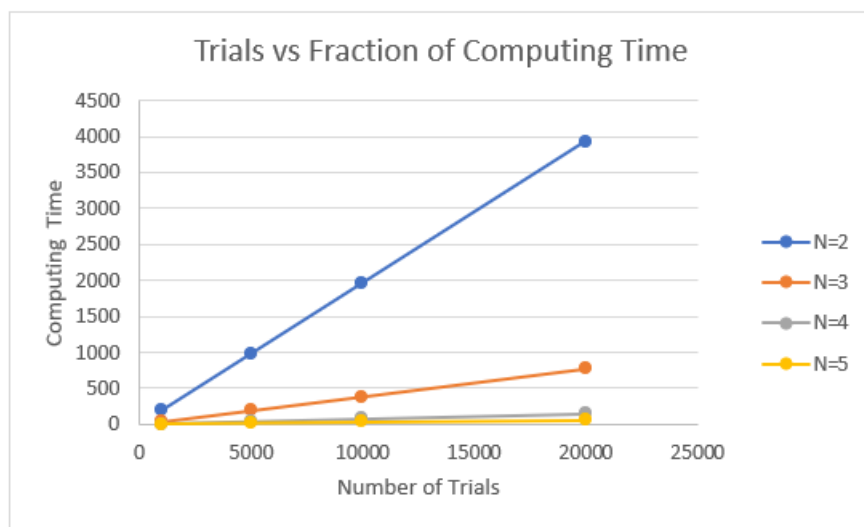


Figure 15: Plot of Trials vs. Computing Time

### Molecular Geometries

As a simplification, molecular geometries in the program have been treated as static. However, molecular geometries and bonds of the molecules studied are not static in reality and are subject to many forms of motion. These include torsional rotations of bonds, as well as vibrational and translational motion of atoms along the bonds. Leaving geometries static leads to the question of which geometry should the user choose for their molecule in the simulation. Generally, the choice will heavily depend on the study being performed. Initial geometries can be generated by certain programs like SPARTAN using methods such as ab initio methods such as Hartree-Fock or Density Functional Theory to estimate the geometry of the molecule<sup>32</sup>. Another possibility is the use of a database such as CCCBDB provided by NIST<sup>48</sup>. The database provides structures for molecules from experiment as well as calculated structures. Experimental structures with rotational constants are provided with literature, and calculated structures using methods including Hartree-Fock, Density Functional theory, Møller-Plesset perturbation theory,

and other calculated values are available. In the case of the water clusters, several models are already available for study.

### Water Models:

Water molecules have been investigated in the literature extensively.<sup>11,49,50,51</sup> There are several different models available that account for different properties of the water molecule. Two-site models model the dipole behavior of water models well. Three site models, as seen in TIP3P used in the current work as well as in TIPS and SPC, represent the three atoms in water individually<sup>52</sup>. Higher site models take into account the lone pairs of oxygen in water molecules to better model Coulombic interactions between molecules. The TIP3P model was chosen as it well studied, and cluster data is readily available.<sup>53,54</sup>

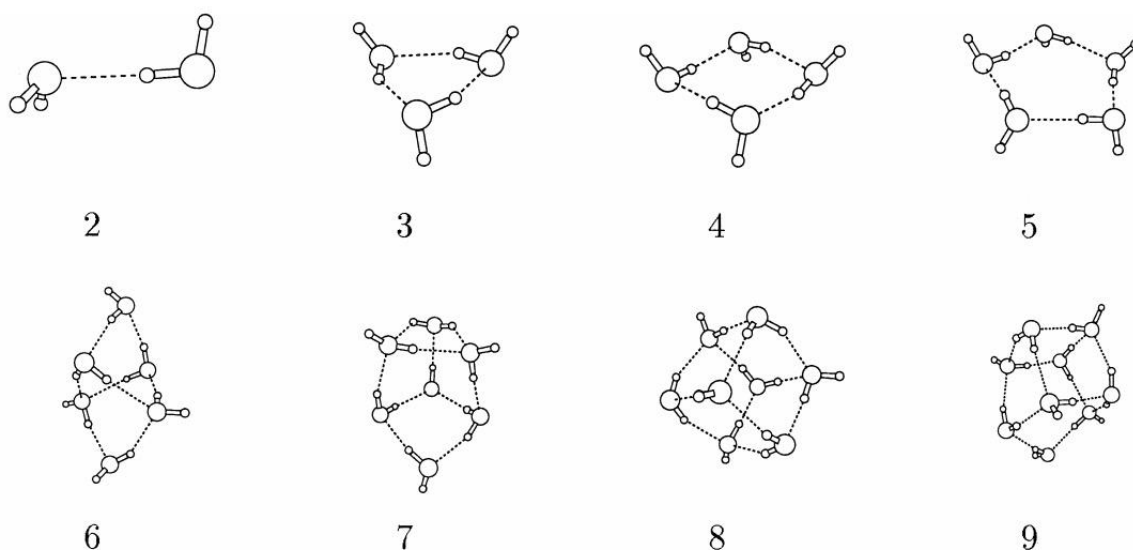


Figure 16: Molecular Geometries (Reprinted from Reference 55)

The TIP3P model water was used to test the annealing algorithm where were performed by sawtooth annealing with 50,000 iterations per temperature with 10 temperatures per tooth and

4 teeth per run. Geometries were found for  $N = 2, 3$ , and 4 and can be compared with Figure 16 reprinted from a paper by Wales and Hodges.<sup>55</sup> Table 14 shows data in kcal/mol from the work of Niesse and Mayne for global minima for different cluster sizes. Figure 17 and Figure 18 show plots of accepted energies for the case of  $N=3$  and  $N=4$  simulations, respectively. Figure 19 shows the results from annealing for 3 and 4 water molecule clusters.

<b>N</b>	<b>Energy (kJ/mol)</b>	<b>Energy (kcal/mol)</b>
2	-26.08757	-6.235085755
3	-69.99387	-16.72895489
4	-116.59042	-27.86580992
5	-152.109	-36.35496365
6	-197.78053	-47.27073335

Table 14: Minimum Energy Data from Database

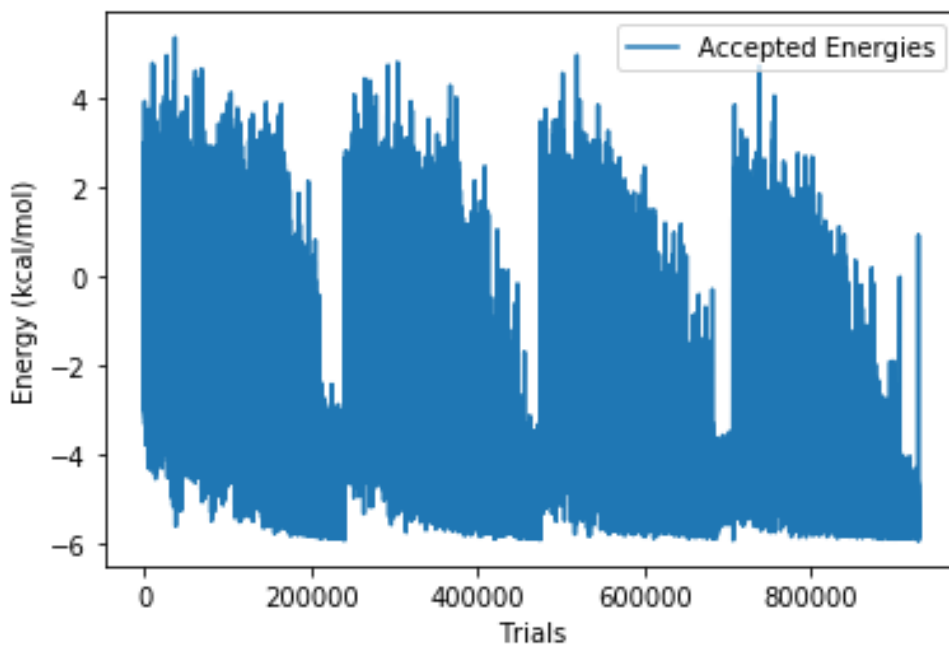


Figure 17: Accepted Energies for  $N = 3$  System

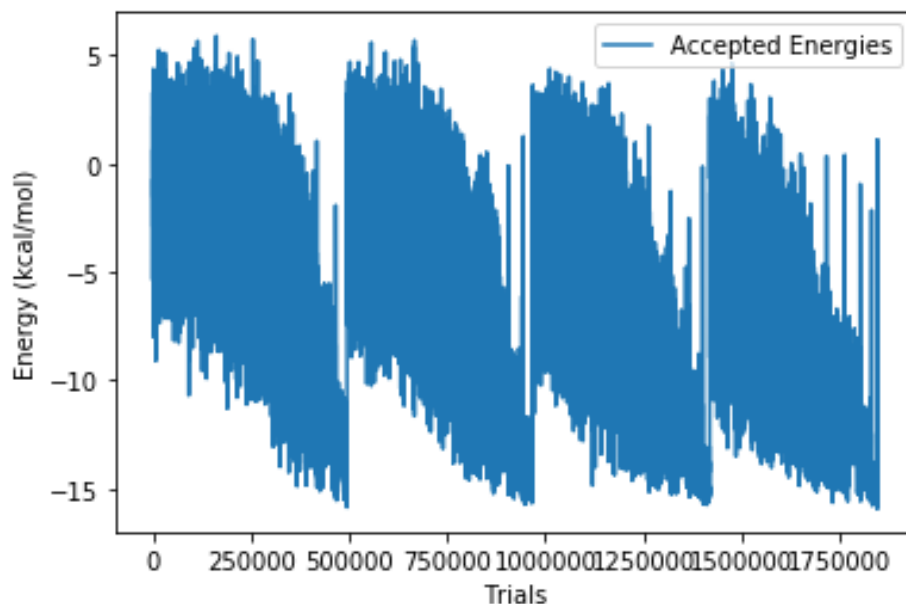


Figure 18: Accepted Energies for  $N = 4$  System

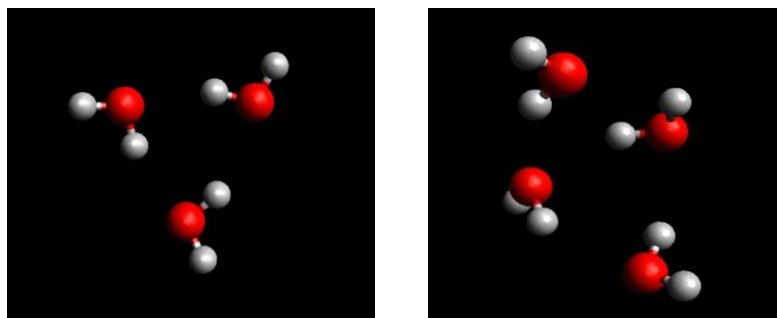


Figure 19: Final Structures for  $N=3, 4$  Water Clusters

From the figure, one can see that both models produced similar results. The potential energy minimum of the 2 particle case was found to be -6.1 kcal/mol, the 3 particle case was found to be -16.5 kcal/mol, and the 4 particle case was found to be -26.8 kcal/mol. These values are relatively close to the above values but they are not exactly the same, even though the models are the same. This suggests that sampling was not performed as efficiently as could be possible.



## **5 Molecule Case:**

The final homogenous simulation was simulating 5 water molecules, which was more difficult due to the large number of iterations combined with a bigger system. In this sawtooth annealing was performed, but only 30,000 iterations were used as one tooth on this cycle required needed 2 hours to perform, which is 200,000 total iterations. Three cycles were performed on the system. The lowest energy configuration had an energy of -32.92 kcal/mole. This value is well above the minimum in literature, suggesting sampling did not occur efficiently in the number of trials performed. The annealing schedule is given in Figure 20.

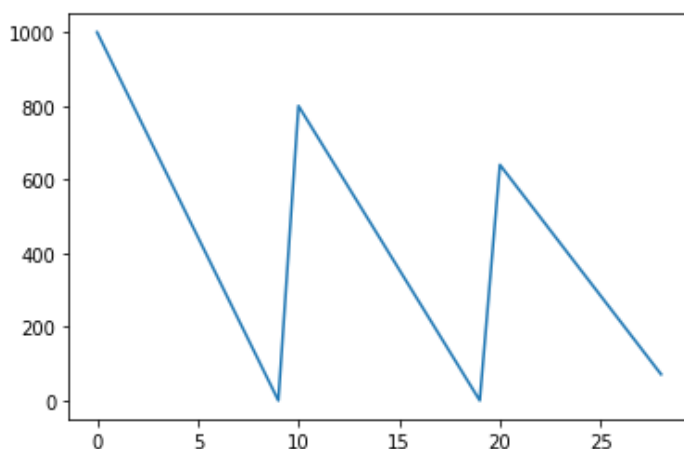


Figure 20: Annealing Schedule for System of 5 TIP3P Molecules

The total energy of the system, excluding the effect from the constraint potential, was also tracked. Energy from the three potential energy functions were saved separately, but one can also obtain it by subtracting the constraint potential from the total energy in the program.

$$E - E_{Constraint} = E_{LJ} + E_{Coulombic} \quad (31)$$

This total energy from the Lennard Jones and Coulombic forces was tracked separately throughout the annealing process. A graph of this total energy is given in Figure 21. From the

graph it is clear that the spread of accepted energies decreases as the system approaches lower temperatures giving the graphs a somewhat conical shape. Another graph was made of a running average of the accepted energies from Figure 16, given in Figure 17.

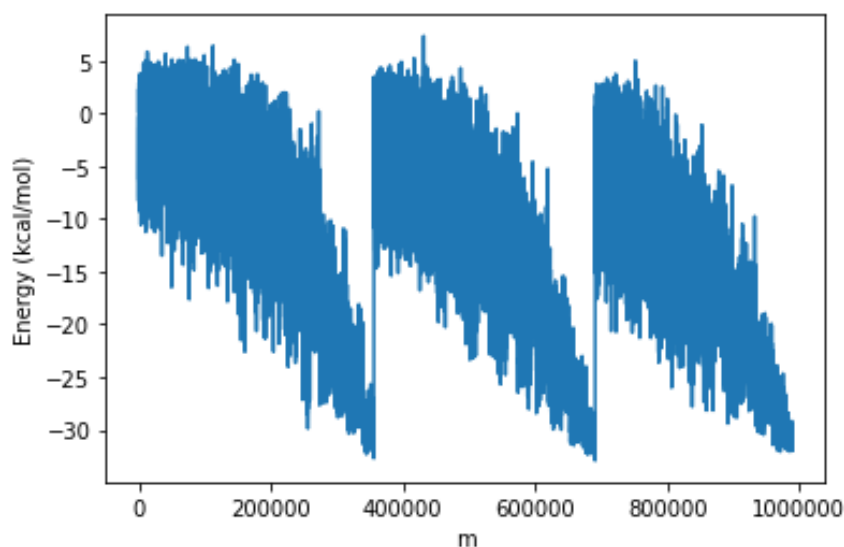


Figure 21: Graph of Accepted Energies for N=5 System

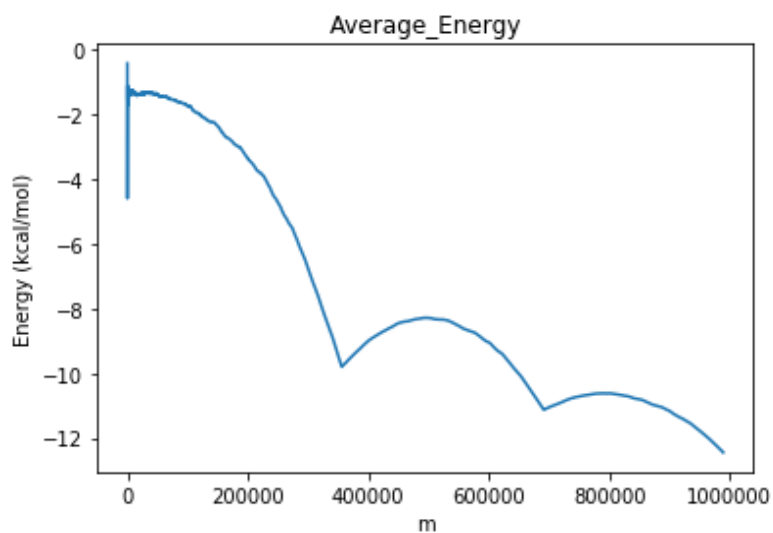


Figure 22: Running Average of Accepted Energies of System

The graph of the running average of accepted energies shows how the system is heating and cooling but tending towards a general lower energy state over the course of the simulation. The cooler trend is to be expected as the temperatures are also slowly decreasing in the sawtooth, but it is interesting to see that the minimum is also approaching lower values.

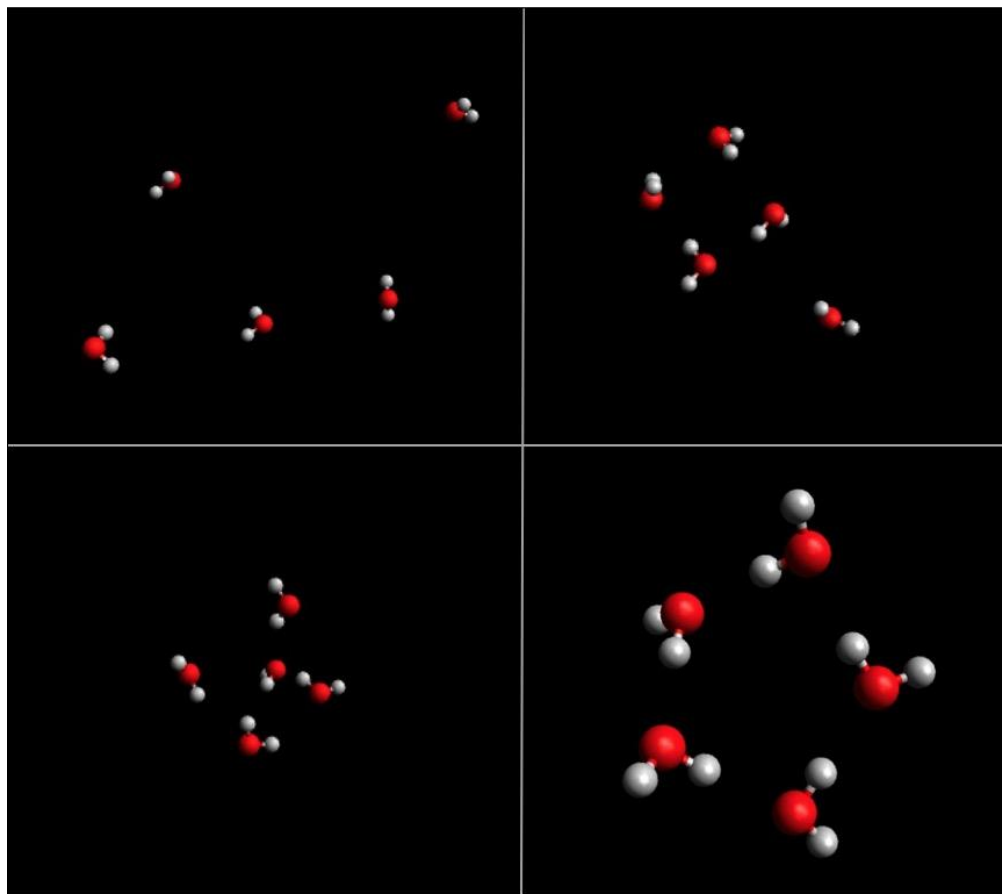


Figure 23: Evolution of Water Clusters for N=5

#### Heterogenous Cluster:

Ionic clusters of ammonium chloride  $(\text{NH}_4\text{Cl})_n$  were annealed and the final geometries were compared with the results Topper et al. for the cases of  $n = 1, 2, \text{and } 4$ .<sup>1</sup> The  $n = 1$  system is better described as an interaction between HCl and  $\text{NH}_3$ .<sup>1</sup> The parameters for the potential

function used are taken from a previous work on ammonium chloride simulations via the jump-walking algorithm by Topper et al.<sup>18</sup> The results from annealing are given in Figure 24.

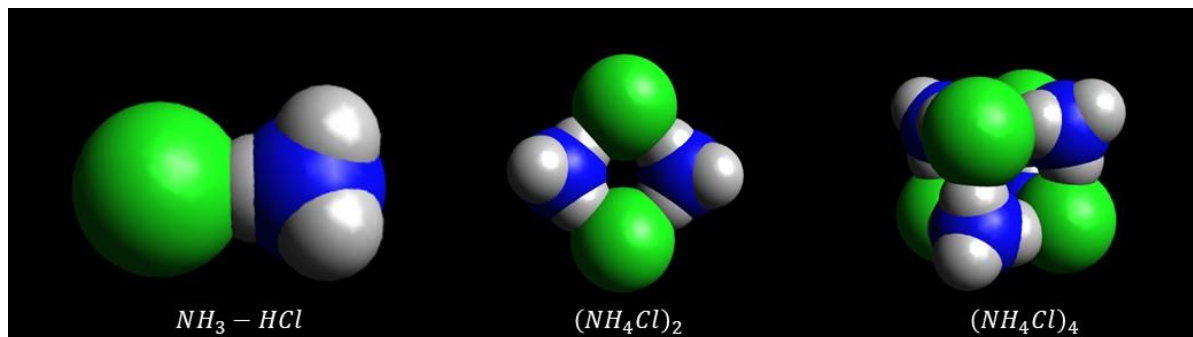


Figure 24: Heterogenous Cluster Results

For comparison, the original structures from Topper et al. are given in Figure 25. Qualitatively, the annealing algorithm converged to the optimal geometries for the given cluster configurations and were able to find the global minimum.

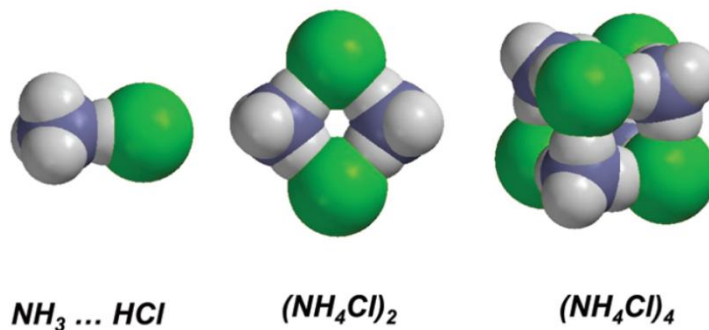


Figure 25: Optimal Ammonium Chloride Geometries (Reprinted with permission from Reference 1)

It is important to note that the performance of the annealing and mag-walking algorithm is measured by its ability to optimize and find the global minimum of a structure. An optimization algorithm can get “stuck” in a local minimum well and require several uphill moves by a Monte Carlo algorithm. The aim behind the modified algorithm is for the system to more efficiently be

sample all minimal configurations and find the lowest in energy, while not getting trapped in the process. An example of a local minimum structure with planar geometry is shown in Figure 26. The  $n = 4$  system explored this configuration before reaching the more stable cubical structure.

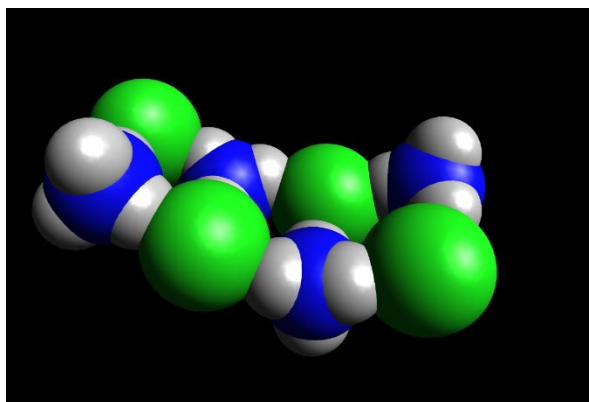


Figure 26: Local Minimum Structure for  $(\text{NH}_4\text{Cl})_4$

## Conclusions and Recommendations

The aim of this project was to create a program in Python that can simulate clusters that are both homogeneous and heterogeneous, using an object-oriented method so that simulations can be performed more conveniently and readily by a user. In this regard, the program has been completed and is ready for use. Further studies could be performed on larger clusters and simulations could be performed with a higher number of trials than used in the current study with more computational resources and get results that are more accurate. The current study used 50,000 trials per simulation, but ideally these should be 100,000 or higher to help sample ergodically, but these gave very high run times and errors on the machine being used to run the simulations.

It is also recommended in the future that the *Atom* class be updated to contain more atoms and be used as the parent class of the *Molecule* class. This would require some restructuring but would make the program more versatile as intramolecular movements could be tracked. As the program currently stands, molecules and atoms are treated as different objects, and the atoms available to use are halides intended for use in ammonium halide simulations.

## Appendix: Python Files

### File: `initial_configuration.py`

```
"""
Initial Array Generator

Creates molecule and atom objects and stores them into an array (M)
Each object has associated properties including mass, atom_names list, and
position array

Position Array Shape (object composed of m atoms):
      x    y    z
1:[x1  y1  z1]
2:[x2  y2  z2]
...
j:[xj  yj  zj]
...
m:[xm  ym  zm]

Particle positions in array:
M[i] = object i
M[i].position = positions for all atoms in object i
M[i].position[j] = will select row j in position array: [xj, yj, zj] for atom
j in object i
"""

## Import modules and define parameters
import numpy as np
import parameters as pm
import math
import calculations as calc
import data_handler as data
from copy import deepcopy
import os

MAXT = 0.2 # Maximum translation
MAXR = 360.0 # Maximum rotation
COL_DIST = 2 # [A] Collision distance
MAX_SEARCH = 100 # Max number of attempts to find a position that isn't
colliding with other particles
INCREASE_MOVE = 1.1 # Factor to increase move size if algorithm fails to find
position without colliding

# Parameters for creating save file fed to create_system function (Don't need
to write xyz, function already adds it)
molName = "H2O" # This is fed to molecule class or custom class to create
instance
saveFolder = "Sim" # Save folder in current directory
fileName = "mixed_sim" # File save name
filePath = os.path.join(os.getcwd(), saveFolder, fileName) # Save Location will
be CurrentDirectory/saveFolder/fileName
```

```

customPath = os.path.join(os.getcwd(), saveFolder, "Custom.xyz") # Path of
custom file

class Atom():
    # Initial Parameters
    def __init__(self, name, custom_path="Sim/Custom.xyz"):
        self.name = name # Specify type of molecule by name
        if self.name.upper() == "CL" or self.name.upper() == "CHLORINE":
            # Basic parameters
            self.name = "Cl"
            self.size = 1 # Number of atoms
            self.atom_names = ["Cl"] # Keys for atoms
            self.col_sphere = pm.Collision_Sphere["Cl"] # Collision-
sphere [A]
            self.mass = pm.AMU["Cl"] # Molecular mass
            self.center = [0.0, 0.0, 0.0] # Initial center is origin

            self.position = np.array([0.0, 0.0, 0.0]) # current in the
form [A1 A2 A3...] where Ai = [xi yi zi]^T

            self.previous_position = self.position.copy()
            self.type = "atom"
        else:
            self.type = None

    def translate(self, delta):
        self.position += np.array(delta)

    def update(self):
        self.center = self.position

    # Copies previous position and then randomly moves the particle
    def random_move(self, dist=1, theta=90):
        self.translate([dist*(2*np.random.random()-1), # Translate each
randomly
                        dist*(2*np.random.random()-1),
                        dist*(2*np.random.random()-1)])
        self.update()

# Creates a molecule with initial position given by self.position under the
__init__() function
# Each molecule has functions to find COM and positions with COM as
origin(COM_FRAME()) for rotations
class Molecule():
    # Initial Parameters
    def __init__(self, name, custom_path=customPath):
        self.name = name # Specify type of molecule by name
        if self.name.upper() == "H2O" or self.name.upper() == "WATER":
            # Basic parameters
            self.name = "H2O"
            self.size = 3 # Number of atoms
            self.atom_names = ["O", "H", "H"] # Keys for atoms

```



```

self.col_sphere = pm.Collision_Sphere["H2O"]    # Collision-
sphere [A]
self.mass = 2*pm.AMU["H"] + pm.AMU["O"] # Molecular mass
self.center = [0,0,0] # Initial center is origin

self.position = np.array([[0,0.06556811, 0], # Default position
                           [-0.757,-0.52043189, 0],
                           [0.757, -0.52043189, 0]])

# The molecule is initially centered at zero so the com_frame is
the position matrix
self.com_frame = self.position.copy()
self.previous_position = self.position.copy()
self.type = "molecule"

elif self.name.upper() == "SO2":
    self.name = "SO2"
    # Basic Parameters
    self.size = 3 # Number of atoms
    self.atom_names = ["S", "O", "O"] # Keys for atoms
    self.col_sphere = pm.Collision_Sphere["SO2"] # Collision-
sphere [A]
    self.mass = pm.AMU["S"]+ 2*pm.AMU["O"]
    self.position = np.array([[0,0,0],
                              [0,1.2371,0.7215],
                              [0,-1.2371,0.7215]]) # Initialize
position
    self.center = [0,0,0.3603716]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "N2" or self.name.upper() == "NITROGEN":
    self.name = "N2"
    # Basic Parameters
    self.size = 2 # Number of atoms
    self.atom_names = ["N", "N"] # Keys for atoms
    self.col_sphere = pm.Collision_Sphere["N2"] # Collision-
sphere [A]
    self.mass = 2*pm.AMU["N"]
    self.position = np.array([[0,0.54875,0],
                              [0,-0.54875,0]]) # Initialize position
    self.center = [0,0,0]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "HCL":
    self.name = "HCL"
    # Basic Parameters
    self.size = 2 # Number of atoms
    self.atom_names = ["Cl", "H"] # Keys for atoms

```

```

sphere [A]
    self.col_sphere = pm.Collision_Sphere["HCl"]           # Collision-
self.mass = 2*pm.AMU["N"]
self.position = np.array([[0, 0, 0],
                           [0, 0, 1.2746]]) # Initialize position
self.center = [0,0,0]
self.com_frame = self.position.copy()
self.previous_position = self.position.copy()
self.type = "molecule"

elif self.name.upper() == "NH3" or self.name.upper() == "AMMONIA":
    self.name = "NH3"

    # Basic Parameters
    self.size = 4
    self.atom_names = ["N", "H", "H", "H"]
    self.col_sphere = pm.Collision_Sphere["NH3"]
    self.mass = pm.AMU["N"] + 3*pm.AMU["H"]

    self.position = np.array([[0, 0, 0.1111],
                              [0, 0.9316, -0.2592],
                              [0.8068, -0.4658, -0.2592],
                              [-0.8068, -0.4658, -0.2592]])

    self.center = [0,0,0.04535749]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "NH4" or self.name.upper() == "AMMONIUM":
    self.name = "NH4"

    # Basic Parameters
    self.size = 5
    self.atom_names = ["N", "H", "H", "H", "H"]
    self.col_sphere = pm.Collision_Sphere["NH4"]
    self.mass = pm.AMU["N"] + 4*pm.AMU["H"]

    self.position = np.array([[0,0,0],
                              [0.5939,0.5939,0.5939],
                              [-0.5939,-0.5939,0.5939],
                              [-0.5939,0.5939,-0.5939],
                              [0.5939,-0.5939,-0.5939]])

    self.center = [0,0,0]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "SF4":
    self.size = 5
    # Basic Parameters
    self.atom_names = ["S", "F", "F", "F", "F"]
    self.col_sphere = pm.Collision_Sphere["SF4"]

```

```

self.mass = pm.AMU["S"] + 4*pm.AMU["F"]
self.position = np.array([ [0,0,0.3825],
                           [0,1.6255,0.2401],
                           [0,-1.6255,0.2401],
                           [1.2055,0,-0.581],
                           [-1.2055,0,-0.581] ])

self.center = [0,0,-0.00606887]
self.com_frame = self.position.copy()
self.previous_position = self.position.copy()
self.type = "molecule"

elif self.name.upper() == "SF6":
    self.size = 7
    # Basic Parameters
    self.atom_names = ["S", "F", "F", "F", "F", "F", "F"]
    self.col_sphere = pm.Collision_Sphere["SF6"]
    self.mass = pm.AMU["S"] + 6*pm.AMU["F"]
    self.position = np.array([ [0,0,0],
                               [0,0,1.554],
                               [0,1.554,0],
                               [1.554,0,0],
                               [0,-1.554,0],
                               [-1.554,0,0],
                               [0,0,-1.554]])

    self.center = [0,0,0]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "H2SO4":
    self.size = 7
    # Basic Parameters
    self.atom_names = ["S", "O", "O", "O", "O", "H", "H"]
    self.col_sphere = pm.Collision_Sphere["H2SO4"]
    self.mass = pm.AMU["S"] + 4*pm.AMU["O"] + 2*pm.AMU["H"]
    self.position = np.array([[0,0,0.1534],
                              [0,1.2438, 0.8192 ],
                              [0,-1.2438, 0.8192],
                              [ 1.2193 ,0.0242,-0.8376],
                              [-1.2193,-0.0242,-0.8376],
                              [1.4709,-0.8641,-1.08],
                              [-1.4709,0.8641, -1.0800]])

    self.center = [0,0,0]
    self.com_frame = self.position.copy()
    self.previous_position = self.position.copy()
    self.type = "molecule"

elif self.name.upper() == "CUBANE":
    self.size = 16
    self.atom_names = ["C"
, "C", "C", "C", "C", "C", "C", "C", "H", "H", "H", "H", "H", "H", "H", "H"]
    self.col_sphere = pm.Collision_Sphere["H2SO4"]
    self.position = np.array([[0.7854,0.7854,0.7854],

```

```

[-0.7854,0.7854,0.7854],
[0.7854,0.7854,-0.7854],
[-0.7854,0.7854,-0.7854],
[0.7854,-0.7854,0.7854],
[-0.7854,-0.7854,0.7854],
[0.7854,-0.7854,-0.7854],
[-0.7854,-0.7854,-0.7854],
[1.4188,1.4188,1.4188],
[-1.4188,1.4188,1.4188],
[1.4188,1.4188,-1.4188],
[-1.4188,1.4188,-1.4188],
[1.4188,-1.4188,1.4188],
[-1.4188,-1.4188,1.4188],
[1.4188,-1.4188,-1.4188],
[-1.4188,-1.4188,-1.4188]]

self.mass = 8*(pm.AMU["C"] + pm.AMU["H"])
self.center = [0,0,0]
self.com_frame = self.position.copy()
self.previous_position = self.position.copy()
self.type = "molecule"

# Format of xyz file is assumed to be size, name, coordinates
elif self.name.upper() == "CUSTOM":
    with open(custom_path, "r") as f:
        f1 = f.readlines()
        self.size = int(f1[0].split()[0])
        self.name = f1[1].split()[0]

        self.atom_names = []
        self.mass = 0

        for i in range(2, self.size+2):
            Split = f1[i].split()
            atom_name = Split[0]
            self.atom_names.append(atom_name)
            self.mass += pm.AMU[atom_name]
            if i == 2:
                self.position = np.array([float(i) for i in
Split[1:]])

            else:
                self.position =
np.vstack([self.position,np.array([float(i) for i in Split[1:]])])

#
        self.position = self.position.transpose()
        self.center = [0,0,0]
        self.com_frame = self.position.copy()
        self.previous_position = self.position.copy()
        self.col_sphere = 6
        self.type = "molecule"

    else:
        self.type = None

def COM(self):
    R = self.position.copy()
    self.center = [0.0,0.0,0.0]

```

```

A = 0
for i in range(self.size):
    A += pm.AMU[self.atom_names[i]] * R[i] # m_i * r_i (Use AMU to
get m_i and self.position list comprehension to find r_i)
self.center = A/self.mass

def COM_FRAME(self):
    self.com_frame = 0
    R = deepcopy(self.position)
    R -= self.center * np.ones(R.shape)
    # for i in range(3):
    #     R[i] -= np.array(self.center)

    self.com_frame = R

def translate(self,delta):
    delArr = np.array(delta)* np.ones(self.position.shape)
    self.position += delArr

def xrotation(self, theta):

    theta = theta*math.pi/180 # converts theta to radians
    xvec = np.array([[1,0,0],
                     [0,np.cos(theta),np.sin(theta)],
                     [0,-np.sin(theta), np.cos(theta)]]])

    R = np.dot(self.com_frame, xvec) # Find new matrix in COM Frame
    Delta = R - self.com_frame      # Find changes in position from
rotation

    self.position += Delta           # Add difference to position
matrix

def yrotation(self, theta):
    theta = theta*math.pi/180.0 # converts theta to radians
    yvec = np.array([[np.cos(theta), 0 , np.sin(theta)],
                     [0,1,0],
                     [-np.sin(theta),0, np.cos(theta)]]])

    R = np.dot(self.com_frame, yvec) # Find new matrix in COM Frame
    Delta = R - self.com_frame      # Find changes in position from
rotation

    self.position += Delta           # Add difference to position
matrix

def zrotation(self, theta):
    theta = theta*math.pi/180 # converts theta Sto radians
    zvec = np.vstack([[np.cos(theta), -np.sin(theta), 0],
                      [np.sin(theta), np.cos(theta), 0],

```

```

                                [0,0,1]])

    R = np.dot(self.com_frame, zvec) # Find new matrix in COM Frame
    Delta = R - self.com_frame      # Find changes in position from
rotation
                                # Add difference to position
    self.position += Delta
matrix

def update(self):
    # self.previous_position = self.position.copy()
    self.COM()
    self.COM_FRAME()

def random_rotation(self, angle):
    x_angle = angle*np.random.uniform(low=-1.0,high=1.0)
    y_angle = angle*np.random.uniform(low=-1.0,high=1.0)
    z_angle = angle*np.random.uniform(low=-1.0,high=1.0)

    self.xrotation(x_angle)
    self.update()

    self.yrotation(y_angle)
    self.update()

    self.zrotation(z_angle)
    self.update()

    return self.position

# Copies previous position and then randomly moves the particle
def random_move(self, dist=1, theta=90):
    if self.size == 1:
        self.translate([dist*(2*np.random.random()-1), # Translate each
randomly
                                dist*(2*np.random.random()-1),
                                dist*(2*np.random.random()-1)])
        self.update()
    else:
        self.xrotation(theta*np.random.random())
        self.update()

        self.yrotation(theta*np.random.random())
        self.update()

        self.zrotation(theta*np.random.random())
        self.update()

        self.translate([dist*(2*np.random.random()-1), # Translate each
randomly
                                dist*(2*np.random.random()-1),
                                dist*(2*np.random.random()-1)])
        self.update()

```

```

class Create_System():

    def homogeneous(Npart, Mol_Class=Molecule, moleculeName=molName,
create_xyz=True, save_path=filePath):
        M=[]
        for i in range(Npart):
            M.append(Mol_Class(moleculeName)) # Initialize the molecule
            M[i].random_move(MAXT, MAXR) # Randomly move particle

            # This portion ensures molecules do not collide
            if i > 0: # Start checking after 1st particle
                j = 0 # Initialize j for while loop
                max_search = 100
                move_size = MAXT # Initial move_size is set by MAXT, can
later be increased
                m = 0
                while j < i:
                    if calc.object_distance(M[i],M[j]) >= 2*COL_DIST: # Check
if within collision distance
                        j += 1
                    else:
                        M[i].position = M[i].previous_position # Set equal to
previous position
                        M[i].update() # Update COM and coordinates with COM
at origin
                        M[i].random_move(move_size, MAXR) # Randomly move
again

                        j = 0
                        m += 1
                    if m > max_search:
                        move_size = INCREASE_MOVE*move_size
                        m = 0

            if create_xyz:
                data.XYZ.create(M, path=save_path)
            return M

    @classmethod
    def extract_species(cls,species_list):
        name_list = []
        number_objects = []
        for i, name in enumerate(species_list):
            name_list.append(name)
            number_objects.append(species_list[name])
        return name_list, number_objects

    @classmethod
    def heterogeneous(cls, species_list, Mol_Class=Molecule, Atom_Class=
Atom, create_xyz=True, save_path=filePath):
        classifications = []
        names, number_obj = cls.extract_species(species_list)
        M = []
        collision_distance = []

```

```

C = 0
for k,name in enumerate(names):
    for i in range(number_obj[k]):
        if Mol_Class(name).type == "molecule":
            classifications.append("molecule")
            M.append(Molecule(name))
        elif Atom_Class(name).type == "atom":
            classifications.append("atom")
            M.append(Atom(name))
        else:
            print("Error in Species List")
            break

    collision_distance.append(M[C].col_sphere)
    if M[C].size == 1:
        M[C].random_move(MAXT)
    else:
        M[C].random_move(MAXT, MAXR) # Randomly move particle

    if C > 0:
        j = 0
        max_search = MAX_SEARCH
        move_size = MAXT # Initial move_size is set by MAXT, can
later be increased
        m = 0
        while j < C:
            if calc.object_distance(M[C], M[j]) >=
collision_distance[j] + collision_distance[C]: # Check if within collision
distance
                j += 1
            else:
                M[C].position = M[C].previous_position # Set
equal to previous position
                M[C].update()
                M[C].random_move(move_size, MAXR) # Randomly move
again

                j = 0
                m += 1
            if m > max_search:
                move_size *= INCREASE_MOVE
                m = 0

        C += 1
    if create_xyz:
        data.XYZ.create(M, path=save_path, homogeneous=False)

    return M

species = {"NH4": 10,
           "Cl": 10}

```



## File: sim.py

"""This program contains a class Simulation() which has the function run() to perform a Monte Carlo simulation

The run function has multiple inputs, of which trials is required and all else will take default values.

The list of inputs and their descriptions are given in the sim\_inputs dictionary

This dictionary can be copied to a run file and the inputs changed to suit the user's needs

It is advisable to keep the variable save high so that the act of saving data too often does

not slow down the program. If the boolean "debug" is True the program will record all moves and

and create a log to help debug errors. This will slow down the program, however.

"""

```
import numpy as np
import initial_configuration as IC # Generates molecules and initial state
import parameters as pm          # Default parameters for simulation
from tqdm import tqdm
import data_handler as data      # Converts data into arrays
import calculations as calc      # Calculates values from data
from copy import deepcopy
import pandas as pd
import os
from copy import copy
```

#default simParameters for Homogeneous

```
N_PART = 2
DEFAULT_TRIALS = 1000
DEFAULT_T = 20
DEFAULT_NEQ = 0
DEFAULT_SAVE = 1000
DEFAULT_STEP = 2.0
DEFAULT_CHECK = 500
DEFAULT_SEED = 10
DEFAULT_LJ_CENTER = 0
DEFAULT_POTENTIAL = [calc.Lennard_Jones]
DEFAULT_MOL_NAME = "H2O"
DEFAULT_FILENAME = "H2O" # File save name
DEFAULT_SAVE_FOLDER = "Sim"
DEFAULT_OUTPATH_PATH =
os.path.join(os.getcwd(),DEFAULT_SAVE_FOLDER,DEFAULT_FILENAME)
DEFAULT_M_IN = []
```

```
class Simulation():
```

```

homogeneous = True

# Simulation Parameters
P_MT = 0.1          # Probability of translational mag-step
P_RT = 0.1          # # Probability of rotational mag-step
MAG_FACTOR = 10     # Factor by which to magnify translational step
MAG_THETA = 180     # Maximum theta for rotational magstep range
will be (-MAG_THETA, MAG_THETA)
COL_DIST = 1.5      # Collision distance [A]
N_PART = 2          # Number of particles by default (Used by
IC.Create_System to make M)

# Seeding random number generator (will skip if Anneal is True)
USE_SEED = True     # Use value for seed below if True, else
use system time to seed
SEED = DEFAULT_SEED # Seed input (using system time and input
needs to be integer)

# Import parameters from parameteers file
A = pm.TIP3P["A"]
B = pm.TIP3P["B"]
K_C = pm.K_C        # Coulomb constant
SIGMA = pm.TIP3P["SIGMA"]
EPSILON = pm.TIP3P["EPSILON"]
K_B = pm.K_B        # Boltzmann constant [kcal/mol-K]

# Simulation parameters
THETA = 90          # Max angle for non-magnified steps
SAVE = 2            # Number of iterations before saving
CHECK = 100         # Check acceptance every IRATIO cycles
BOX = 10

# You can enter Simulation inputs for function directly into parenthesis
or they can be unpacked from a dictionary using ** operator -->
Simulation(**simInputs)
# Showing dictionary here which is easier to copy and paste and edit in a
different file
# Values for N_part and trials are needed, while others take default
keyword arguments (see inside parentheses of run())
sim_inputs = {
    "trials": DEFAULT_TRIALS,          # Number of trials
    "temperature": DEFAULT_T,          # Temperature of run
    "n_eq": DEFAULT_NEQ,              # Equilibration trials
    "save": DEFAULT_SAVE,             # No. trials before
saving xyz files
    "step_size": DEFAULT_STEP,         # Initial step size
    "step_check": DEFAULT_CHECK,       # No. of trials before
correcting step size
    "outputPath": DEFAULT_OUTPATH_PATH, # Location of save file
    "seeding": True,                  # If True, will perform
random seed
    "debug": False,                   # If True, will print a
log of moves and energies to help debug the code
    "potential_list": DEFAULT_POTENTIAL, # List of potentials
(class names) to be used

```

```

        "diffCenter": True,                                # Use different center
point to calculate LJ
        "LJCenter": DEFAULT_LJ_CENTER                    # Atom to use in LJ
distance calculations
    }

M = DEFAULT_M_IN

@classmethod
def update_M(cls,M_in):
    cls.M = deepcopy(M_in)                                # This function updates M used by
the simulation. Can also be done manually

@classmethod
def run(cls, trials, temperature= DEFAULT_T,  n_eq= DEFAULT_NEQ, save=
DEFAULT_SAVE, step_size= DEFAULT_STEP,
        step_check= DEFAULT_CHECK, outputPath= DEFAULT_OUTPATH_PATH,
seeding= True, debug= False,
        potential_list= DEFAULT_POTENTIAL, diffCenter= True,  LJCenter=
DEFAULT_LJ_CENTER):

    UP = calc.User_Potential
    UP.class_list = potential_list  # Use potential with 3 terms
    potentials, potentials2 = UP.create_potentials()
    potential_names = UP.potential_names_list()

    # Seed using parameters above (seeding==False by default)
    if seeding:
        calc.seedRandom(cls.SEED, cls.USE_SEED)

    # Initialize arrays
    Energy_Store=[]
    Accepted_Energies=[]
    dataLog = []

    # Import molecular array and calculate initial energy
    M = cls.M
    N_part = len(M)
    data.XYZ.create(M, path=outputPath)
    U_arr = UP.calculate_potentials(M, potentials)

    # Store the initial values
    storage_dict= data.create_storage_dict(U_arr,potential_names)
    Accepted_Energies.append(storage_dict)
    Energy_Store.append(storage_dict)

    # Initialize acceptance counters (format [accept, reject, total])
    total_acceptance_array = np.array([0,0,0])
    loop_acceptance_array = np.array([0,0,0])
    count_iter = 0

    # k = trial iterator, i = molecule/atom iterator
    for k in tqdm(range(trials)):
        for i in range(N_part):

```

```

count_iter += 1

# Adjust step size
if k%step_check == 0 and i == 0:
    step_size, acceptance_list =
calc.update_step_size(step_size, temperature, loop_acceptance_array,
step_check)

# Find energies relative to molecule/atom i
UI_arr = UP.calculate_potentials2(M, i, potentials2)

# Save old position in case move is not accepted
M[i].previous_position = deepcopy(M[i].position) # Use
deepcopy to store previous position separately

# Translation condition for magwalk
if np.random.random() < cls.P_MT:
    vector = cls.MAG_FACTOR*step_size *
np.array([np.random.uniform(low=-1.0,high=1.0), np.random.uniform(low=-
1.0,high=1.0), np.random.uniform(low=-1.0,high=1.0) ])
    M[i].translate(vector)
    M[i].update()
else:
    vector = step_size * np.array([np.random.uniform(low=-
1.0,high=1.0), np.random.uniform(low=-1.0,high=1.0), np.random.uniform(low=-
1.0,high=1.0) ])
    M[i].translate(vector)
    M[i].update()

# Rotation condition (only applied to molecules, i.e. size>1)
if M[i].size > 1:
    if np.random.random() < cls.P_RT:
        M[i].random_rotation(cls.MAG_THETA)
    else:
        M[i].random_rotation(cls.THETA)

if k%save == 0:
    data.XYZ.append(M, path = outputPath, homogeneous =
cls.homogeneous)

Collide = False

# Loop to check if molecule collides with other molecules
for j in range(N_part):
    if j == i:
        pass
    elif calc.distance(M[i].center, M[j].center) <
2*cls.COL_DIST:
        Collide = True
        break
    else:
        pass

# REJECT MOVE BASED ON COLLISION

```

```

        if Collide:

            if k>= n_eq:
                storage_dict=
data.create_storage_dict(U_arr,potential_names)
                Energy_Store.append(storage_dict)

            # Update counters
            loop_acceptance_array += np.array([0,1,1])
            total_acceptance_array += np.array([0,1,1])

            if debug:
                log_dict = {"Trial":k, "Molecule Moved": i, "Random
Number":"N/A", "Boltzmann":"N/A", "Result": "Reject", "Why":"Collision",
"DeltaU":"N/A"}

                dataLog.append(**log_dict, **storage_dict))

            M[i].position = M[i].previous_position
            M[i].update()

            # If not colliding, perform energy test
            else:
                # Find test energy and deltaU using the molecule/atom
moved

                test_UI_arr = UP.calculate_potentials2(M, i, potentials2)
                delta_UI_arr = test_UI_arr - UI_arr
                test_U_arr = U_arr + delta_UI_arr

                rand_n = np.random.random()
                # Set Boltzmann factor to 0 for T=0 so no high energy
moves are accepted

                if temperature == 0:
                    bolz = 0
                else:
                    if isinstance(delta_UI_arr, np.ndarray):
                        bolz = np.exp(-delta_UI_arr[-
1]/cls.K_B/temperature)
                    else:
                        bolz = np.exp(-delta_UI_arr/cls.K_B/temperature)

                # Find deltaUI total depends on how long array of
potential energies is

                if isinstance(delta_UI_arr, np.ndarray):
                    delta_UI_tot = delta_UI_arr[-1]
                else:
                    delta_UI_tot = delta_UI_arr

                # Energy Decrease/No change = ACCEPT
                if np.sign(delta_UI_tot) <= 0:

                    if k >= n_eq:
                        if k%save == 0:
                            data.XYZ.append(M, path=outputPath,
homogeneous = cls.homogeneous)

```

```

        # Update counters
        loop_acceptance_array += np.array([1,0,1])
        total_acceptance_array += np.array([1,0,1])

        # Store new energies
        storage_dict=
data.create_storage_dict(test_U_arr,potential_names)
        Accepted_Energies.append(storage_dict)
        Energy_Store.append(storage_dict)

        if debug:
            log_dict = {"Trial":k, "Molecule Moved": i,
"Random Number":"N/A", "Boltzmann":"N/A", "Result": "Accept", "Why":"Energy
Decrease", "DeltaU":delta_UI_arr[-1]}
            dataLog.append(**log_dict, **storage_dict))

        # Update U when accepted
        U_arr = copy(test_U_arr)

        # PASSES BOLTZMANN TEST = ACCEPT
        elif rand_n <= bolz:

            if k >= n_eq:
                if k%save == 0:
                    data.XYZ.append(M, path=outputPath,
homogeneous = cls.homogeneous)

                # Update counters
                loop_acceptance_array += np.array([1,0,1])
                total_acceptance_array += np.array([1,0,1])

                # Store new energies
                storage_dict=
data.create_storage_dict(test_U_arr,potential_names)
                Accepted_Energies.append(storage_dict)
                Energy_Store.append(storage_dict)

                if debug:
                    log_dict = {"Trial":k, "Molecule Moved": i,
"Random Number":rand_n, "Boltzmann":bolz, "Result": "Accept", "Why":"Boltzmann
Test Pass", "DeltaU":delta_UI_arr[-1]}
                    dataLog.append(**log_dict, **storage_dict))

                # Update U when accepted
                U_arr = copy(test_U_arr)

        # Fails Boltzmann Test = REJECT
        else:

            if k >= n_eq:
                if k%save == 0:
                    data.XYZ.append(M, path=outputPath,
homogeneous = cls.homogeneous)

```

```

        # Update counters
        loop_acceptance_array += np.array([0,1,1])
        total_acceptance_array += np.array([0,1,1])

        # Store new energies
        rej_storage_dict=
data.create_storage_dict(test_U_arr,potential_names)
        Energy_Store.append(rej_storage_dict)

        if debug:
            storage_dict1 =
data.create_storage_dict(U_arr,potential_names)
            log_dict = {"Trial":k, "Molecule Moved": i,
"Random Number":rand_n, "Boltzmann":bolz, "Result": "Reject", "Why":"Boltzmann
Test Fail", "DeltaU":delta_UI_arr[-1]}
            dataLog.append(**log_dict, **storage_dict1))

        M[i].position = M[i].previous_position
        M[i].update()

        # Allow variable no. of outputs using dictionary, have more outputs
when debugging
        if debug:
            Outputs = {"Energy_Store":pd.DataFrame(Energy_Store),
"Accepted_Energies": pd.DataFrame(Accepted_Energies),
"Acceptance": total_acceptance_array,
"M_fin": M,
"step_size": step_size,
"dataLog": pd.DataFrame(dataLog[1:])}
        else:
            Outputs = {"Energy_Store":pd.DataFrame(Energy_Store),
"Accepted_Energies": pd.DataFrame(Accepted_Energies),
"Acceptance": total_acceptance_array,
"M_fin": M,
"step_size": step_size}

        return Outputs

# Simulation.M = IC.Create_System.homogeneous(N_PART, moleculeName="H2O",
Mol_Class=IC.Molecule, create_xyz=False)
# Outputs = Simulation.run(**Simulation.sim_inputs)

```

## File: calculations.py

```
"""
This Program contains functions used for calculations in both the setup and
during
the course of a simulation. Most of the functions in this library require the
input of
a molecular system, M, to iterate on.
"""
import numpy as np
import parameters as pm
import time

# Takes any two position vectors and finds distance between them
def distance(v1,v2):
    diffv = v1-v2
    D = 0
    for iter1,term in enumerate(diffv):
        D += term**2
    D = np.sqrt(D)
    return D

# Finds distance between molecules using COM
def object_distance(M1,M2):
    return distance(M1.center, M2.center)

def seedRandom(inputSeed,condition):
    if condition:
        np.random.seed(inputSeed)
    else:
        np.random.seed(int(time.time()))
    return None

def system_center(M):
    array_sum = np.array([0.0, 0.0, 0.0])
    total_mass = 0.0

    for obj in M:
        array_sum += obj.mass * obj.center
        total_mass += obj.mass

    return array_sum/total_mass

def update_step_size(t_step, temp, acceptance_list, step_check_num):
    """ Takes inputs step-size, temperature, acceptance_list, step_check"""
    accepted,rejected,total = acceptance_list
    if total == 0:
        pass
    else:
        acceptance = accepted/total

        if acceptance <= 0.5:
```



```

        if temp == 0:
            t_step = 0.01
        elif acceptance <= 0.01:
            if t_step < 0.001:
                pass
            else:
                t_step *= 0.1
        else:
            t_step *= 0.8

        acceptance_list = np.array([0,0,0])

    elif acceptance > 0.5:
        if acceptance >= 0.7:
            t_step *= 5
        else:
            t_step *= 1.2
        acceptance_list = ([0,0,0])

    return t_step, acceptance_list

#####
# Potential Classes
#####

class Lennard_Jones():
    """ Calculates Lennard Jones 6-12 potential of the form  $U = 4\epsilon((\sigma/r)^{12} - (\sigma/r)^6)$  for a system of molecules"""

    # Class variables used as parameters by class methods using syntax
    cls.sigma, etc.
    sigma = pm.SIGMA # Parameter sigma for LJ 6-12
    epsilon = pm.EPSILON # Parameter epsilon for LJ 6-12
    calculate_atom_by_atom = False # If True, calculate potential using all
the atoms, not molecules only
    diff_center = False # If molecule only: choose if distance r
calculated using default COM of molecule or particular atom
    center_atom = 0 # if particular atom, give atom position
in atom_names list
    print_string = "U_LJ" # Used when printing outputs

#####
# Potential 1 (Entire system)
#####

# class method decorator allows function to use class variables ^^
@classmethod
def potential(cls, M): # Will be in kcal/mol
    En = 0

    if cls.calculate_atom_by_atom == False:
        for iter1, obj1 in enumerate(M[:-1]):
            for iter2, obj2 in enumerate(M[iter1+1:], start = iter1+1):
                if cls.diff_center:

```

```

        C1 = obj1.position[cls.center_atom]
        C2 = obj2.position[cls.center_atom]
    else:
        C1 = obj1.center
        C2 = obj2.center

    r = distance(C1,C2) # Currently takes first molecule as
center

    r6, s6 = r**6, cls.sigma**6
    r12, s12 = r6**2, s6**2
    En += s12/r12 - s6/r6

    En *= 4 * cls.epsilon

else:
    for iter1, obj1 in enumerate(M[:-1]):
        for iter2,obj2 in enumerate(M[iter1+1:], start = iter1+1):
            for a in range(obj1.size):
                for b in range(obj2.size):

                    if cls.epsilon[a][b]==0 or cls.sigma[a][b]==0:
                        pass

                    else:
                        dist =
distance(obj1.position[a],obj2.position[b])
                        r6, s6 = dist**6, cls.sigma[a][b]**6
                        r12, s12 = r6**2, s6**2
                        En += 4*cls.epsilon[a][b]*(s12/r12 - s6/r6)

    return En

#####
# Potential 2 (Relative to one molecule only --> fast computations)
#####

# Used to compute deltaU quickly if object i was moved (computes U
relative to i only)
@classmethod # Class method decorator
def potential2(cls, M, i): # Will be in kcal/mol
    En = 0
    if cls.calculate_atom_by_atom == False:

        for iter1, obj in enumerate(M):
            if i == iter1:
                pass

            else:
                if cls.diff_center: # Pick atom to calculate distances
                    C1 = M[i].position[cls.center_atom]
                    C2 = obj.position[cls.center_atom]
                else:
                    C1 = M[i].center

```

```

        C2 = obj.center

        dist = distance(C1,C2) # Currently takes first molecule
as center

        r6, s6 = dist**6, cls.sigma**6
        r12, s12 = r6**2, s6**2
        En += s12/r12 - s6/r6

    En *= 4 * cls.epsilon

else:

    for iter1,obj in enumerate(M):
        if i == iter1:
            pass

        else:
            for a in range(M[i].size):
                for b in range(obj.size):
                    if cls.epsilon[a][b] == 0 or cls.sigma[a][b] ==
0:

                        pass
                    else:
                        dist = distance(M[i].position[a],
obj.position[b])

                        r6, s6 = dist**6, cls.sigma[a][b]**6
                        r12, s12 = r6**2, s6**2
                        En += 4*cls.epsilon[a][b]*(s12/r12 - s6/r6)

    return En

class Coulombic():
    """ Coulombic potential: calculates potential in the form  $U = k_c \cdot q_i \cdot q_j / r_{ij}$  for all atoms in a system"""

    # Class variables used as parameters by class methods using cls.k_c, etc.
    k_c = pm.K_C # Coulomb constant (kcal/mol)
    print_string = "U_C" # Used when printing outputs

    #####
    # Potential 1 (Entire System)
    #####

    # class method decorator allows function to use class variables ^^
    @classmethod
    def potential(cls,M):
        En = 0
        for iter1, obj1 in enumerate(M[:-1]):
            for iter2,obj2 in enumerate(M[iter1+1:],start=iter1+1):
                EIJ = 0
                for a in range(obj1.size):
                    for b in range(obj2.size):
                        dist = distance(obj1.position[a],obj2.position[b])

```

```

        EIJ += obj1.charges[a] * obj2.charges[b] / dist

        EIJ *= cls.k_c
        En += EIJ

    return En

#####
# Potential 2 (Relative to one molecule only --> fast computations)
#####

@classmethod
def potential2(cls, M, i):
    En = 0
    for iter1,obj in enumerate(M):
        EIJ = 0
        if iter1 == i:
            pass
        else:
            for a in range(M[i].size):
                for b in range(obj.size):
                    dist = distance(M[i].position[a], obj.position[b])
                    EIJ += M[i].charges[a] * obj.charges[b] / dist

            EIJ *= cls.k_c # Coulombic Term
            En += EIJ
    return En

class N_Well():
    """ Constraint potential in the form  $U = r^{(2n)}$  to a reference point"""
    # Function parameters
    n = 2 # input exponent for potential
    box = 10 # Constraint box size
    ref = np.zeros(3) # Reference point for distance (can be
    origin, system COM, etc.)
    print_string = "U_Constr" # Used when printing outputs

    #####
    # Potential 1
    #####
    @classmethod
    def potential(cls, M):
        En = 0
        for obj in M:
            En += (distance(cls.ref, obj.center) / cls.box)**(2*cls.n)
        return En

    #####
    # Potential 2
    #####
    @classmethod
    def potential2(cls, M, i):
        En = (distance(cls.ref, M[i].center) / cls.box)**(2*cls.n)
        return En

```

```

#####
# Description of inputs (not used in calculations anymore)
#####

inputs = {"M":[],                                # Leave blank for input, program will
automatically update                             # Box size for simulation
          "box": 10}

inputs2 = {"M":[],                                # Leave blank for input, program
will automatically update                         # Molecule number (loop will update
          "i": 0,                                # Box size for simulation
this on its own)
          "box": 10}

class TIP3P():
    """ Calculate potential via equation:  $U = k_c \cdot q_i \cdot q_j / r + A / r^{12} - B / r^6$  """
    k_c = pm.K_C                                # Coulomb constant
    A = pm.TIP3P["A"]                           # Parameter A for TIP3P
    B = pm.TIP3P["B"]                           # Parameter B for TIP3P potential
    ref = np.zeros(3)

    @classmethod
    def compute(cls, M): # Will be in kcal/mol
        E = 0
        for i in range(len(M)-1):
            for j in range(i+1, len(M)):

                # Inter Oxygen distance
                r_oo = distance(M[i].Ox, M[j].Ox)
                r6 = r_oo ** 6
                r12 = r6 ** 2

                # Initialize variable for charge potential energy
                EIJ = 0
                for a in range(3):
                    for b in range(3):
                        EIJ += M[i].charges[a] * M[j].charges[b] /
distance(M[i].position[a], M[j].position[b])

                EIJ = cls.k_c * EIJ # Coulombic Term
                EIJ = EIJ + cls.A/r12 - cls.B/r6 # Pair Potential Energy
between
                E += EIJ
            E += distance(np.zeros(3), M[i].center) # Add Pair Potential to
Total Potential Energy
        E += distance(np.zeros(3), M[-1].center)
        return E

    @classmethod
    def compute2(cls, M, i):
        En = 0
        for j in range(len(M)):
            if j == i:
                pass

```

```

        else:
            EIJ = 0

            r_oo = distance(M[i].center, M[j].center)
            r6, s6 = r_oo**6, cls.sigma**6
            r12, s12 = r6**2, s6**2
            r6 = r_oo ** 6
            r12 = r6 ** 2
            for a in range(3):
                for b in range(3):
                    EIJ += M[i].charges[a] * M[j].charges[b] /
distance(M[i].position[a],M[j].position[b])
            EIJ = cls.k_c * EIJ
            EIJ += 4*cls.epsilon*(s12/r12 - s6/r6)
            En += EIJ
        En += distance(np.zeros(3),M[i].center)
    return En

class User_Potential():
    """ Creates custom potential based on user input: By default calculates
    using LJ, Coulombic, N_Well"""
    class_list = [Lennard_Jones, Coulombic, N_Well]
    sum_at_end = True

    @classmethod
    def create_potentials(cls):
        f1 = []
        f2 = []

        for potential_class in cls.class_list:
            f1.append(potential_class.potential)
            f2.append(potential_class.potential2)
        return f1, f2

    @classmethod
    def calculate_potentials(cls, M, funcs):
        if len(funcs) > 1:
            U_list = []
            for func in funcs:
                result = func(M)
                if type(result) == list or type(result) == np.ndarray:
                    U_list = [*U_list, *result]
                else:
                    U_list.append(func(M))
            if cls.sum_at_end == True:
                U_list.append(sum(U_list))
        else:
            U_list = funcs[0](M)
        return np.array(U_list)

    @classmethod
    def calculate_potentials2(cls, M, i, funcs):
        if len(funcs) > 1:
            U_list = []
            for func in funcs:
                result = func(M, i)

```

```

        if type(result) == list or type(result) == np.ndarray:
            U_list = [*U_list, *result]
        else:
            U_list.append(func(M,i))
    if cls.sum_at_end == True:
        U_list.append(sum(U_list))
else:
    U_list = funcs[0](M,i)
return np.array(U_list)

@classmethod
def potential_names_list(cls):
    if len(cls.class_list) > 1:
        names_list = []
        for pot in cls.class_list:
            if type(pot.print_string) == list:
                names_list = [*names_list, *pot.print_string]
            else:
                names_list.append(pot.print_string)

        if cls.sum_at_end == True:
            names_list.append("U_tot")
    else:
        names_list = cls.class_list[0].print_string
    return names_list

class JBRQT():
    A, B, C, D, Q = pm.AmmCl["A"], pm.AmmCl["B"], pm.AmmCl["C"],
pm.AmmCl["D"], pm.AmmCl["Q"]
    k_c = pm.K_C
    print_string = ["U_buck", "U_coul", "ULJ6", "ULJ12", "U_tot"]
    test_val = 10

#####
# Potential 1
#####

@classmethod
def potential(cls, M):
    U_buck = 0.0
    U_coul = 0.0
    ULJ6 = 0.0
    ULJ12 = 0.0
    for iter1, mol1 in enumerate(M[:-1]): # Using iter1 and iter2 so that
i in loop is not changed
        for iter2, mol2 in enumerate(M[iter1+1:],start=iter1+1):
            for a in range(mol1.size):
                for b in range(mol2.size):
                    if mol1.size == 1:
                        if mol2.size == 1:
                            dist = distance(mol1.position, mol2.position)
                        else:
                            dist = distance(mol1.position,
mol2.position[b])
                    elif mol2.size == 1:

```

```

        dist = distance(mol1.position[a], mol2.position)
    else:
        dist = distance(mol1.position[a],
mol2.position[b])

    r6 = dist**6
    r12 = r6**2
    k1 = mol1.atom_names[a] # atom1 name or key
    k2 = mol2.atom_names[b] # atom2 name or key
    U_buck += cls.A[k1][k2] * np.exp(-cls.B[k1][k2]*dist)
    U_coul += cls.k_c*cls.Q[k1][k2] / dist
    ULJ6 += -cls.C[k1][k2]/r6
    ULJ12 += cls.D[k1][k2]/r12

    U_arr = [U_buck, U_coul, ULJ6, ULJ12] # Put all energies into array
    U_arr.append(sum(U_arr)) # Add total energy to array
    return np.array(U_arr)

@classmethod
def potential2(cls, M, i):
    U_buck = 0.0
    U_coul = 0.0
    ULJ6 = 0.0
    ULJ12 = 0.0
    for iter1, obj in enumerate(M): # Using iter1 and iter2 so that i in
loop is not changed
        if iter1 == i:
            pass
        else:
            for a in range(M[i].size):
                for b in range(obj.size):
                    if M[i].size == 1:
                        if obj.size == 1:
                            dist = distance(M[i].position, obj.position)
                        else:
                            dist = distance(M[i].position,
obj.position[b])

                    elif obj.size == 1:
                        dist = distance(M[i].position[a], obj.position)
                    else:
                        dist = distance(M[i].position[a],
obj.position[b])

                r6 = dist**6
                r12 = r6**2
                k1 = M[i].atom_names[a] # atom1 name or key
                k2 = obj.atom_names[b] # atom2 name or key

                U_buck += cls.A[k1][k2] * np.exp(-cls.B[k1][k2]*
dist)
                U_coul += cls.k_c*cls.Q[k1][k2] / dist

```



```

        ULJ6 += -cls.C[k1][k2]/r6
        ULJ12 += cls.D[k1][k2]/r12

    U_arr = [U_buck, U_coul, ULJ6, ULJ12] # Put all energies into array
    U_arr.append(sum(U_arr)) # Add total energy to array
    return np.array(U_arr)

```

## File: anneal.py

```
""" This file performs simulated annealing calculations on the Simulation
function. The annealing schedule is currently saw tooth
with N_PT representing the number of temperatures per tooth and N_T as the
number of teeth. """

import data_handler as data
import numpy as np
import matplotlib.pyplot as plt
import os
import calculations as calc
import pandas as pd
import tip3p_run as t3
import initial_configuration as IC
import parameters as pm
from sim import Simulation

calc.seedRandom(10, True)
# Folder names
SAVE_FOLDER = "revisedTIP3P" # <---Folder name that may need to be changed
(Currently saved inside Sim)
MAIN_FOLDER = "Sim" # Usually the folder where other sims are already saved
DATA_FOLDER = "Data"
XYZ_FOLDER = "XYZ"
PLOT_FOLDER = "Plots"
# Create file path names (CWD = current working directory)
TRIAL_PATH = os.path.join(os.getcwd(), MAIN_FOLDER, SAVE_FOLDER) #
CWD/mainFolder/saveFolder
DATA_PATH = os.path.join(TRIAL_PATH, DATA_FOLDER) #
CWD/mainFolder/saveFolder/dataFolder
XYZ_PATH = os.path.join(TRIAL_PATH, XYZ_FOLDER) #
CWD/mainFolder/saveFolder/XYZFolder
PLOT_PATH = os.path.join(TRIAL_PATH, PLOT_FOLDER) #
CWD/mainFolder/saveFolder/plotFolder

paths = [TRIAL_PATH, DATA_PATH, XYZ_PATH, PLOT_PATH]
data.createPaths(paths) # Creates paths, or ignores if they already exist

RUNS = 1000
NEQ = 1
SAVE = 500
CHECK = 300
DEFAULT_STEP = 2.0
showFigs = False
Tmax= 1000 # K
N_PT = 5# Temperatures per tooth
N_T = 2# Number of teeth
alpha = 0.7 # Factor by which max T decreases in subsequent teeth
N_PART = 2
SAVE= 100

Water_Mol= t3.Water_Mol
```

```

class_list = [calc.Lennard_Jones, calc.Coulombic, calc.N_Well]
calc.Lennard_Jones.sigma = pm.TIP3P["SIGMA"]
calc.Lennard_Jones.epsilon = pm.TIP3P["EPSILON"]
calc.N_Well.n = 10
Simulation.M = IC.Create_System.homogeneous(N_PART, moleculeName="H2O",
Mol_Class=Water_Mol, create_xyz=False)

```

```

class Anneal():

```

```

    sim_inputs = {
        "trials": RUNS,                # Number of trials
        "n_eq": NEQ,                   # Equilibration trials
        "temperature": 1000,           # Temperature of run
        "save": SAVE,                  # No. trials before saving xyz
    files
        "step_size": DEFAULT_STEP,     # Initial step size
        "step_check": CHECK,           # No. of trials before correcting
    step size
        "outputPath": TRIAL_PATH,      # Location of save file
        "seeding": True,               # If True, will perform random seed
        "debug": False,                # If True, will print a log of
    moves and energies to help debug the code
        "potential_list": class_list,
        "diffCenter": True,            # Use different center point to
    calculate LJ
        "LJCenter": 0                  # Atom to use in LJ distance
    calculations
    }

```

```

    @classmethod

```

```

    def run(cls):

```

```

        dataStore = []
        temperatures = []
        EAcceptedCont = []
        T1 = Tmax

        for P in range(1, N_T+1):
            T = T1
            delT = T1/(N_PT - 1)

            for N in range(N_PT):
                print("\n\nP=", P, "N=", N, "T=", T)
                PNTString = "P" + str(P) + "_N" + str(N) + "_T" + str(T) #
                #Set up xyz filename
                xyz_file = os.path.join(XYZ_PATH, PNTString)

                if P == 1 and N == 0:
                    cls.sim_inputs["temperature"] = T
                    cls.sim_inputs["outputPath"] = xyz_file

                    # RUN SIMULATION

```

```

Outputs = Simulation.run(**cls.sim_inputs)
M_fin = Outputs["M_fin"]

U_accepted = Outputs["Accepted_Energies"].iloc[:,-1]
all_accepted_U = U_accepted # Save to a running tally
print("\n", U_accepted)
print(Outputs["Accepted_Energies"])

# Create Movie file for storing final position

data.XYZ.create(M_fin,path=os.path.join(XYZ_PATH,"Movie"))

else:
    if np.round(T) == 0:
        T = 0

    # Simulation.M = deepcopy(M_fin )
    cls.sim_inputs["temperature"] = T
    cls.sim_inputs["outputPath"] = xyz_file

    # RUN SIMULATION
    Outputs = Simulation.run(**cls.sim_inputs)
    M_fin = Outputs["M_fin"]

    U_accepted = Outputs["Accepted_Energies"].iloc[:,-1]
    all_accepted_U = pd.concat([all_accepted_U , U_accepted])

    # Create Movie file for storing final position

data.XYZ.append(M_fin,path=os.path.join(XYZ_PATH,"Movie"))

    if T == 0:
        data.XYZ.create(M_fin, path=
os.path.join(XYZ_PATH,"Final_Structure_P={}".format(P)))
    else:
        T -= delT

    # Save all and accepted energies: LJ, Coulombic, Constraint,
Total

Outputs["Energy_Store"].to_csv(os.path.join(DATA_PATH,"Energy_Store_"+PNTString+".csv"))

Outputs["Accepted_Energies"].to_csv(os.path.join(DATA_PATH,"Accepted_Energies_
_"+PNTString+".csv"))

    dataStore.append([T, M_fin, Outputs["step_size"]])
    temperatures.append(T)

T1 = alpha*T1

    output_dict = {"temperatures": temperatures, "data_store": dataStore,
"M_fin": M_fin, "all_accepted_U": all_accepted_U }

```

```
    return output_dict

results = Anneal.run()
```

## File: data\_handler.py

```
""" This file collects data from molecular arrays and stores them into either
xyz files to dataframes
```

```
    This file also contains function used in displaying potential energies
and other data formatting needs"""
```

```
import pandas as pd
import numpy as np
import parameters as pm
import os
import matplotlib.pyplot as plt
```

```
defaultSavePath = os.path.join(os.getcwd(), "Sim", "testPrint")
```

```
class XYZ():
```

```
    """ Used to create and append xyz files """
```

```
    def create(M, path=defaultSavePath, homogeneous = True):
```

```
        if homogeneous:
```

```
            name = M[0].name
```

```
            size = sum([mol.size for mol in M])
```

```
            f = open(path+".xyz", 'w+')
```

```
            f.write(str(int(size))+"\n")
```

```
            f.write(str(name)+"\n")
```

```
        else:
```

```
            size = sum([mol.size for mol in M]) # sum up the sizes to get
```

```
total # atoms
```

```
            comment = "Mixed Species Sim"
```

```
            f = open(path+".xyz", 'w+')
```

```
            f.write(str(int(size))+"\n")
```

```
            f.write(comment+"\n")
```

```
        for obj in M:
```

```
            if obj.size == 1:
```

```
                f.write("{} \t {:.6f} \t {:.6f} \t {:.6f}
```

```
\n".format(obj.atom_names[0], obj.position[0], obj.position[1],
obj.position[2]))
```

```
            else:
```

```
                for j, coords in enumerate(obj.position):
```

```
                    f.write("{} \t {:.6f} \t {:.6f} \t {:.6f}
```

```
\n".format(obj.atom_names[j], coords[0], coords[1], coords[2]))
```

```
            f.close()
```

```
# Appends already existing xyz file. Default filename is testprint.
```

```
def append(M, path=defaultSavePath, homogeneous = True):
```

```
    if homogeneous:
```

```
        name = M[0].name
```

```
        size = sum([mol.size for mol in M])
```

```
        f = open(path+".xyz", "a")
```

```
        f.write(str(int(size))+"\n")
```

```
        f.write(str(name)+"\n")
```

```
    else:
```

```
        size = sum([mol.size for mol in M]) # sum up the sizes to get
```

```
total # atoms
```

```

comment = "Mixed Species System"
f = open(path+".xyz", "a")
f.write(str(int(size))+"\n")
f.write(comment+"\n")

for obj in M:
    if obj.size == 1:
        f.write("{} \t {:.6f} \t {:.6f} \t {:.6f}
\n".format(obj.atom_names[0], obj.position[0], obj.position[1],
obj.position[2]))
    else:
        for iter1,coords in enumerate(obj.position):
            f.write("{} \t {:.6f} \t {:.6f} \t {:.6f}
\n".format(obj.atom_names[iter1], coords[0], coords[1], coords[2]))

f.close()

def create_df(M):
    """ Creates a dataframe for a given system """
    R = []
    for i,obj in enumerate(M):
        if i == 0:
            if obj.type == "molecule":
                R = pd.DataFrame(obj.position)
                T = pd.DataFrame(obj.atom_names)
                R = pd.concat([T, R],axis=1,ignore_index=True)
            else:
                B = list(obj.position)
                B.insert(0,obj.name)
                R = pd.DataFrame(B).T

        else:
            if obj.type == "molecule":
                B = pd.DataFrame(M[i].position)
                T = pd.DataFrame(M[i].atom_names)
                B = pd.concat([T, B],axis=1,ignore_index=True)
                R = pd.concat([R,B], ignore_index = True)
            else:
                B = list(obj.position)
                B.insert(0,obj.name)
                B = pd.DataFrame(B).T
                R = pd.concat([R, B], ignore_index = True)

    R.columns = ["Atom", "X", "Y", "Z"]
    R.index = range(1,len(R)+1)
    return R

def create_dictionary(loc ="Spec_List.txt"):
    """ Creates dictionary from a text file: notation and example given in
    Spec_List.txt"""
    with open(loc,mode='r') as f:
        fl = f.readlines()[1:] # 1st line (comment) omitted

```

```

        x = [line.replace(",","").split() for line in f1] # List
        comprehension to store each line within list
        x = list(filter(None,x)) # Removes blank lines
        dictionary = {element[0]: int(element[1]) for element in x} #
        Dictionary comprehension converts list to dictionary
        return dictionary

def createPaths(PATHS):
    """ Checks if paths exist or creates them if needed for saving files"""
    for path in PATHS:
        try:
            os.mkdir(path)
        except:
            pass
    return None

def create_storage_dict(U_list, names_list):
    try:
        len(U_list)
        return {name: u for (name, u) in zip(names_list, U_list)}
    except:
        return {names_list: U_list}

def names(M):
    names_list = []
    for obj in M:
        names_list.append(obj.name)
    return names_list

def functionNames(funcs):
    names = []
    for func in funcs:
        names.append(func.__name__)
    return names

```



## File: tip3p\_run.py

```
""" TIP3P Run file with special class added for model 3-site model """

import numpy as np
import initial_configuration as IC # Generates molecules and initial state
import parameters as pm          # Default parameters for simulation
import calculations as calc      # Calculates values from data
from copy import deepcopy
from sim import Simulation
import os

# Sim save location (Don't need to write xyz, function already adds it)
MOL_NAME = "H2O" # This is fed to molecule class or custom class to create
instances
SAVE_FOLDER = "Sim" # Save folder in current directory
FILENAME = "TIP3P" # File save name
FILE_PATH = os.path.join(os.getcwd(), SAVE_FOLDER, FILENAME) # Save Location
will be CurrentDirectory/saveFolder/fileName
SAVE = 1
CHECK = 1000
N_PART = 2

# Import TIP3P paramters from paramters file
SIGMA = pm.TIP3P["SIGMA"]
EPSILON = pm.TIP3P["EPSILON"]

class_list = [calc.Lennard_Jones, calc.Coulombic, calc.N_Well] # Use 3
potentials
calc.Lennard_Jones.sigma = SIGMA # Modify sigma and epsilon
to match TIP3P
calc.Lennard_Jones.epsilon = EPSILON
calc.Lennard_Jones.diff_center = True # Don't use COM to calculate
r,
calc.Lennard_Jones.center_atom = 0 # Use oxygen to calculate
distance for LJ 6-12
calc.N_Well.n = 2 # Use quadwell potential
(r/C)^2n = (r/C)^4

### MODIFIED MOLECULE CLASS FOR SIMULATION
# Create modified molecule class
class Water_Mol(IC.Molecule):
    # Modify __init__() to include charges and specify oxygen and hydrogen
    positions separately
    def __init__(self, name = "H2O"):
        super().__init__("H2O")
        self.name = name

    # Add partial charges based on TIP3P model
    self.charges = [pm.TIP3P["Q_O"], pm.TIP3P["Q_H"], pm.TIP3P["Q_H"]] #
Arrangement is [O H H]

    # Test position for particle
```

```

        self.previous_position = deepcopy(self.position)

    def update(self):
        super().update()

    def random_rotation(self, angle):
        x_angle = angle*np.random.uniform(low=-1.0,high=1.0)
        y_angle = angle*np.random.uniform(low=-1.0,high=1.0)
        z_angle = angle*np.random.uniform(low=-1.0,high=1.0)

        self.xrotation(x_angle)
        self.update()

        self.yrotation(y_angle)
        self.update()

        self.zrotation(z_angle)
        self.update()

    return self.position
### END MODIFIED CLASS

# Create M
Simulation.M = IC.Create_System.homogeneous(N_PART, moleculeName="H2O",
Mol_Class=Water_Mol, create_xyz=False)

# You can enter Simulation inputs for function directly into parenthesis or
they can be unpacked from a dictionary using ** operator -->
Simulation(**simInputs)
# Showing dictionary here to help to explain the individual inputs more
easily
simInputs = {
    "trials": 100,                # Number of trials
    "n_eq":0,                    # Equilibration trials
    "temperature":100,           # Temperature of run
    "save": SAVE,                # No. trials before saving xyz

files
    "step_size": 0.5,            # Initial step size
    "step_check": 100,          # No. of trials before correcting

step size
    "outputPath":FILE_PATH,      # Location of save file
    "seeding": True,             # If True, will seed using given

number
    "debug": False,             # If True, will print a log of

moves and energies to help debug the code
    "potential_list": class_list, # List of potential classes to be

used
    "diffCenter": True,         # Use different center point to

calculate LJ
    "LJCenter": 0               # Atom to use in LJ distance

calculations
}

# Run and store outputs to a dictionary variable

```

```
Outputs = Simulation.run(**simInputs)
M = Outputs["M_fin"]

list_of_outputs = Outputs.keys()
```

## File: NH4Cl\_run.py

```
""" NH4Cl Run file: Uses JBRQT potential and a constraint well"""
import numpy as np
import initial_configuration as IC # Generates molecules and initial state
import calculations as calc      # Calculates values from data
from sim import Simulation
import os

np.set_printoptions(suppress=True)

# Simulation save location (Don't need to write xyz, data_handler already
# adds it)
SAVE_FOLDER = "Sim" # Save folder in current directory
FILENAME = "NH4CL" # File save name
FILE_PATH = os.path.join(os.getcwd(), SAVE_FOLDER, FILENAME) # Save Location
# will be CurrentDirectory/saveFolder/fileName
SAVE = 10
CHECK = 1000

# Species List, currently using 2 NH4 and 2 Cl ions
speciesList = {"NH4": 4,
               "Cl": 4}

# Create system and modify energy parameters
Simulation.M = IC.Create_System.heterogeneous(speciesList, create_xyz =
False)
class_list= [calc.JBRQT, calc.N_Well] # Create potentials
# list to use
calc.N_Well.n = 10 # Change constraint n
# in (r/box)^2n

sim_inputs = {
    "trials": 10000, # Number of trials
    "n_eq": 0, # Equilibration trials
    "temperature": 50, # Temperature of run
    "save": SAVE, # No. trials before saving xyz
    "files": {
        "step_size": 2.0, # Initial step size
        "step_check": 200, # No. of trials before correcting
    },
    "step size": {
        "outputPath": FILE_PATH, # Location of save file
        "seeding": True, # If True, will seed using given
    },
    "seed": {
        "debug": True, # If True, will print a log of
    },
    "moves and energies to help debug the code": {
        "potential_list": class_list, # List of potentials (class names)
    },
    "to be used": {
        "diffCenter": False, # Use different center point to
    },
    "calculate LJ": {
        "LJCenter": 0 # Atom to use in LJ distance
    },
    "calculations": {}
}

# Run system and store outputs
```

```
Outputs = Simulation.run(**sim_inputs)
list_of_outputs = Outputs.keys()
```

# File:NH4Cl\_anneal.py

```
import data_handler as data
import numpy as np
import os
import calculations as calc
import pandas as pd
import initial_configuration as IC
import parameters as pm
from sim import Simulation

calc.seedRandom(10,True)
# Folder names
SAVE_FOLDER = "NH4Cl_Anneal_last" # <---Folder name that may need to be
changed (Currently saved inside Sim)
MAIN_FOLDER = "Sim" # Usually the folder where other sims are already saved
DATA_FOLDER = "Data"
XYZ_FOLDER = "XYZ"
PLOT_FOLDER = "Plots"
# Create file path names (CWD = current working directory)
TRIAL_PATH = os.path.join(os.getcwd(), MAIN_FOLDER, SAVE_FOLDER) #
CWD/mainFolder/saveFolder
DATA_PATH = os.path.join(TRIAL_PATH, DATA_FOLDER) #
CWD/mainFolder/saveFolder/dataFolder
XYZ_PATH = os.path.join(TRIAL_PATH, XYZ_FOLDER) #
CWD/mainFolder/saveFolder/XYZFolder
PLOT_PATH = os.path.join(TRIAL_PATH, PLOT_FOLDER) #
CWD/mainFolder/saveFolder/plotFolder

paths = [TRIAL_PATH, DATA_PATH, XYZ_PATH, PLOT_PATH]
data.createPaths(paths) # Creates paths, or ignores if they already exist

RUNS = 60000
NEQ = 9000
SAVE = 1000
CHECK = 500
DEFAULT_STEP = 2.0
showFigs = False
Tmax= 5000 # K
N_PT = 5# Temperatures per tooth
N_T = 2# Number of teeth
alpha = 0.7 # Factor by which max T decreases in subsequent teeth

# Species List, currently using 2 NH4 and 2 Cl ions
speciesList = {"NH4": 4,
               "Cl": 4}

# Create system and modify energy parameters
Simulation.M = IC.Create_System.heterogeneous(speciesList, create_xyz =
False)
class_list= [calc.JBRQT, calc.N_Well] # Create potentials
list to use
```

```

calc.N_Well.n = 10                                # Change constraint n
in (r/box)^2n

class Anneal():

    sim_inputs = {
        "trials": RUNS,                            # Number of trials
        "n_eq": NEQ,                                # Equilibration trials
        "temperature": 0,                            # Temperature of run
        "save": SAVE,                                # No. trials before saving xyz
    files
        "step_size": DEFAULT_STEP,                  # Initial step size
        "step_check": CHECK,                         # No. of trials before correcting
    step size
        "outputPath": TRIAL_PATH,                   # Location of save file
        "seeding": True,                             # If True, will perform random seed
        "debug": False,                              # If True, will print a log of
    moves and energies to help debug the code
        "potential_list": class_list,
        "diffCenter": True,                          # Use different center point to
    calculate LJ
        "LJCenter": 0                               # Atom to use in LJ distance
    calculations
    }

    @classmethod
    def run(cls):

        dataStore = []
        temperatures = []
        EAcceptedCont = []
        T1 = Tmax

        for P in range(1, N_T+1):
            T = T1
            delT = T1/(N_PT - 1)

            for N in range(N_PT):
                print("\n\nP=", P, "N=", N, "T=", T)
                PNTString = "P"+ str(P) + "_N" + str(N) + "_T" + str(T) #
                Use in file names to identify N, P, Temperature
                #Set up xyz filename
                xyz_file = os.path.join(XYZ_PATH, PNTString)

                if P == 1 and N == 0:
                    cls.sim_inputs["temperature"] = T
                    cls.sim_inputs["outputPath"] = xyz_file

                    # RUN SIMULATION
                    Outputs = Simulation.run(**cls.sim_inputs)
                    M_fin = Outputs["M_fin"]

```

```

        U_accepted = Outputs["Accepted_Energies"].iloc[:,-1]
        all_accepted_U = U_accepted # Save to a running tally

        # Create Movie file for storing final position

data.XYZ.create(M_fin,path=os.path.join(XYZ_PATH,"Movie"))

    else:
        if np.round(T) == 0:
            T = 0

            # Simulation.M = deepcopy(M_fin )
            cls.sim_inputs["temperature"] = T
            cls.sim_inputs["outputPath"] = xyz_file

            # RUN SIMULATION
            Outputs = Simulation.run(**cls.sim_inputs)
            M_fin = Outputs["M_fin"]

            U_accepted = Outputs["Accepted_Energies"].iloc[:,-1]
            all_accepted_U = pd.concat([all_accepted_U , U_accepted])

            # Create Movie file for storing final position

data.XYZ.append(M_fin,path=os.path.join(XYZ_PATH,"Movie"))

        if T == 0:
            data.XYZ.create(M_fin, path=
os.path.join(XYZ_PATH,"Final_Structure_P={}".format(P)))
        else:
            T -= delT

            # Save all and accepted energies: LJ, Coulombic, Constraint,
Total

Outputs["Energy_Store"].to_csv(os.path.join(DATA_PATH,"Energy_Store_"+PNTString+".csv"))

Outputs["Accepted_Energies"].to_csv(os.path.join(DATA_PATH,"Accepted_Energies
_"+PNTString+".csv"))

        dataStore.append([T, M_fin, Outputs["step_size"]])
        temperatures.append(T)

        T1 = alpha*T1

        output_dict = {"temperatures": temperatures, "data_store": dataStore,
"M_fin": M_fin, "all_accepted_U": all_accepted_U }

        return output_dict

results = Anneal.run()
```



**File: parameters.py**

```
"""This file contains constants to be used as parameters for simulation"""
import numpy as np
import csv

# Atomic Masses of Elements [amu]
AMU = {'H': 1.00794,
       'He': 4.002602,
       'Li': 6.941,
       'Be': 9.01218,
       'B': 10.811,
       'C': 12.011,
       'N': 14.00674,
       'O': 15.9994,
       'F': 18.998403,
       'Ne': 20.1797,
       'Na': 22.989768,
       'Mg': 24.305,
       'Al': 26.981539,
       'Si': 28.0855,
       'P': 30.973762,
       'S': 32.066,
       'Cl': 35.4527,
       'Ar': 39.948,
       'K': 39.0983,
       'Ca': 40.078,
       'Sc': 44.95591,
       'Ti': 47.88,
       'V': 50.9415,
       'Cr': 51.9961,
       'Mn': 54.93805,
       'Fe': 55.847,
       'Co': 58.9332,
       'Ni': 58.6934,
       'Cu': 63.546,
       'Zn': 65.39,
       'Ga': 69.723,
       'Ge': 72.61,
       'As': 74.92159,
       'Se': 78.96,
       'Br': 79.904,
       'Kr': 83.8,
       'Rb': 85.4678,
       'Sr': 87.62,
       'Y': 88.90585,
       'Zr': 91.224,
       'Nb': 92.90638,
       'Mo': 95.94,
       'Tc': 97.9072,
       'Ru': 101.07,
       'Rh': 102.9055,
       'Pd': 106.42,
       'Ag': 107.8682,
       'Cd': 112.411,
```

```

'In': 114.818,
'Sn': 118.71,
'Sb': 121.76,
'Te': 127.6,
'I': 126.90447,
'Xe': 131.29,
'Cs': 132.90543,
'Ba': 137.327,
'La': 138.9055,
'Ce': 140.115,
'Pr': 140.90765,
'Nd': 144.24,
'Pm': 144.9127,
'Sm': 150.36,
'Eu': 151.965,
'Gd': 157.25,
'Tb': 158.92534,
'Dy': 162.5,
'Ho': 164.93032,
'Er': 167.26,
'Tm': 168.93421,
'Yb': 173.04,
'Lu': 174.967,
'Hf': 178.49,
'Ta': 180.9479,
'W': 183.84,
'Re': 186.207,
'Os': 190.23,
'Ir': 192.22,
'Pt': 195.08,
'Au': 196.96654,
'Hg': 200.59,
'Tl': 204.3833,
'Pb': 207.2,
'Bi': 208.98037,
'Po': 208.9824,
'At': 209.9871,
'Rn': 222.0176,
'Fr': 223.0197,
'Ra': 226.0254,
'Ac': 227.0278,
'Th': 232.0381,
'Pa': 231.03588,
'U': 238.0289,
'Np': 237.048,
'Pu': 244.0642,
'Am': 243.0614,
'Cm': 247.0703,
'Bk': 247.0703,
'Cf': 251.0796,
'Es': 252.083,
'Fm': 257.0951,
'Md': 258.1,
'No': 259.1009,
'Lr': 262.11}

```

```

AMU.update({"foo": 0.0}) # Add this term for placeholder (TIP4P)

Collision_Sphere = {"H2O": 2,
    "N2": 0.548,
    "SO2": 2.47,
    "NH3": 3.0,
    "SF4": 4.0,
    "SF6": 3.1214+1,
    "H2SO4": 4.0492+1,
    "NH4": 1.68,
}

# TIP3P Data
TIP3P = {"OH_distance": 0.9572, # From TIP3P model
    "HH_distance": 1.514, # Calculated manually OH distance and dihedral
    "Dihedral": 104.52, # Dihedral angle
    "A": 582.0*1000, # Parameter A for Lennard Jones
    "B": 595.0, # Parameter B for Lennard Jones
    "sigma": 3.1506,
    "epsilon": 76.54*1.987e-3,
    "q_O": -0.834, # Charge of oxygen
    "q_H": 0.417, # Charge of hydrogen
    "k_c": 332.1 # [A kcal/ mol e^2] Electrostatic constant
}

TIP4P = {"r_OH": 0.9572,
    "r_OM": 0.15,
    "HOH_angle": 104.52,
    "sigma": 3.154,
    "A": 600.0e-3,
    "B": 610.0,
    "q_M": -1.04,
    "q_H": 0.52,
    "k_c": 332.1 # [A kcal/ mol e^2] Electrostatic constant
}

SO2 = { "S": {"epsilon": 73.8, "sigma": 3.39, "q": 0.59},
    "O": {"epsilon": 79.0, "sigma": 3.05, "q": -0.295}}

NH3 = {"N": {"epsilon": 85.458, "sigma": 3.42, "q": -1.02},
    "H": { "q": 0.34}}

CO2 = { "C": {"epsilon": 52.84, "sigma": 3.75, "q": 0.7},
    "O": {"epsilon": 63.41, "sigma": 2.96, "q": -0.35}}

# System Temperature
T = 1.5

# Maximum step size for normal steps (Angstroms)
L = 2.0

# Magnification factor for L
beta = 10

# Probability of a translational magstep
P_MT = 0.1

```

```
# Probability of a rotational magstep
P_RT = 0.1

k_B = 0.0019872041 # Boltzmann constant [kcal/mol-K]

L_MAG = beta * L # Magnified step

R = 1.987e-3 # Gas constant [kcal/mol/K]
```

## References

---

- <sup>1</sup> Topper, R. Q.; Feldmann, W. V.; Markus, I. M.; Bergin, D.; Sweeney, P. R. Simulated Annealing and Density Functional Theory Calculations of Structural and Energetic Properties of the Ammonium Chloride Clusters  $(\text{NH}_4\text{Cl})_n$ ,  $(\text{NH}_4^+)(\text{NH}_4\text{Cl})_n$ , and  $(\text{Cl}^-)(\text{NH}_4\text{Cl})_n$ ,  $n = 1-13$ . *The Journal of Physical Chemistry A* **2011**, 115 (38), 10423–10432.
- <sup>2</sup> Kirkpatrick S.; Gelatt, C.D. Jr.; Vecchi, M.P. (1983). Optimization by Simulated Annealing. *Science* 220 (4598); pp. 671-680.
- <sup>3</sup> Bertsimas, D., Tsitsiklis, J. (1992). Simulated Annealing. *Probability and Algorithms*.
- <sup>4</sup> Eckhardt, R. San Ulam, John Von Neumann, and the Monte Carlo Method. Los Alamos Science, Special Issue, **1987**.
- <sup>5</sup> Allen, M.P., and D.J. Tildesley. **1989**. Computer Simulation of Liquids. Oxford: Oxford Science Publications.
- <sup>6</sup> Metropolis, N.; Rosenbluth, A. W.; Rosenbluth, M. N.; Teller, A. H.; Teller, E. Equation of State Calculations by Fast Computing Machines. *The Journal of Chemical Physics* **1953**, 21 (6), 1087–1092.
- <sup>7</sup> Vlught, T. J. H. Introduction to Molecular Simulation and Statistical Thermodynamics; Vlught: Delft, The Netherlands, **2009**.
- <sup>8</sup> McQuarrie, D. Statistical Mechanics ; University Science Books: Sausalito, CA, **2000**.
- <sup>9</sup> Robert, C. P. The Metropolis–Hastings algorithm. <https://arxiv.org/pdf/1504.01896.pdf> (accessed Oct 12, 2020).
- <sup>10</sup> Topper, R. Q.; Freeman, D. L.; Bergin, D.; Lamarche, K. R. Computational Techniques and Strategies for Monte Carlo Thermodynamic Calculations, with Applications to Nanoclusters. *Reviews in Computational Chemistry* **2003**, 1–41.
- <sup>11</sup> Barker, J. A., & Watts, R. O. Structure of water; A Monte Carlo calculation. *Chemical Physics Letters*, **1969**. 3(3), 144–145. doi:10.1016/0009-2614(69)80119-3
- <sup>12</sup> Halgren, T. A. Potential Energy Functions. *Current Opinion in Structural Biology* **1995**, 5 (2), 205–210.
- <sup>13</sup> Dosso, S. E.; Oldenburg, D. W. Magnetotelluric Appraisal Using Simulated Annealing. *Geophysical Journal International* **1991**, 106 (2), 379–385. *Journal of the American Chemical Society*, 118(45), 11225–11236. doi:10.1021/ja9621760
- <sup>14</sup> Cornell, W. D., Cieplak, P., Bayly, C. I., Gould, I. R., Merz, K. M., Ferguson, D. M., Spellmeyer, D. C., Fox, T., Caldwell, J. W., & Kollman, P. A. A Second Generation Force Field for the Simulation of Proteins, Nucleic Acids, and Organic Molecules. *Journal of the American Chemical Society*, **1995**. 117(19), 5179–5197. <https://doi.org/10.1021/ja00124a002>
- <sup>15</sup> Mark, P., Nilsson, L. Structure and Dynamics of the TIP3P, SPC, and SPC/E Water Models at 298K. *Journal of Physical Chemistry*. **2001**, 105, 9954–9960.
- <sup>16</sup> Lorentz, H. A. Ueber Die Anwendung Des Satzes Vom Virial in Der Kinetischen Theorie Der Gase. *Annalen der Physik* **1881**, 248 (1), 127–136.
- <sup>17</sup> Berthelot, D. *Comptes. Rendus. Acad. Sci.* **1898**, 126, 1703–1855.
- <sup>18</sup> Matro, A.; Freeman, D. L.; Topper, R. Q. Computational Study of the Structures and Thermodynamic Properties of Ammonium Chloride Clusters Using a Parallel Jump-Walking Approach. *The Journal of Chemical Physics* **1996**, 104 (21), 8690–8702.
- <sup>19</sup> *Python Release Python 3.6.0*. (2016, December 23). Python.Org.

- 
- <sup>20</sup> Harris, C. R.; Millman, K. J.; Walt, S. J. V. D.; Gommers, R.; Virtanen, P.; Cournapeau, D.; Wieser, E.; Taylor, J.; Berg, S.; Smith, N. J.; Kern, R.; Picus, M.; Hoyer, S.; Kerkwijk, M. H. V.; Brett, M.; Haldane, A.; Río, J. F. D.; Wiebe, M.; Peterson, P.; Gérard-Marchant, P.; Sheppard, K.; Reddy, T.; Weckesser, W.; Abbasi, H.; Gohlke, C.; Oliphant, T. E. Array Programming with NumPy. *Nature* **2020**, 585 (7825), 357–362.
- <sup>21</sup> Hunter, J. D. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering* **2007**, 9 (3), 90–95.
- <sup>22</sup> Pyplot tutorial. <https://matplotlib.org/tutorials/introductory/pyplot.html> (accessed Oct 10, 2020).
- <sup>23</sup> McKinney, W. Data Structures for Statistical Computing in Python. *Proceedings of the 9th Python in Science Conference* **2010**.
- <sup>24</sup> XYZ (format) - Open Babel. **2007**. Open Babel. [http://openbabel.org/wiki/XYZ\\_%28format%29](http://openbabel.org/wiki/XYZ_%28format%29)
- <sup>25</sup> Hanwell, M. D.; Curtis, D. E.; Lonie, D. C.; Vandermeersch, T.; Zurek, E.; Hutchison, G. R. Avogadro: an Advanced Semantic Chemical Editor, Visualization, and Analysis Platform. *Journal of Cheminformatics* **2012**, 4 (1).
- <sup>26</sup> Smith, D. G. A.; Burns, L. A.; Simmonett, A. C.; Parrish, R. M.; Schieber, M. C.; Galvelis, R.; Kraus, P.; Kruse, H.; Remigio, R. D.; Alenaizan, A.; James, A. M.; Lehtola, S.; Misiewicz, J. P.; Scheurer, M.; Shaw, R. A.; Schriber, J. B.; Xie, Y.; Glick, Z. L.; Sirianni, D. A.; O'Brien, J. S.; Waldrop, J. M.; Kumar, A.; Hohenstein, E. G.; Pritchard, B. P.; Brooks, B. R.; Schaefer, H. F.; Sokolov, A. Y.; Patkowski, K.; DePrince, A. E.; Bozkaya, U.; King, R. A.; Evangelista, F. A.; Turney, J. M.; Crawford, T. D.; Sherrill, C. D. Psi4 1.4: Open-Source Software for High-Throughput Quantum Chemistry. *The Journal of Chemical Physics* **2020**, 152 (18), 184108.
- <sup>27</sup> Indiana University. <https://kb.iu.edu/d/agsz> (accessed Oct 24, 2020).
- <sup>28</sup> Jorgensen, W. William L. Jorgensen Research Group. <http://zarbi.chem.yale.edu/software.html> (accessed Oct 24, 2020).
- <sup>29</sup> Jorgensen, W. L.; Tirado-Rives, J. Molecular Modeling of Organic and Biomolecular Systems Using BOSS And MCPRO. *Journal of Computational Chemistry* **2005**, 26 (16), 1689–1700.
- <sup>30</sup> Jorgensen, W. L. BOSS, Version 4.9. <http://zarbi.chem.yale.edu/doc/boss49.pdf> (accessed Oct 24, 2020).
- <sup>31</sup> <http://ww2.wavefun.com/products/spartan.html> (accessed Oct 13, 2020).
- <sup>32</sup> Tutorial and User's Guide. Spartan '18. **2020**. <http://downloads.wavefun.com/FAQ/Spartan18Manual.pdf>
- <sup>33</sup> Kühne, T. D.; Iannuzzi, M.; Ben, M. D.; Rybkin, V. V.; Seewald, P.; Stein, F.; Laino, T.; Khaliullin, R. Z.; Schütt, O.; Schiffmann, F.; Golze, D.; Wilhelm, J.; Chulkov, S.; Bani-Hashemian, M. H.; Weber, V.; Borštnik, U.; TAILLEFUMIER, M.; Jakobovits, A. S.; Lazzaro, A.; Pabst, H.; Müller, T.; Schade, R.; Guidon, M.; Andermatt, S.; Holmberg, N.; Schenter, G. K.; Hehn, A.; Bussy, A.; Belleflamme, F.; Tabacchi, G.; Glöß, A.; Lass, M.; Bethune, I.; Mundy, C. J.; Plessl, C.; Watkins, M.; Vandevondele, J.; Krack, M.; Hutter, J. CP2K: An Electronic Structure and Molecular Dynamics Software Package - Quickstep: Efficient and Accurate Electronic Structure Calculations. *The Journal of Chemical Physics* **2020**, 152 (19), 194103.
- <sup>34</sup> Cassandra. <https://cassandra.nd.edu/> (accessed Oct 23, 2020).

- 
- <sup>35</sup> Shah, J. K.; Marin-Rimoldi, E.; Mullen, R. G.; Keene, B. P.; Khan, S.; Paluch, A. S.; Rai, N.; Romaniello, L. L.; Rosch, T. W.; Yoo, B.; Maginn, E. J. Cassandra: An Open Source Monte Carlo Package for Molecular Simulation. *Journal of Computational Chemistry* **2017**, 38 (19), 1727–1739.
- <sup>36</sup> Tinker - Software Tools for Molecular Design. <https://dasher.wustl.edu/tinker/> (accessed Oct 24, 2020).
- <sup>37</sup> Ponder, J. TINKER Tools for Molecular Design <https://dasher.wustl.edu/tinker/downloads/tinker-guide.pdf>. <https://dasher.wustl.edu/tinker/downloads/tinker-guide.pdf> (accessed Oct 24, 2020).
- <sup>38</sup> Python Simulation Interface for Molecular Modeling. <https://pysimm.org/> (accessed Oct 23, 2020).
- <sup>39</sup> Demidov, A. G.; Fortunato, M. E.; Colina, C. M. Update 0.2 to “Pysimm: A Python Package for Simulation of Molecular Systems.” *SoftwareX* **2018**, 7, 70–73.
- <sup>40</sup> Openmm. <https://github.com/openmm/openmm> (accessed Oct 23, 2020).
- <sup>41</sup> P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, R. P. Wiewiora, B. R. Brooks, and V. S. Pande. “OpenMM 7: Rapid development of high performance algorithms for molecular dynamics.” *PLOS Comp. Biol.* 13(7): e1005659. (2017)
- <sup>42</sup> Tyson, J. How Sword Making Works. <https://science.howstuffworks.com/sword-making.htm> (accessed Sep 4, 2020). <https://science.howstuffworks.com/sword-making5.htm>
- <sup>43</sup> Dosso, S. E.; Oldenburg, D. W. Magnetotelluric Appraisal Using Simulated Annealing. *Geophysical Journal International* **1991**, 106 (2), 379–385.
- <sup>44</sup> Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. *Science*, 220(4598), 671–680. <https://doi.org/10.1126/science.220.4598.671>
- <sup>45</sup> Torres, F. M.; Agichtein, E.; Grinberg, L.; Yu, G.; Topper, R. Q. A note on the application of the "Boltzmann simplex"-simulated annealing algorithm to global optimizations of argon and water clusters. <https://www.sciencedirect.com/science/article/abs/pii/S0166128097001954> (accessed Sep 8, 2020).
- <sup>46</sup> [https://physics.nist.gov/cgi-bin/Compositions/stand\\_alone.pl?ele=](https://physics.nist.gov/cgi-bin/Compositions/stand_alone.pl?ele=) (accessed Nov 18, 2020).
- <sup>47</sup> Bondi, A. van der Waals Volumes and Radii. *Journal of Physical Chemistry*. 1964. Vol. 68, No. 3, 441–451.
- <sup>48</sup> National Institute of Standards and Technology. CCCBDB Experimental Geometry Data. Computational Chemistry Comparison and Benchmark DataBase. **2019**. <https://cccbdb.nist.gov/expgeom1x.asp>
- <sup>49</sup> Jorgensen, W. L., Jenson, C. Temperature Dependence of TIP3P, SPC, and TIP4P Water from NPT Monte Carlo Simulations: Seeking Temperatures of Maximum Density. *Journal of Computational Chemistry*. **1998**. Vol. 19, No. 10, 1179–1186.
- <sup>50</sup> Rakshit, A.; Bandyopadhyay, P.; Heindel, J. P.; Xantheas, S. S. Atlas of putative minima and low-lying energy networks of water clusters  $n = 3–25$ . *The Journal of Chemical Physics*. **2019**. 151(21), 214307. doi:10.1063/1.5128378
- <sup>51</sup> Yin, J.; Landau, D.P. Wang Landau approach to the simulation of water clusters. *Molecular Simulation*. **2018**. 45:4–5, 241–248.
- <sup>52</sup> Fyta, Maria. Water models in classical simulations. <https://www2.icp.uni-stuttgart.de/>
- <sup>53</sup> 8.5.2. TIP3P water model — LAMMPS documentation. **2020**. LAMMPS. [https://lammps.sandia.gov/doc/Howto\\_tip3p.html](https://lammps.sandia.gov/doc/Howto_tip3p.html)

- 
- <sup>54</sup> Niese, J. A.; Mayne, H. R. Global Optimization of Atomic and Molecular Clusters Using the Space-Fixed Modified Genetic Algorithm Method. *Journal of Computational Chemistry* **1997**, 18 (9), 1233–1244.
- <sup>55</sup> Wales, D. J.; Hodges, M. P. Global Minima of Water Clusters (H<sub>2</sub>O)<sub>n</sub>,  $n \leq 21$ , Described by an Empirical Potential. *Chemical Physics Letters* **1998**, 286 (1-2), 65–72.