4G04 Assignment 2

Shayaan Siddiqui
400247500
siddis41

October 25,2021

Abstract

This second assignment incorporates random numbers and Monte Carlo methods to evaluate functions and stochastic models. Initially, a Gaussian integral will be evaluated using the Monte Carlo integral and uniform random numbers. Then using importance sampling, non uniform random numbers can be implemented to solve the integral. This is compared with uniform random numbers through resultant accuracy and evaluating the error of each method. The code implemented also has two unique c++ functions that use non-uniform random numbers which will also be compared with the other methods implemented. The last topic studied is the Ising model. Using the Metropolis algorithm, a net probability can be solved for using the Markov chain method implementing transition probabilities such that a state of a particle in a lattice would flip and eventually reach a steady state after repeated random measurements of the lattice.

1 Introduction

The root concept of understanding how a Monte Carlo simulation works is by thoroughly understanding how random numbers can assist in making random measurements that create a probabilistic outcome with some error that can accumulate due to repetitive nature of the measurements. To preform a Monte Carlo integral, the average value of the function in between the bound of the integral can be solved using a similar method as the trapezoidal method. Rather this is the expectation value of the function in between the bounds. The area under the curve can be reformed into a rectangle where the height is the expectation value and the width is the difference between the bounds of the system. Occasionally, the initial probability density function has a very high peak which makes it difficult to extract high precision values as certain values have greater impact than others. Importance sampling flattens out the probability density thus giving greater importance to values that did not initially have any impact onto the result. This will reduce the variance in the computed integral.

At times the probability is too difficult to compute. To compensate for this, the Markov method in implemented. This uses transition probabilities that essentially walk through the net probability density until the expectation value is reached. The computational application of this method involves using a algorithm called the Metropolis Algorithm which pertains to the Ising model. The Ising model is a probabilistic model that will determine magnetic moments caused by the spins of individual atoms in a lattice of atoms. Using a fixed relationship for this model called the detailed balance equation, the probability of switching a particular state can be calculated. Running this algorithm for many iterations, also known as Monte Carlo time, can provide an accurate assessment of what the expectation values of the lattice would be at a specific temperature. By graphing these values in respect to time or temperature, the properties of this algorithm can be further studied.

1.1 Monte Carlo Integral

When programming in C++, an effective way to calculate the area under a curve involves using the Monte Carlo simulation. In the previous assignment, the trapezoidal integral was explored. The Monte Carlo integral follows a similar process. Equation 1 will show how an integral can be put into

a summation using the Monte Carlo method.

$$I = \int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{i=0}^{N} f(X_i) = \frac{1}{N}\sum_{i=0}^{N}\frac{f(X_i)}{pdf(X_i)} \tag{1}$$

Using equation 1 solving an integral using the Monte Carlo method becomes fairly trivial. Note this is when the probability density is uniformly distributed. A non-uniform probability density will be explored later on. The argument of the function within the summation, X_i can be equated to $a+\xi(b-a)$ where ξ is a uniform random number in between 0 and 1. To compute the error produced by the Monte Carlo simulation the square root of the variance is taken. This is also known as the standard deviation σ_N. The standard deviation uses the expectation value of the function all squared and argument squared. This is shown below.

$$\sigma_N = (b-a)\sqrt{\frac{(\frac{1}{N}\sum_{i=0}^{N} f(X_i)^2) - (\frac{1}{N}\sum_{i=0}^{N} f(X_i))^2}{N}} \tag{2}$$

Equation 2 shows that $\sigma_N \propto \frac{1}{\sqrt{N}}$[1]. Thus, when N is very larger, the error approaches 0 and $N-1$ becomes a value so close to N that it does not make a significant difference to the error evaluation by not subtracting 1 from N. For the purpose of this first exercise, a very popular function is being studied. It is known as the Gaussian function which has many applications related to probabilities and statistics. This function will be integrated from the bounds 0 to 1.

$$\int_0^1 e^{-x^2}\,dx \tag{3}$$

To evaluate this integral, the first method used will be the Monte Carlo method. A C++ code will be implemented to solve the approximate area under the curve and the error associated with the solution.

```
Integral using Monte Carlo method evaluates to:0.746305726704451 +- 0.000636007310074676
```

Figure 1: Evaluated Integral using Crude Monte Carlo Method.

The known value of Equation 3 is 0.74682413281243. The value found in Figure 3 fits within 0.069 % of the known value. The error found with the program fits within the actual value therefore this evaluation of the integral is reasonable. Using uniform random numbers are beneficial when the functions shape is unknown but if the functions distribution is known, a method known as importance sampling can be used. Importance sampling uses a probability density (PDF) that is similar to the original function's density. Using this PDF the integral is manipulated such that the evaluation of the integral is the same but certain values will have more effects that others. Thus the term importance sampling. The Gaussian function has larger peak in the domain of (-0.5,0.5). Since the integral being computed for this exercise is in the positive x region, a probability density localized within the positive x domain and denser around 0 is desirable.
The PDF used for this integral is $\frac{e^{-x}e}{e-1}$.
The probability density function gives importance to the values closer to zero as shown in Figure 5. To incorporate this PDF into the solution, the Cumulative and Quantile function have to be solved. The Quantile function is the integral of the Probability Density function and the Cumulative Density function is the inverse of the Quantile function.
Set the PDF as g(x).

$$cdf(x) = G(x) = \int_0^x g(x)\,dx = \int_0^x \frac{e^{-x}e}{e-1}\,dx = \frac{(1-e^{-x})e}{e-1} = u \tag{4}$$

$$qdf(u) = G^{-1}(u) = -ln(1-u\frac{e-1}{e}) = x \tag{5}$$

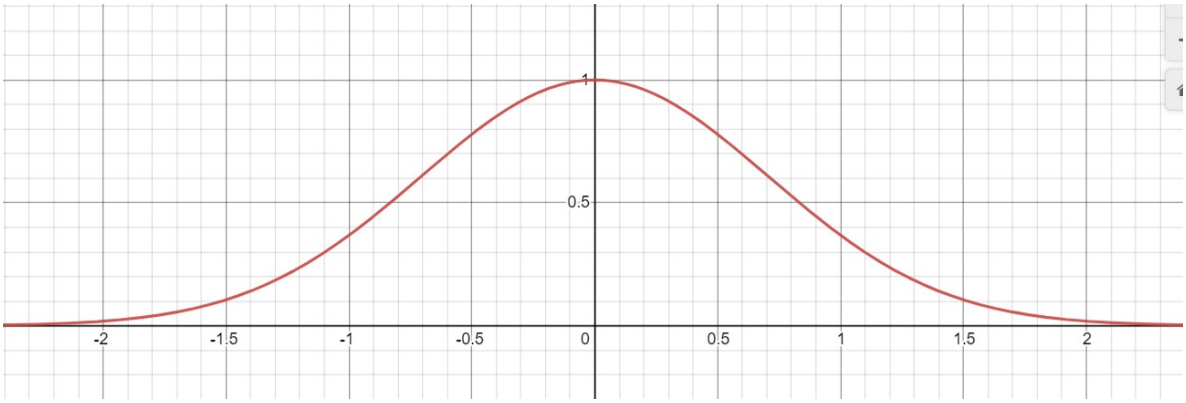

Figure 2: Gaussian curve for e^{-x^2}.

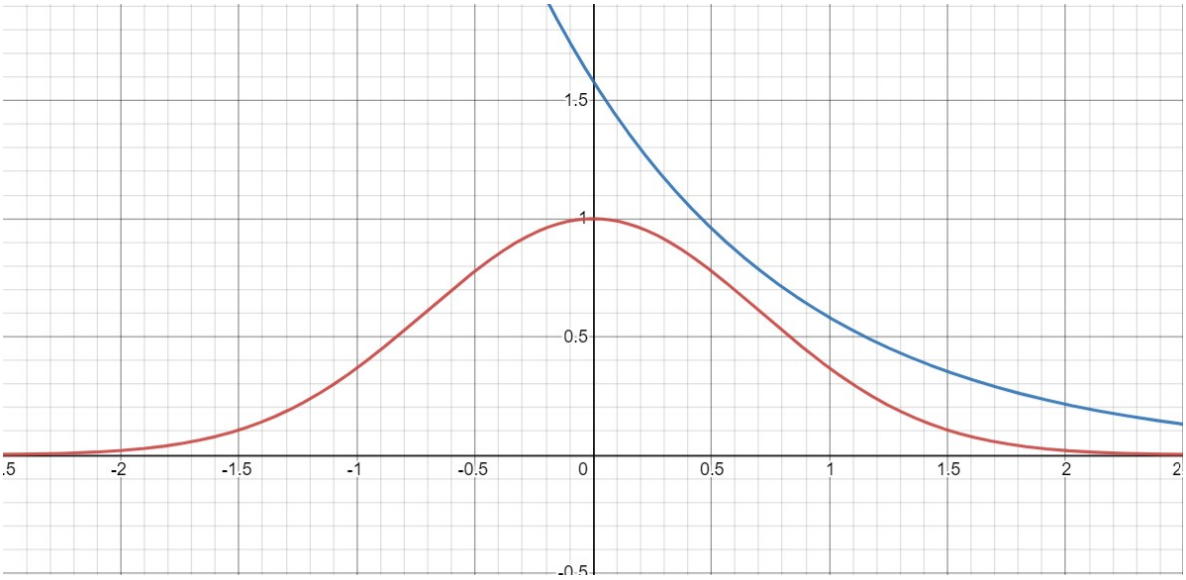

Figure 3: Probability density function in respect to original function.

The QDF is also the x value. The variable u is a uniform random number between 0 and 1. By using a uniform random number in a non-uniform distributed function, the outcome is a set of non-uniform random numbers that outputs values where the probability is most dense.

$$\int_a^b \frac{f(x)}{g(x)}\, g(x)\, dx \tag{6}$$

Using the previous information from equation 4, the relationship between the PDF and CDF can also be described by equation 7. The following formulas will show how to use a CDF, QDF and PDF to transform the integral to create variance reduction when preforming this integral.

$$g(x)\, dx = dG(x) \tag{7}$$

$$\int_a^b \frac{f(x)}{g(x)}\, dG(x) \tag{8}$$

$$dG(G^{-1}(u)) = du \tag{9}$$

$$\int_{G(a)}^{G(b)} \frac{f(G^{-1}(u))}{g(G^{-1}(u))}\, du \tag{10}$$

Implementing the above formula into C++ should in turn cause a value where the variance is significantly smaller.To implement this into C++, a function is made for each of the desired equations that will be implemented. The Monte Carlo integral will have it's own function and will pass a function as one of it's parameters, the upper and lower bounds and the amount of iterations. Both figure 5 and figure 1 will have the same amount of iterations being 100000.

```
//initial function
double myfunc(double x)
{
    double r;
    r = exp(-(x * x));
    return r;
}
//probability density
double density(double x)
{
    double r;
    r = (exp(-x)*exp(1))/(exp(1)-1);
    return r;
}
//CDF
double normal(double x) {
    double r;
    r = exp(1) * (1 - exp(-x)) / (exp(1) - 1);
    return r;
}
//Quantile function
double inv_normal(double x) {
    double r;
    r = -log(1-x*(exp(1)-1)/exp(1));
    return r;
}
//importance sampling function
double new_func(double x) {
    double r;
    r = myfunc(inv_normal(x))/ density(inv_normal(x));
    return r;
}
```

Figure 4: Importance Sampling C++ functions.

```
Integral using Importance Sampling evaluates to:0.746733651146968 +- 0.000173823714162198
```

Figure 5: Error Reduction using Importance Sampling.

When comparing the two values the error for the importance sampling is approximately 3 times less than the error when using the crude Monte Carlo method. This is not necessary due to just using non-uniform number generation but due to the integral transformation done using the probability density, cumulative density and quantile density function all together. The code shown in 4.1 contains two non uniform random number generators, the first using the exponential distribution and the second using the Box-Muller distribution.

```
Integral using non-uniform random numbers and crude Monte Carlo integral: 0.746722360530054 +- 0.00063675197288867
```

Figure 6: Monte Carlo solution using non-uniform random numbers from exponential distribution.

```
Integral using Box Muller: 0.793081494074472 +- 0.0006005423858155 34
```

Figure 7: Monte Carlo solution using non-uniform random numbers from Box Muller distribution.

Rather than manipulating the integral, instead of using uniform random number generator, a non uniform random number was generated and implemented into the crude Monte Carlo method. This creates no difference as the error is essentially the same and the evaluated integral for the Box Muller method creates a greatly inaccurate evaluated integral. This is likely due to the fact the probability density of the Box Muller method is too great in unimportant locations of the Gaussian function.

1.2 Monte Carlo Integral Modification

The reason why $X_i = a + \xi(b-a)$ is due to the fact that since ξ is a uniform number in domain interval of (0,1). The length $(b-a)$ defines the integral bounds and by multiplying by a random float, adding it to the lower bound will produce a uniform random number within the bounds of the integral. Having this embedded into the Monte Carlo function made into C++ and passing the bounds as parameters ensures the output is accurate. Notice in Figure 8, what the variable function 1 equates to

```
//monte carlos function
double MC(double (*f)(double x),double N,double b, double a) {
    double function1;
    double ds = 0;
    double sum=0,avgf;

    for (int n = 0; n < N; ++n) {
        // comment and uncomment next 2 lines to use uniform or non uniform random numbers
         function1 = f(a + (b - a) * uniform_rand());
         //function1=f(a+(b-a)*non_uniform_rand2());
         sum += function1;//summation for monte carlos integration
         ds += function1*function1; //used for error
    }
    sum = (b-a)*sum / N;//area under the curve
    avgf = sum / (b - a);//expectation value
    ds = ds/N;//function squared expectation value
    ds = (b - a) * sqrt( abs(ds-pow(avgf,2))/N);//error
    //cout << "Integral using Monte Carlo method evaluates to: " << setprecision(15) << sum << " +- " << setprecision(15) << ds << endl << endl;
    cout << "Integral using Importance Sampling evaluates to: " << setprecision(15) << sum << " +- " << setprecision(15) << ds << endl<<endl;
    //cout << "Integral using non-uniform random numbers and crude Monte Carlo integral: " << setprecision(15) << sum << " +- " << setprecision(15) << ds << endl << endl;
    //cout << "Integral using Box Muller: " << setprecision(15) << sum << " +- " << setprecision(15) << ds << endl << endl;
    return 0;
}
int main()
{
    double b = 10, a = 0;
    //MC(&myfunc, 1000000, b, a);// using crude monte carlo method
    MC(&new_func, 1000000, normal(b), normal(a)); //using importance sampling
```

Figure 8: Code used to solve integral using Monte Carlo method.

is exactly the argument of the sum as shown in Equation 1. The code contains comments to show how to interchange the output, which is dependant on what function to input, and what type of random number is being used.
Now, the integral bounds have changed from where $b = 10$ and $a = 0$. To gain a higher accuracy the amount of iterations, N will be increased to 1000000. The integral solution is 0.88622692545276.

When comparing the outputs from importance sampling and the Crude Monte Carlo method, it

```
Integral using Monte Carlo method evaluates to: 0.886769265526471 +- 0.0023407637602788
```

Figure 9: Gaussian integral from bounds 0-10 using Crude Monte Carlo Integral.

```
Integral using Importance Sampling evaluates to: 0.886333627730001 +- 0.000443263962011566
```

Figure 10: Gaussian integral from bounds 0-10 using Importance sampling.

is clear that using importance sampling is much more beneficial to output a higher accuracy value. Not only is the error significantly smaller, reduced by a factor of 5, but the evaluated integral is also closer to the solution. It is not only due to the use of non-uniform random numbers, as shown from figures 6, and 7 which does not increase accuracy. The reason using non uniform random numbers is beneficial is due to the relationships of the PDF, CDF and QDF's implementation within the integral transformation when using the PDF notation of the summation as shown in 1 rather than the format without the PDF.

2 The Two Dimensional Ising Model

The Ising Model describes the magnetization of a lattice. This is described by the individual spins, which is known as the magnetic moment, of an atom in the lattice summed to solve for total magnetization. The magnetization is dependant on the temperature of the lattice and the interaction constant. The initial configuration will determine how the lattice will finally coalesce considering the

net probability. This probability is based of the Boltzmann Distribution where $\beta = \frac{1}{K_B T}$, K_B is the Boltzmann constant and T is a temperature value.

$$P_\beta(\sigma) = \frac{e^{-\beta H(\sigma)}}{Z_\beta} \tag{11}$$

$$Z_\beta = \sum_\sigma e^{-\beta H(\sigma)} \tag{12}$$

Since the Boltzmann distribution is dependant on a sum in the denominator, known as the partition function, this probability distribution can become quite difficult to calculate as there are many possible configurations and σ are the spins within the lattice. Thus, a Monte Carlo method makes this process much simpler. Rather than calculating the net probability, a method to solve individual transition probabilities of a single atom in the lattice can be repeatedly done to find the final configuration of the lattice. This method is called the Markov process, where the next individual transition probability is dependant on the previous probability to determine the final net probability. The chosen algorithm used to find variables of interest from the Ising model is called the Metropolis algorithm and is a variation of the Markov process. This algorithm follows 4 simple steps to decide what the final lattice is and is heavily dependent on a relationship called the detailed balance.

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{p_\nu}{p_\mu} = e^{-\beta(E_\nu - E_\mu)} \tag{13}$$

Both μ and ν are unique configurations of the lattice and the following shows the transition probability. The denominator can be set to 1 which would be the initial state and the transition probability would be $e^{-\beta(E_\nu - E_\mu)}$. This works vice versa setting p_ν as 1 or p_μ as 1.

The Metropolis Algorithm

1. Configure a 2D lattice of spins in a random arrangement
2. Choose a random spin in the lattice
3. Find the change in energy associated with flipping the spin
4. Generate a uniform random number and if less than the probability given in 13 then flip the state. Start over at step 2 and repeat.

For the purposes of this assignment, the magnetization, average energy, magnetic susceptibility, heat capacity and the finite scaling at various temperatures and lattice sizes.

The first step into making this simulation is generating a $L \times L$ lattice consisting of a random arrange of atomic spin values. To do this, a uniform random generator from 0 to 1 was used and if the values were equal to or greater than 0.75, the spin is set to positive, otherwise this the spin is negative. This creates a matrix consisting of mainly negative spin particles. For the Monte Carlo simulation to run,

```
double uniform_rand() {
    uniform_real_distribution<> u(0, 1);

    random_device r; // Seed with a real random device
    seed_seq seed{ r(), r(), r(), r(), r(), r(), r(), r() };
    mt19937 engine2(seed);
    return u(engine2);
}
//uses random number to output a random spin value with higher probability of negative
double spin() {
    double val;
    if (uniform_rand() >= 0.75) {
        val = 1;
    }
    else {
        val = -1;
    }
    return val;
}
//creates lattice full of randomly arranged spin 75% negative 25% positive probability density
vector<vector<double>> Matrix(double L) {
    vector<vector<double>> v;

    for (int i = 0; i < L; ++i) {

        v.push_back(vector<double>());
        for (int j = 0; j < L; ++j) {
            v[i].push_back(spin());
        }
    }
    return v;
}
```

Figure 11: Lattice Generation

the energy has to be calculated for the entire lattice. The lattice energy is calculated using a nearest neighbour summation.

$$E_\mu = -J \sum_{<i,j>} \sigma_i \sigma_j \tag{14}$$

Equation 14 is the energy associated with a unique configuration μ. Instead of calculating just E_μ, the simulation calculates $\frac{E_\mu}{J}$. This is reasonable due to the fact that temperature will be measured in units of $\frac{K_B T}{J}$ which will end up canceling out the J term. The variable J is known as the interaction strength. The following figure will show how to attain the energy of each particle within the lattice.

```
//individual particle calculation in lattice using lattice matrix, x coordinate and y coordinate as parameters
double Energy(vector<vector<double>> lattice, double x1,double y1) {
    double L = lattice.size() - 1,//due to indexing
        xsum,ysum,sum,//using x,y components to solve for net nearist neighbours energy
        x=x1-1,y=y1-1,//indexing actual value
        xm=x1-2,ym=y1-2;//1 less than actual coordinate
    //boundary conditions
    if (x==0) {//if leftmost atom then nearest nearbour sum in x is spin of atom times atom to right of it
        xsum = lattice[y][x1]* lattice[y][x];
    }
    else if(x==L){//if rightmost atom then xsum is atom times atom left to it
        xsum = lattice[y][xm] * lattice[y][x];
    }
    else {//any other atom in lattice, xsum is spin times both left and right atom summed together
        xsum = lattice[y][x1] * lattice[y][x] + lattice[y][xm] * lattice[y][x];
    }
    if (y == 0) {//upper most atom then ysum is atom times atom below it
        ysum = lattice[y1][x] * lattice[y][x];
    }
    else if (y == L) {//bottom atom then ysum is atom times atom above it
        ysum = lattice[ym][x] * lattice[y][x];
    }
    else {//any other atom in lattice y, atom times sum of atom above and below it
        ysum = lattice[y1][x] * lattice[y][x] + lattice[ym][x] * lattice[y][x];
    }
    sum = (xsum + ysum);//total energy of atom is atom below and above
    return sum;
}
//solves each atom in lattice energy and returns list for each atom energy
vector<double> particles_energy(vector<vector<double>> lattice){
    double size = lattice.size(),E;
    vector<double> E_list = {};
    //for loop references atoms using lattice size and not actual index, therefore indexing is done in Energy function
    for (double i = 1; i <= size; ++i) {
        for (double j = 1; j <= size; ++j) {
            E = Energy(lattice, j, i);

            //cout << E<<"    ";
            E_list.push_back(E);
        }
    }
    return E_list;
}
```

Figure 12: Return list containing energy of each atom in lattice

To solve for the energy, the sum of all index within the returned "particles_energy" list multiplied by negative one will give the energy of the lattice.

```
//sum of each particles energy or magnetization depending on the 1 D vector passed through parameter.
double system_energy(vector<double> IE) {
    double size = IE.size(), totalE = 0;
    for (int i = 0; i < size; i++) {
        totalE += IE[i];
    }
    return (-1 * totalE);// default is energy but if magnetization then multiply output by negative 1
}
```

Figure 13: Takes 1D list and sums all index within the list

When passing the list from the function in 12 into the function shown in 13, the energy of the lattice can be extracted in terms of $\frac{E}{J}$.
The energy will be used in the metropolis algorithm to determine the probabilities. The solution of the metropolis algorithm is to output the total Magnetization of the final lattice.

$$M = \sum_{i=1}^{N} \sigma_i \tag{15}$$

The total magnetization is equated to the sum of all the spin within the lattice. A function is built to

take the value of each spin within the lattice and put within a list, then using the function in 13, the sum of the 1D list would produce the net magnetization.

```
vector<double> magnetization(vector<vector<double>> lattice) {
    double size = lattice.size(), M;
    vector<double> M_list = {};
    for (double i = 0; i < size; ++i) {
        for (double j = 0; j < size; ++j) {
            M = lattice[j][i];
            M_list.push_back(M);
        }
    }

    return M_list;

}
-1 * system_energy(magnetization(new_lattice))
```

Figure 14: How to use functions to solve for lattice magnetization

Since the function shown in 13 returns the negative sum of a 1D vector, to accommodate, the value in 14 is multiplied by negative one again to accurately evaluate the magnetization using 15.

Now to actually preform the metropolis algorithm, a function called "MCSweeps" is made. This function will take three parameters, the lattice, the amount of Monte Carlo Sweeps, and the temperature of the system. The size of the lattice is not a parameter as the size is solved for within the function using the build in size() function in C++. A Monte Carlo sweep is defined as a $L \times L$ amount spin flip attempts where L is the length and width of the lattice. Then using the built in round() function applying it to the multiplication of the size of the lattice minus one by a uniform random number, a random x and y coordinate of the lattice can be found. Then set the found spin at the coordinates of the lattice as the initial spin and find the energy of the lattice. Then flip the spin by multiplying the initial spin by negative one and find the energy of the new lattice. The final aspect of this function is the create a few conditions to determine whether the final spin or initial spin is chosen for the updated lattice. If the final energy is greater than the initial energy and a random uniform number is less than the probability associated with the spin flip, the flipped spin is kept. If the final energy is less than the initial energy then the flipped spin is kept. If none of these condition are met, then keep the initial state. The final lattice after running multiple Monte Carlo sweeps will be returned from the function.

```
vector<vector<double>> MCSweeps(vector<vector<double>> lattice, double Nmcs, double B3) {
    pair<vector<double>, vector<double>> net_E;//output for function
    //variables needed
    double x, y,
        initial_spin, dE,
        final_spin, energy,
        L = lattice.size() - 1;
    //mutable matrix
    vector<vector<double>> spin_arrangement = lattice;
    //runs for loop for amount of monte carlo sweeps
    for (double n = 0; n < Nmcs; n++) {
        for (double i = 0; i < ((L+1)*(L+1)); i++) {
            double E_i = 0, E_f = 0; //continously reset initial energy and final energy

            //retrieve a random x and y coordinate in the range of the length of the LxL matrix
            x = round(L * uniform_rand());
            y = round(L * uniform_rand());
            //initial spin is the value of the lattice at this coordinate
            initial_spin = spin_arrangement[y][x];
            //calculate initial energy
            E_i = system_energy(particles_energy(spin_arrangement));

            //possible flipped state which would be final state
            final_spin = -1 * initial_spin;

            //setting the spin at lattice coordinate to flipped state
            spin_arrangement[y][x] = final_spin;

            //calculating lattice energy considering flipped spin
            E_f = system_energy(particles_energy(spin_arrangement));

            //reset coordinate to original state
            spin_arrangement[y][x] = initial_spin;
            //change in energy associated with flipping the state
            dE = E_f - E_i;
            //generate random number and if random number less than boltzman probability and flipped state energy is greater than initial state, flip the state evalutate new energy for lattice
            if (E_f > E_i && uniform_rand() < exp(-B3 * dE)) {
                spin_arrangement[y][x] = final_spin;
            }
            else if (E_i > E_f) {//if initial energy greater, then flip state, find new lattice energy
                spin_arrangement[y][x] = final_spin;
            }
            else {//any other situation, keep initial state and find energy
                spin_arrangement[y][x] = initial_spin;
            }
        }
    }
    return spin_arrangement;
}
```

Figure 15: Function to preform Monte Carlo sweeps and return final lattice.

Figure 15 follows the above steps to output the final lattice but to get reasonable results for desired values, it is better to do measurements after the system has reached equilibrium. A function "SQIsing" will take three parameters, the first being the amount of sweeps for the lattice to reach equilibrium, the second being the amount of observables recorded and the amount of sweeps taken in between each observable. To get better data results, running the simulation after the lattice reaches equilibrium. This function will output a list of energies and magnetization at points of interest.

```
pair <vector<double>, vector<double>>SQsing(double NWarmup, double NStep,double Nmeas, double L,double BJ) {
    vector<vector<double>> v = Matrix(L);
    pair<vector<double>, vector<double>> net_E;
    vector<vector<double>>new_lattice = MCSweeps(v, NWarmup,BJ);
    for (double i = 0; i < Nmeas; i++) {
        new_lattice=MCSweeps(new_lattice, NStep, BJ);
        net_E.first.push_back(system_energy(particles_energy(new_lattice)));//total energy of lattice at each sweep stored in vector
        net_E.second.push_back(-1 * system_energy(magnetization(new_lattice)));//total magnetization at each sweep stored in vector

    }
    return net_E;
}
```

Figure 16: Function to obtain measurements after equilibrium.

For the purpose of this assignment, depending on how iterations are preformed by applying the function in Figure 16, some of the data presented may use this function whereas others may take a data point at every iteration in the function "MCSweeps" shown in Figure 15. This would require a slight modification in the function "MCSweeps" which returns a pair list exactly shown in 16. This is due to computation time and on other occasions to provide better data sets.
Note, since a list is returned, the average, also known as the expectation value, shows which value is reasonable from all the measurements taken. When multiple measurements are taken, an error is also associated with the measurements taken. The expectation value is expressed as $< O^n >$, where O is any observable and n is the order of the observable.

$$< O^n >= \frac{1}{N} \sum_{i=1}^{N} O_i^n \tag{16}$$

$$\sigma_O^n = \sqrt{\frac{(< O^{2n} > - < O^n >^2}{N}} \tag{17}$$

Equation 16 shows how to calculate the expectation value of an observable and Equation 17 shows how to calculate the error of any data set.
The first data set measured is the magnetization at the temperature value of $\frac{K_B T}{J} = 2.26$ which is the same as setting $\beta J = \frac{1}{2.26}$ and comparing the magnetization at $\beta J = \frac{1}{2.45}$ The lattice size is a 50×50 lattice. Each Monte Carlo sweep would preform 2500 attempts at a spin flip. By preforming multiple sweeps, the amount of attempts become too large. The data is to be recorded on a histogram which is more effective with large sets of data. As such, rather than taking a data set using function "SQIsing", an observable is recorded at each spin flip attempt. The amount of attempts were 3000.

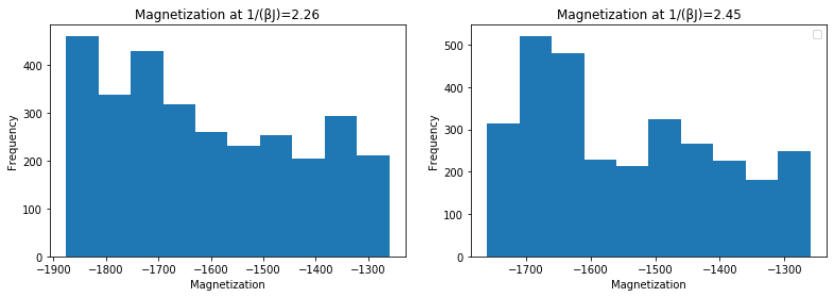

Figure 17: Function to obtain measurements after equilibrium.

It is clear that when the temperature is greater, the frequency of the magnetization is focused at a

lower magnitude whereas the frequency at a lower temperature, the magnetization is focused at a higher magnitude. This simulation data supports theory as temperature increases, the magnetization magnitude will decrease due to the disorder caused by the increase in temperature. The histogram shows that as more spin flip attempts are preformed, the magnetization of the lattice will begin to reach an equilibrium state. Since the the spin of the initial lattice is mainly negative, the equilibrium state tends toward a negative magnetization. Evidently it does not many spin flip attempts to reach a value where the magnetization reaches an approximate equilibrium state and accurately shows the trend when approaching an equilibrium state. Since the average magnetization of the lattice is not measure and the total magnetization is, histogram does not take form of the probability distribution. Also the measurements were not taken during equilibrium since at least 100 measurements would need to be taken and this is too many flip attempts taking over days to compute. Even so, the histogram distribution approaches the assumed distribution of a Gaussian distribution centered around zero when $T > T_c$ and two Gaussian distributions centered around $-M_0, M_0$ respectively when $T \lesssim T_c$.

Measuring Lattice Properties at Different Temperatures

Now a function will be made to measure the average energy, the magnetic susceptibility and the heat capacity at temperatures of $\frac{K_B T}{J} \in [2.0...2.5]$. Each of the values that are to be studied will have their own function associated with them. The temperature will increase from 0.01 and a measurement will taken at each of these temperature values. The amount of warmup sweeps preformed is five and in between every measurement there is one sweep. The amount of measurements taken is fifteen. This is sufficient enough to produce an accurate data set.

```
}//function for avg energy
vector<double>EE(double L) {
    vector<vector<double>> v = Matrix(L);
    vector<double>E1;

    for (double T = 2.0; T <= 2.5; T += 0.01) {
        double BJ = pow(T, -1);
        auto E = SQIsing(5,1,15, L, BJ);
        vector<double> Energy = (E.first);
        E1.push_back(avg(Energy, 1));
    }
    return E1;
}
```

Figure 18: Function to output various expectation value of energy at various temperatures.

A reasonable presumption made is that by increasing the temperature of the system, the energy of the system should increase. This is due to the fact that an increase in temperature introduces a thermal energy applied to the system. Thermal energy will cause excitation to the system to each particle in the system and the trend found by increasing the temperature should by that the temperature change is directly proportional to the energy change. To validate this theory, data collected is displayed using python plotting and a line of best fit is included to show the general trend.

When analyzing the data presented on the graph, it may not be immediately apparent what the trend is due to the probabilistic nature of this simulation. Using the matlab library, the polyfit function can output a linear trend line. Figure 19 clearly shows using the trend line that as the temperature increases, the average energy of the lattice will also increase. This may not be completely consistent for all increasing temperatures as there may be outliers but this is the general trend.

The magnetic susceptibility is a materials ability to interact with an eternal magnetic field. There are two cases, either the susceptibility is positive or negative. The polarity of the susceptibility will describe the properties of the material. When positive the material of interest is paramagnetic and

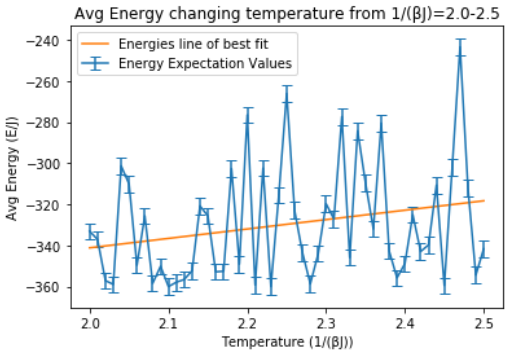

Figure 19: Expectation value of energy at various temperatures.

if negative, then the material is diamagnetic. A paramagnetic material is where the spin is randomly oriented and when a magnetic field is applied, the material will become slightly magnetized. A diamagnetic has no magnetic moment due to the fact that there are no unpaired electrons. An applied magnetic field on a paramagnetic material will cause the spin to line up with the magnetic field whereas a diamagnetic material spin will oppose the applied magnetic field lines.

$$\chi \approx \frac{< M^2 >}{TL^2} \tag{18}$$

Note, since the temperature is in units of $\frac{K_B T}{J}$, the units of magnetic susceptibility are in $\chi \frac{J}{K_B}$. Since the argument of the expectation value $< M^2 >$ in equation 18 is squared, this means the value of the susceptibility will always be positive. Thus no matter outcome or trend of the simulation, the subject of interest will always be prone to react to a magnetic field. A trivial method of reaching this conclusion is the fact that the spin arrangement in the lattice do not cancel with the amount of positives and negatives, thus having a magnetic moment caused by the spin. In fact, this means the material is ferromagnetic as the lattice has a permanent magnetic moment even when the system has no applied magnetic field and Formula 18 is the susceptibility related to ferromagnetism. A ferromagnet is a permanent magnet but is subject to change it's properties by changing temperature.

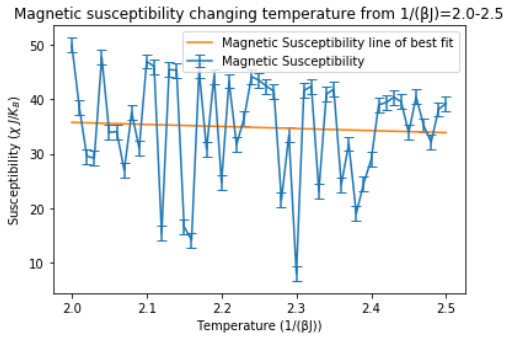

Figure 20: Magnetic susceptibility at various temperatures.

The trend line of the susceptibility shows that as the temperature increases, the susceptibility decreases. This is expected as the magnetization of the lattice decreases when the temperature increases. This is the net magnetic moment and the susceptibility is a function of this. The larger energy created by the increase of temperature causes the average spin approach to a value of zero since there is more disorder the spins start to oppose and reach an detailed balance susceptibility that also approaches zero. This means as the temperature of the lattice is increased, it is less reactive to an external magnetic field. The ferromagnetivity of the material decreases as the temperature of the material increases and reaches a state of low paramagnetivity where the spin will align in respect to an applied magnetic field. To produce Figure 20, there were 7 warmup sweeps, 1 sweep in between each measurement and

15 measurements taken.

The heat capacity is an extensive property of a material that will determine the amount of heat within a subject of interest for a specific value of temperature. Since heat capacity is an extensive property, various parameters can effect the output, consequently, all values are fixed and only the temperature is varying.

$$\frac{C_v}{k_B} = \frac{(< E^2 > - < E >^2)}{L^2 k_b T^2} \tag{19}$$

The heat capacity is proportional to the variance in energy and inversely proportional to the size, temperature squared. Since the heat capacity is known as $\frac{C_v}{k_B}$ and the energy is in units of $\frac{E}{J}$, when inputting the temperature in units of $\frac{K_B T}{J}$, the units of heat capacity is $\frac{C_v}{k_B^2}$

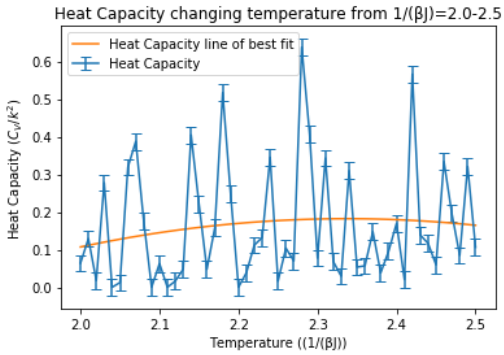

Figure 21: Heat Capacity at various temperatures.

Note, for Figure 21, a parabolic line of best fit is used. This is due to the fact that the trend varies at a point of interest. To produce this data, 7 warmup sweeps, a measurement at each sweep and 15 measurements are taken as previously done. In C++, a pair function is made to output a list of both the heat capacity and magnetic susceptibility. The following image shows the implementation.

```
//out puts X and heat capacity at various temperatures
pair<vector<double>, vector<double>>CP(double L) {
    //creates lattice
    vector<vector<double>> v = Matrix(L);
    pair<vector<double>, vector<double>> cp_x;
    double order1 = 1, order2 = 2;//order required to solve for
    for (double T = 2.0; T <= 2.5; T += 0.01) {//various temperatures

        double BJ = pow(T, -1);//beta is 1/kt
        auto E = SQIsing(7,1,15,  L, BJ);//preforms mc sweeps algorithm
        vector<double> Energy = (E.first);//energy vector
        vector<double> Magnetization = (E.second);//magnetization vector

        //expectation vaues
        double ExpE1 = avg(Energy, order1),
            ExpE2 = avg(Energy, order2),
            ExpM2 = avg(Magnetization, order2);
        //create vector of susceptibility
        cp_x.first.push_back(ExpM2 * BJ / pow(L, 2));
        //create vector of heat capacity
        cp_x.second.push_back(pow(BJ, 2) * (ExpE2 - pow(ExpE1, 2)) / (pow(L, 2)));

    }
    return cp_x;
}
```

Figure 22: C++ function to produce Magnetic Susceptibility and Heat Capacity data .

Figure 22 runs a loop through the various temperatures, inverts it as this is the value of beta and subs into the "SQIsing" function made in 16, then uses the respective formula 18 and 19 to create a list of data at various temperatures.
The critical temperature of the Ising model is a significant point of interest and this can be seen when analysing the previous data plots that are in respect to temperature. This point is known to be:

$$\frac{K_B T_c}{J} = \frac{2}{\ln(1+\sqrt{2})} = 2.2691853.. \tag{20}$$

The critical temperature of the Ising model is the temperature at which a material will undergo phase change and the properties will drastically fluctuate. It is not readily apparent in Figures 19, 20, and 21 but if the data is split into two separate portions where T_c is the midpoint, and make polynomial trend lines, the change in data becomes more apparent.

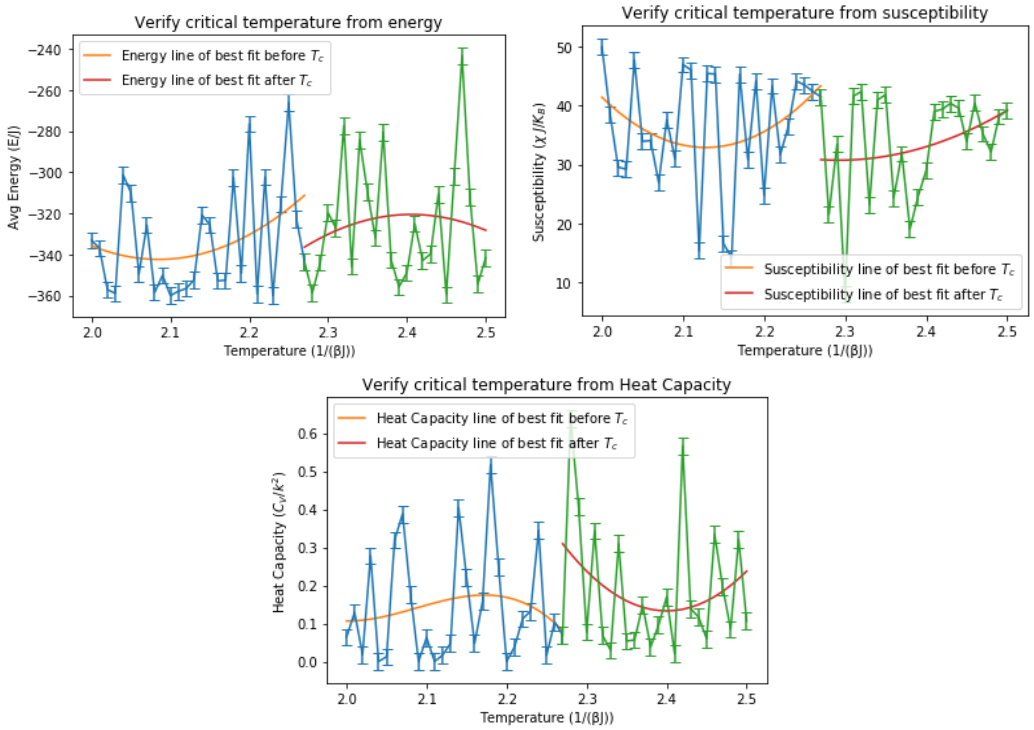

Figure 23: Verification of critical temperature using previous figures.

For the energy, the critical temperature is the approximate inflection point. From the temperature of 2.0-T_c, the concavity is upwards and after the critical temperature the concavity is downwards.
For the susceptibility, Figure 23 shows that before the critical temperature, the concavity is upwards. This is reasonable when using a polyfit with an order of two but the relationship is rather an increasing exponential in nature. This is when the material would be ferromagnetic. After the critical temperature, the concavity is still upwards but the actual relation ship is a decreasing exponential with a lower rate. Now the material would be a paramagnetic. The critical temperature determines when this this magnetic property changes.
The heat capacity at the critical temperature, the theoretical plot would have an asymptote at this value. Therefore around the critical temperature on both sides, the heat capacity would be approaching infinity. Decreasing when going from the critical temperature to the left and also from the critical temperature to the right.

When comparing Figure 23 to Figure 24, the trends do not differ by great magnitude although the simulation ran for this paper does not take a large amount of measurements for the range of temperature due to computation time. This is a drawback as the more iterations, the greater the accuracy of the simulation but it may take days to compute.

The final component of this report is to prove finite scaling theory using the simulation built. The function to explore this is known as the binder cumulant.

$$g(T,L) = \frac{1}{2}(3 - \frac{< M^4 >}{< M^2 >^2}) \tag{21}$$

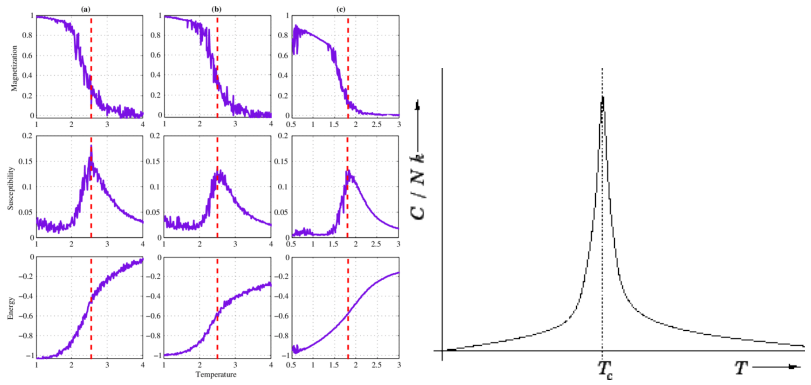

Figure 24: How the simulation graph output should look like with many more iterations [2] [3].

Finite scaling theory helps observe the fluctuations in properties when increasing the size of the lattice. The two pieces of data that are explored is when Equation is independent of the lattice size. This value is known as $g*$. What is the value that the simulation extracts.The binder cumulant will be calculated with lattice sizes 8, 12, 16 where the function will be the following, $g(T, 8), g(T, 12), g(T, 16)$ with the temperature ranging from $\frac{K_B T}{J} \in [2.0...2.5]$.

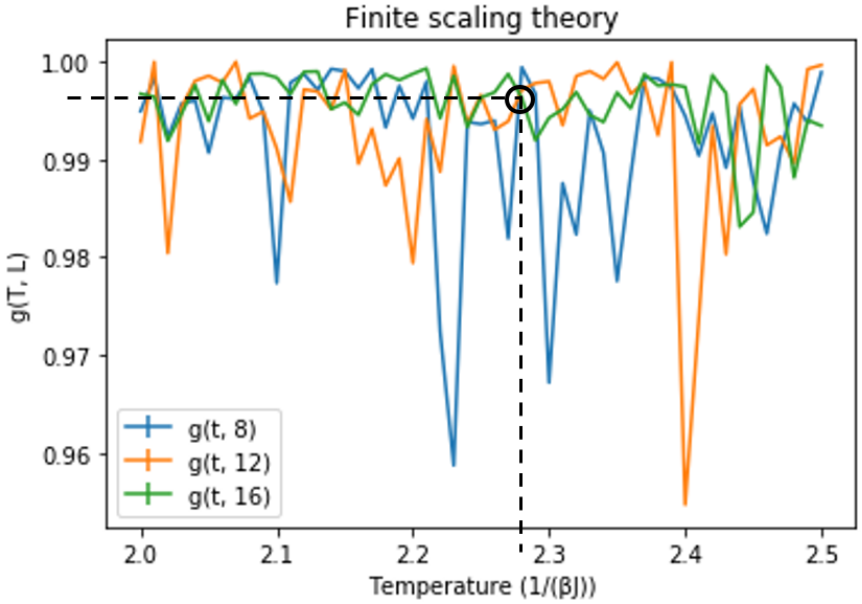

Figure 25: Binder Cumulant .

Figure 25 has a circle at the point of where the lines intersect around the critical temperature and lines are made to show the approximate x and y coordinate. The Binder Cumulant is independent of size at the critical temperature and this is shown using the output figure 25, as the function is all equal at the temperature T_c. The approximate temperature where the function is equal for the simulation is at a temperature of $\frac{K_B T}{J} \approx 2.27 - 2.28$. This is reasonably close to the actual known critical temperature of $T_c = 2.2691853$. The $g*$ value found in the simulation is approximately $g(T_c) \approx 0.996$. This is largely dependant on magnetization and the size of the lattice to create an accurate trend. This simulation had 1 warm up step, a measurement at every step and 5 measurements.
All of the data found for this paper is input into a textfile which python then accesses the data and plots using the matplotlib.pyplot library.

3 Conclusion

This report and assignment was very insightful as multiple methods and models were explored. Various Monte Carlo methods were used and these method's concepts can be applied and modified to complete much more complex simulations and scientific research.

4 Code Appendix

4.1 Code Q1

```
// 4G03_assign2_q1.cpp : This file contains the 'main' function. Program execution begins and ends th
//
#include <iostream>
#include <math.h>
#include<stdio.h>
#include <iomanip>
#include <ctime>
#include <vector>
#include <random>
using namespace std;
//uniform random double between 0 and 1
double uniform_rand() {
        uniform_real_distribution<> u(0, 1);

        random_device r; // Seed with a real random device
        seed_seq seed{ r(), r(), r(), r(), r(), r(), r(), r() };
        mt19937 engine2(seed);
        return u(engine2);
}
//practice

//using CDF for non uniform random number
double non_uniform_rand() {
        double r,y;
        //quantile function
        y = -1 * log(1 - uniform_rand());
        //CDF - inverse quantile
        r = (1 - exp(-y));

        return r;
}
//Box-Muller transformation
double non_uniform_rand2() {
        double r, pi = atan(1) * 4;
        r = abs(cos(2 * pi * uniform_rand())) * sqrt(-2 * log(uniform_rand()));
        while (r > 1) {
                r -= 1;
        }
        return r;
}//end of practice

//initial function
double myfunc(double x)
{
        double r;
        r = exp(-(x * x));
        return r;
}
//probability density
double density(double x)
{
        double r;
        r = (exp(-x)*exp(1))/(exp(1)-1);
```

```
        return r;
}
//CDF
double normal(double x) {
        double r;
        r = exp(1) * (1 - exp(-x)) / (exp(1) - 1);
        return r;
}
//Quantile function
double inv_normal(double x) {
        double r;
        r = -log(1-x*(exp(1)-1)/exp(1));
        return r;
}
//importance sampling function
double new_func(double x) {
        double r;
        r = myfunc(inv_normal(x))/ density(inv_normal(x));
        return r;
}
//monte carlos function
double MC(double (*f)(double x),double N,double b, double a) {
        double function1;
        double ds = 0;
        double sum=0,avgf;

        for (int n = 0; n < N; ++n) {
                // comment and uncomment next 2 lines to use uniform or non uniform random numbers
                 function1 = f(a + (b - a) * uniform_rand());
                 //function1=f(a+(b-a)*non_uniform_rand2()); //experimental used to understand concep
                 sum += function1;//summation for monte carlos integration
                 ds += function1*function1; //used for error
        }
        sum = (b-a)*sum / N;//area under the curve
        avgf = sum / (b - a);//expectation value
        ds = ds/N;//function squared expectation value
        ds = (b - a) * sqrt( abs(ds-pow(avgf,2))/N);//error
        //cout << "Integral using Monte Carlo method evaluates to: " << setprecision(15) << sum << "
        cout << "Integral using Importance Sampling evaluates to: " << setprecision(15) << sum << " +
        //cout << "Integral using non-uniform random numbers and crude Monte Carlo integral: " << set
        //cout << "Integral using Box Muller: " << setprecision(15) << sum << " +- " << setprecision(
        return 0;
}
int main()
{
        double b = 10, a = 0;
        //MC(&myfunc, 1000000, b, a);// using crude monte carlo method
        MC(&new_func, 1000000, normal(b), normal(a)); //using importance sampling

}
```

4.2 Code Q2, For magnetization histogram

```
// Monte Carlo measurement at every flip.cpp: This file contains the 'main' function. Program executi
//

#include <iostream>
#include <math.h>
#include<stdio.h>
#include <iomanip>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
#include <iterator>
#include <fstream>
using namespace std;


//outputss uniform random number
double uniform_rand() {
        uniform_real_distribution<> u(0, 1);

        random_device r; // Seed with a real random device
        seed_seq seed{ r(), r(), r(), r(), r(), r(), r(), r() };
        mt19937 engine2(seed);
        return u(engine2);
}

//uses random number to output a random spin value with higher probability of negative
double spin() {
        double val;
        if (uniform_rand() >= 0.75) {
                val = 1;
        }
        else {
                val = -1;
        }
        return val;
}

//creates lattice full of randomly arranged spin 75% negative 25% positive probability density
vector<vector<double>> Matrix(double L) {
        vector<vector<double>> v;

        for (int i = 0; i < L; ++i) {

                v.push_back(vector<double>());
                for (int j = 0; j < L; ++j) {
                        v[i].push_back(spin());
                }

        }

        return v;
}
```

```
//individual particle calculation in lattice using lattice matrix, x coordinate and y coordinate as p
double Energy(vector<vector<double>> lattice, double x1,double y1) {
        double L = lattice.size() - 1,//due to indexing
                xsum,ysum,sum,//using x,y components to solve for net nearist neighbours energy
                x=x1-1,y=y1-1,//indexing actual value
                xm=x1-2,ym=y1-2;//1 less than actual coordinate
        //boundary conditions
        if (x==0) {//if leftmost atom then nearest nearbour sum in x is spin of atom times atom to ri
                xsum = lattice[y][x1]* lattice[y][x];
        }
        else if(x==L){//if rightmost atom then xsum is atom times atom left to it
                xsum = lattice[y][xm] * lattice[y][x];
        }
        else {//any other atom in lattice, xsum is spin times both left and right atom summed togethe
                xsum = lattice[y][x1] * lattice[y][x] + lattice[y][xm] * lattice[y][x];
        }
        if (y == 0) {//upper most atom then ysum is atom times atom below it
                ysum = lattice[y1][x] * lattice[y][x];
        }
        else if (y == L) {//bottom atom then ysum is atom times atom above it
                ysum = lattice[ym][x] * lattice[y][x];
        }
        else {//any other atom in lattice y, atom times sum of atom above and below it
                ysum = lattice[y1][x] * lattice[y][x] + lattice[ym][x] * lattice[y][x];
        }
        sum = (xsum + ysum);//total energy of atom is atom below and above
        return sum;
}
//solves each atom in lattice energy and returns list for each atom energy
vector<double> particles_energy(vector<vector<double>> lattice){
        double size = lattice.size(),E;
        vector<double> E_list = {};
        //for loop references atoms using lattice size and not actual index, therefore indexing is do
        for (double i = 1; i <= size; ++i) {
                for (double j = 1; j <= size; ++j) {
                        E = Energy(lattice, j, i);

                        //cout << E<<"    ";
                        E_list.push_back(E);
                }
        }
        return E_list;
}
//magnitization function storing each spin in lattice into a vector
vector<double> magnetization(vector<vector<double>> lattice) {
        double size = lattice.size(), M;
        vector<double> M_list = {};
        for (double i = 0; i < size; ++i) {
                for (double j = 0; j < size; ++j) {
                        M = lattice[j][i];
                        M_list.push_back(M);
                }
        }

        return M_list;
```

```
}
//sum of each particles energy or magnetization depending on the 1 D vector passed through parameter.
double system_energy(vector<double> IE) {
        double size = IE.size(),totalE=0;
        for (int i = 0; i < size; i++) {
                totalE += IE[i];
        }
        return (-1*totalE);// default is energy but if magnetization then multiply output by negative
}

pair<vector<double>, vector<double>> MCSweeps(vector<vector<double>> lattice, double Nmcs, double BJ)
        pair<vector<double>, vector<double>> net_E;//output for function
        //variables needed
        double x,
                y,
                initial_spin,
                dE,
                final_spin,energy,
                L = lattice.size() - 1;
        //mutable matrix
        vector<vector<double>> spin_arrangement = lattice;
        //runs for loop for amount of monte carlo sweeps
        for (double i = 0; i < Nmcs; i++) {
                double E_i = 0, E_f = 0; //continously reset initial energy and final energy

                //retrieve a random x and y coordinate in the range of the length of the LxL matrix
                x = round(L * uniform_rand());
                y = round(L * uniform_rand());
                //initial spin is the value of the lattice at this coordinate
                initial_spin = spin_arrangement[y][x];
                //calculate initial energy
                E_i = system_energy(particles_energy(spin_arrangement));


                //possible flipped state which would be final state
                final_spin = -1 * initial_spin;

                //setting the spin at lattice coordinate to flipped state
                spin_arrangement[y][x] = final_spin;

                //calculating lattice energy considering flipped spin
                E_f = system_energy(particles_energy(spin_arrangement));


                //reset coordinate to original state
                spin_arrangement[y][x] = initial_spin;
                //change in energy associated with flipping the state
                dE = E_f - E_i;
                //generate random number and if random number less than boltzman probability and flip
                if (E_f>E_i && uniform_rand() < exp(-BJ*dE)) {
                        spin_arrangement[y][x]= final_spin;
                        energy = system_energy(particles_energy(spin_arrangement));
                }
                else if (E_i > E_f) {//if initial energy greater, then flip state, find new lattice e
                        spin_arrangement[y][x] = final_spin;
                        energy = system_energy(particles_energy(spin_arrangement));
```

```
                }
                else {//any other situation, keep initial state and find energy
                        spin_arrangement[y][x] = initial_spin;
                        energy = system_energy(particles_energy(spin_arrangement));
                }
                net_E.first.push_back(energy);//total energy of lattice at each sweep stored in vecto
                net_E.second.push_back(-1*system_energy(magnetization(spin_arrangement)));//total mag
        }


        return net_E;
}
//return average of any vector eg. energy
//considers power of arguement for expectation value
double avg(vector<double> list,double power) {
        double size = list.size(),
                avg=0;
        for (double i = 0; i < size; i++) {
                avg =avg+ pow(list[i],power);
        }
        avg = avg/size;
        return avg;
}
//error calculation
double avg_err(vector<double> list, double power) {
        double N = list.size();
        double error = sqrt(abs(avg(list, 2 * power) - pow(avg(list, power), 2)) / N);
        return error;
}
//out puts X and heat capacity at various temperatures
pair<vector<double>, vector<double>>CP(double L) {
        //creates lattice
        vector<vector<double>> v = Matrix(L);
        pair<vector<double>, vector<double>> cp_x;
        double order1 = 1, order2 = 2;//order required to solve for
        for (double T = 2.0; T <= 2.5; T += 0.025) {//various temperatures

                double BJ = pow(T, -1);//beta is 1/kt
                auto E = MCSweeps(v, pow(L,2)+5*L, BJ);//preforms mc sweeps algorithm
                vector<double> Energy = (E.first);//energy vector
                vector<double> Magnetization = (E.second);//magnetization vector

                //expectation vaues
                double ExpE1 = avg(Energy, order1),
                        ExpE2 = avg(Energy, order2),
                        ExpM2 = avg(Magnetization, order2);
                //create vector of susceptibility
                cp_x.first.push_back(ExpM2 * BJ / pow(L, 2));
                //create vector of heat capacity
                cp_x.second.push_back(pow(BJ, 2) * (ExpE2 - pow(ExpE1,2)) / (pow(L, 2)));

        }
        return cp_x;
}
//function for finite scaling theory
vector<double>G(double L) {
```

```
        vector<double>G_;
        vector<vector<double>> v = Matrix(L);
        double M2, M4, order2 = 2, order4 = 4;
        for (double T = 2.0; T <= 2.5; T += 0.01) {
                double BJ = pow(T, -1);
                auto E = MCSweeps(v, pow(L,2)+5*L, BJ);
                vector<double> Magnetization1 = (E.second);
                M2 = avg(Magnetization1, order2),
                M4 = avg(Magnetization1, order4),
                G_.push_back(0.5 * (3 - M4 / pow(M2, 2)));
        }
        return G_;
}//function for avg energy
vector<double>EE() {
        vector<vector<double>> v = Matrix(10);
        vector<double>E1;

        for (double T = 2.0; T <= 2.5; T += 0.025) {
                double BJ = pow(T, -1);
                auto E = MCSweeps(v, pow(10, 2)+50, BJ);
                vector<double> Energy = (E.first);
                E1.push_back(avg(Energy,1));
        }
        return E1;
}//output data that is in respect to monte carlo time
double output_data_time(vector<double> M, string filename) {
        double L = M.size(), error = avg_err(M, 1);
        ofstream MyFile(filename);
        for (int j = 0; j < L; j++) {
                MyFile << j + 1 << "," << M[j] << "," << error << endl;

        }
        MyFile.close();
        return 0;
}//output data in respect to chane in time
double output_data_temp(vector<double> M, string filename) {
        double L = M.size(), error = avg_err(M, 1);
        double count = 0;
        ofstream MyFile(filename);
        for (double j = 2.0; j <= 2.5; j = j + 0.01) {
                MyFile << j << "," << M[count] << "," << error << endl;
                count += 1;
        }
        MyFile.close();
        return 0;
}
int main()
{
        double size = 50;
        vector<vector<double>> v = Matrix(size);

        auto E = MCSweeps(v, 3000, 1/2.26);
        vector<double> Energy = (E.first);

        vector<double> Magnetization = (E.second);
        output_data_time(Magnetization,"Magnetization@226.txt");
```

```
        auto E1 = MCSweeps(v, 3000, 1 / 2.45);
        vector<double> Energy1 = (E1.first);
        vector<double> Magnetization1 = (E1.second);
        output_data_time(Magnetization1, "Magnetization@245.txt");

}
```

4.3 Code Q2, For rest of data

```
// Metropolis.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>
#include <math.h>
#include<stdio.h>
#include <iomanip>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
#include <iterator>
#include <fstream>
using namespace std;


//outputss uniform random number
double uniform_rand() {
	uniform_real_distribution<> u(0, 1);

	random_device r; // Seed with a real random device
	seed_seq seed{ r(), r(), r(), r(), r(), r(), r(), r() };
	mt19937 engine2(seed);
	return u(engine2);
}
//uses random number to output a random spin value with higher probability of negative
double spin() {
	double val;
	if (uniform_rand() >= 0.75) {
		val = 1;
	}
	else {
		val = -1;
	}
	return val;
}
//creates lattice full of randomly arranged spin 75% negative 25% positive probability density
vector<vector<double>> Matrix(double L) {
	vector<vector<double>> v;

	for (int i = 0; i < L; ++i) {

		v.push_back(vector<double>());
		for (int j = 0; j < L; ++j) {
			v[i].push_back(spin());
		}
	}
	return v;
}
//return average of any vector eg. energy
//considers power of arguement for expectation value
double avg(vector<double> list, double power) {
	double size = list.size(),
		avg = 0;
	for (double i = 0; i < size; i++) {
```

```
                avg = avg + pow(list[i], power);
        }
        avg = avg / size;
        return avg;
}
//error calculation
double avg_err(vector<double> list, double power) {
        double N = list.size();
        double error = sqrt(abs(avg(list, 2 * power) - pow(avg(list, power), 2)) / N);
        return error;
}

//individual particle calculation in lattice using lattice matrix, x coordinate and y coordinate as p
double Energy(vector<vector<double>> lattice, double x1, double y1) {
        double L = lattice.size() - 1,//due to indexing
                xsum, ysum, sum,//using x,y components to solve for net nearist neighbours energy
                x = x1 - 1, y = y1 - 1,//indexing actual value
                xm = x1 - 2, ym = y1 - 2;//1 less than actual coordinate
        //boundary conditions
        if (x == 0) {//if leftmost atom then nearest nearbour sum in x is spin of atom times atom to
                xsum = lattice[y][x1] * lattice[y][x];
        }
        else if (x == L) {//if rightmost atom then xsum is atom times atom left to it
                xsum = lattice[y][xm] * lattice[y][x];
        }
        else {//any other atom in lattice, xsum is spin times both left and right atom summed togethe
                xsum = lattice[y][x1] * lattice[y][x] + lattice[y][xm] * lattice[y][x];
        }
        if (y == 0) {//upper most atom then ysum is atom times atom below it
                ysum = lattice[y1][x] * lattice[y][x];
        }
        else if (y == L) {//bottom atom then ysum is atom times atom above it
                ysum = lattice[ym][x] * lattice[y][x];
        }
        else {//any other atom in lattice y, atom times sum of atom above and below it
                ysum = lattice[y1][x] * lattice[y][x] + lattice[ym][x] * lattice[y][x];
        }
        sum = (xsum + ysum);//total energy of atom is atom below and above
        return sum;
}
//solves each atom in lattice energy and returns list for each atom energy
vector<double> particles_energy(vector<vector<double>> lattice) {
        double size = lattice.size(), E;
        vector<double> E_list = {};
        //for loop references atoms using 1 -lattice size and not actual index, therefore indexing is
        for (double i = 1; i <= size; ++i) {
                for (double j = 1; j <= size; ++j) {
                        E = Energy(lattice, j, i);

                        //cout << E<<"    ";
                        E_list.push_back(E);
                }

        }

        return E_list;
```

```
}
//magnitization function storing each spin in lattice into a vector
vector<double> magnetization(vector<vector<double>> lattice) {
        double size = lattice.size(), M;
        vector<double> M_list = {};
        for (double i = 0; i < size; ++i) {
                for (double j = 0; j < size; ++j) {
                        M = lattice[j][i];
                        M_list.push_back(M);
                }
        }

        return M_list;

}
//sum of each particles energy or magnetization depending on the 1 D vector passed through parameter.
double system_energy(vector<double> IE) {
        double size = IE.size(), totalE = 0;
        for (int i = 0; i < size; i++) {
                totalE += IE[i];
        }
        return (-1 * totalE);// default is energy but if magnetization then multiply output by negati
}

vector<vector<double>> MCSweeps(vector<vector<double>> lattice, double Nmcs, double BJ) {
        pair<vector<double>, vector<double>> net_E;//output for function
        //variables needed
        double x, y,
                initial_spin, dE,
                final_spin, energy,
                L = lattice.size() - 1;
        //mutable matrix
        vector<vector<double>> spin_arrangement = lattice;
        //runs for loop for amount of monte carlo sweeps
        for (double o = 0; o < Nmcs; o++) {
                for (double i = 0; i < ((L+1)*(L+1)); i++) {
                        double E_i = 0, E_f = 0; //continously reset initial energy and final energy

                        //retrieve a random x and y coordinate in the range of the length of the LxL
                        x = round(L * uniform_rand());
                        y = round(L * uniform_rand());
                        //initial spin is the value of the lattice at this coordinate
                        initial_spin = spin_arrangement[y][x];
                        //calculate initial energy
                        E_i = system_energy(particles_energy(spin_arrangement));

                        //possible flipped state which would be final state
                        final_spin = -1 * initial_spin;

                        //setting the spin at lattice coordinate to flipped state
                        spin_arrangement[y][x] = final_spin;

                        //calculating lattice energy considering flipped spin
                        E_f = system_energy(particles_energy(spin_arrangement));
```

```
                        //reset coordinate to original state
                        spin_arrangement[y][x] = initial_spin;
                        //change in energy associated with flipping the state
                        dE = E_f - E_i;
                        //generate random number and if random number less than boltzman probability
                        //state energy is greater than initial state, flip the state evalutate new en
                        if (E_f > E_i && uniform_rand() < exp(-BJ * dE)) {
                                spin_arrangement[y][x] = final_spin;
                        }
                        else if (E_i > E_f) {//if initial energy greater, then flip state, find new l
                                spin_arrangement[y][x] = final_spin;
                        }
                        else {//any other situation, keep initial state and find energy
                                spin_arrangement[y][x] = initial_spin;
                        }
                }
        }
        return spin_arrangement;
}
pair <vector<double>, vector<double>>SQIsing(double NWarmup, double NStep,double Nmeas, double L,doub
        vector<vector<double>> v = Matrix(L);
        pair<vector<double>, vector<double>> net_E;
        vector<vector<double>>new_lattice = MCSweeps(v, NWarmup,BJ);
        for (double i = 0; i < Nmeas; i++) {
                new_lattice=MCSweeps(new_lattice, NStep, BJ);
                net_E.first.push_back(system_energy(particles_energy(new_lattice)));//total energy of
                net_E.second.push_back(-1 * system_energy(magnetization(new_lattice)));//total magnet

        }
        return net_E;
}

//out puts X and heat capacity at various temperatures
pair<vector<double>, vector<double>>CP(double L) {
        //creates lattice
        vector<vector<double>> v = Matrix(L);
        pair<vector<double>, vector<double>> cp_x;
        double order1 = 1, order2 = 2;//order required to solve for
        for (double T = 2.0; T <= 2.5; T += 0.01) {//various temperatures

                double BJ = pow(T, -1);//beta is 1/kt
                auto E = SQIsing(7,1,15,  L, BJ);//preforms mc sweeps algorithm
                vector<double> Energy = (E.first);//energy vector
                vector<double> Magnetization = (E.second);//magnetization vector

                //expectation vaues
                double ExpE1 = avg(Energy, order1),
                        ExpE2 = avg(Energy, order2),
                        ExpM2 = avg(Magnetization, order2);
                //create vector of susceptibility
                cp_x.first.push_back(ExpM2 * BJ / pow(L, 2));
                //create vector of heat capacity
                cp_x.second.push_back(pow(BJ, 2) * (ExpE2 - pow(ExpE1, 2)) / (pow(L, 2)));

        }
        return cp_x;
```

```
}
//function for finite scaling theory
vector<double>G(double L) {
        vector<double>G_;
        vector<vector<double>> v = Matrix(L);
        double M2, M4, order2 = 2, order4 = 4;
        for (double T = 2.0; T <= 2.5; T += 0.01) {
                double BJ = pow(T, -1);
                auto E = SQIsing(1 ,1 ,5, L, BJ);
                vector<double> Magnetization1 = (E.second);
                M2 = avg(Magnetization1, order2),
                M4 = avg(Magnetization1, order4);
                G_.push_back(0.5 * (3 - M4 / pow(M2, 2)));
        }
        return G_;
}//function for avg energy
vector<double>EE(double L) {
        vector<vector<double>> v = Matrix(L);
        vector<double>E1;

        for (double T = 2.0; T <= 2.5; T += 0.01) {
                double BJ = pow(T, -1);
                auto E = SQIsing(5,1,15, L, BJ);
                vector<double> Energy = (E.first);
                E1.push_back(avg(Energy, 1));
        }
        return E1;
}

//output data that is in respect to monte carlo time
double output_data_time(vector<double> M, string filename) {
        double L = M.size(), error = avg_err(M, 1);
        ofstream MyFile(filename);
        for (int j = 0; j < L; j++) {
                MyFile << j + 1 << "," << M[j] << "," << error << endl;

        }
        MyFile.close();
        return 0;
}//output data in respect to chane in time
double output_data_temp(vector<double> M, string filename) {
        double L = M.size(), error = avg_err(M, 1);
        double count = 0;
        ofstream MyFile(filename);
        for (double j = 2.0; j <= 2.5; j = j + 0.01) {
                MyFile << j << "," << M[count] << "," << error << endl;
                count += 1;
        }
        MyFile.close();
        return 0;
}

int main()
{
        double size = 50;
        auto E = SQIsing(1,1,15, size, 1/2.26);
```

```
        vector<double> Energy = (E.first);

        vector<double> Magnetization = (E.second);
        output_data_time(Magnetization,"Magnetization@226.txt");
        auto E1 = SQIsing(1,1,15, size, 1/2.45);
        vector<double> Energy1 = (E1.first);
        vector<double> Magnetization1 = (E1.second);
        output_data_time(Magnetization1, "Magnetization@245.txt");

        output_data_temp(EE(10), "Avg E @ various temps.txt");
        auto cp_x = CP(10);
        vector<double>X = cp_x.first;
        vector<double>cp = cp_x.second;
        output_data_temp(X, "X @ various temps.txt");
        output_data_temp(cp, "Heat Capacity @ various temps.txt");/**/
        output_data_temp(G(8), "G@8.txt");
        output_data_temp(G(12), "G@12.txt");
        output_data_temp(G(16), "G@16.txt");
}
```

4.4 Python Plotting for Magnetization Histogram

```
import matplotlib.pyplot as plt
import numpy as np

X1, Y1,error1= np.loadtxt('Magnetization@226.txt', delimiter=',', unpack=True)
X2, Y2,error2= np.loadtxt('Magnetization@245.txt', delimiter=',', unpack=True)

plot1=plt.figure(1)
plt.hist(Y1)
plt.title('Magnetization at 1/(J)=2.26')
plt.xlabel('Magnetization')
plt.ylabel('Frequency')
plot2=plt.figure(2)
plt.hist(Y2)
plt.title('Magnetization at 1/(J)=2.45')
plt.xlabel('Magnetization')
plt.ylabel('Frequency')
```

4.5 Python Plotting for rest of data

```
import matplotlib.pyplot as plt
import numpy as np

X3, Y3,error3= np.loadtxt('Avg E @ various temps.txt', delimiter=',', unpack=True)
X4, Y4,error4= np.loadtxt('X @ various temps.txt', delimiter=',', unpack=True)
X5, Y5,error5= np.loadtxt('Heat Capacity @ various temps.txt', delimiter=',', unpack=True)
X6, Y6,error6= np.loadtxt('G@8.txt', delimiter=',', unpack=True)
X7, Y7,error7= np.loadtxt('G@12.txt', delimiter=',', unpack=True)
X8, Y8,error8= np.loadtxt('G@16.txt', delimiter=',', unpack=True)

fig3, ax = plt.subplots()
m3, b3 = np.polyfit(X3, Y3, 1)
ax.errorbar(X3, Y3,label='Energy Expectation Values',
            yerr=error3,
            capsize=4)
plt.plot(X3, m3*X3 + b3,label='Energies line of best fit')
plt.legend()
plt.title('Avg Energy changing temperature from 1/(J)=2.0-2.5')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('Avg Energy (E/J)')


fig4, ax = plt.subplots()
m4, b4 = np.polyfit(X4, Y4, 1)
ax.errorbar(X4, Y4,label='Magnetic Susceptibility',
            yerr=error4,
            capsize=4)
plt.plot(X4, m4*X4 + b4,label='Magnetic Susceptibility line of best fit')
plt.legend()
plt.title('Magnetic susceptibility changing temperature from 1/(J)=2.0-2.5')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('Susceptibility ($\chi$ $J/K_B$)')


fig5, ax = plt.subplots()
a5,m5, b5 = np.polyfit(X5, Y5, 2)
ax.errorbar(X5, Y5,label='Heat Capacity',
            yerr=error5,
            capsize=4)
plt.plot(X5,a5*X5**2+m5*X5 + b5,label='Heat Capacity line of best fit')
plt.legend()
plt.title('Heat Capacity changing temperature from 1/(J)=2.0-2.5')
plt.xlabel('Temperature ((1/(J))')
plt.ylabel('Heat Capacity $(C_V / k^2$)')


fig6, ax = plt.subplots()
plt.title('Finite scaling theory')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('g(T, L)')
ax.errorbar(X6, Y6, label='g(t, 8)',
            yerr=0,#error6,
            capsize=0)#4)
```

```
ax.errorbar(X7, Y7, label='g(t, 12)',
            yerr=0,#error7,
            capsize=0)#4)



ax.errorbar(X8, Y8, label='g(t, 16)',
            yerr=0,#error8,
            capsize=0)#4)
plt.legend()
plt.show()


X9=(X3[0:28])
Y9=(Y3[0:28])
X10=(X3[27:52])
Y10=(Y3[27:52])
fig7, ax = plt.subplots()
plt.title('Verify critical temperature from energy')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('Avg Energy (E/J)')
a9,m9, b9 = np.polyfit(X9, Y9, 2)
ax.errorbar(X9, Y9,
            yerr=error3[0:28],
            capsize=4)#4)
plt.plot(X9,a9*X9**2+m9*X9 + b9,label='Energy line of best fit before $T_c$')
a10,m10, b10 = np.polyfit(X10, Y10, 2)
ax.errorbar(X10, Y10,
            yerr=error3[27:52],
            capsize=4)#4)
plt.plot(X10,a10*X10**2+m10*X10 + b10,label='Energy line of best fit after $T_c$')
plt.legend()
plt.show()

X11=(X4[0:28])
Y11=(Y4[0:28])
X12=(X4[27:52])
Y12=(Y4[27:52])
fig8, ax = plt.subplots()
plt.title('Verify critical temperature from susceptibility')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('Susceptibility ($\chi$ $J/K_B$)')
a11,m11, b11 = np.polyfit(X11, Y11, 2)
ax.errorbar(X11, Y11,
            yerr=error4[0:28],
            capsize=4)#4)
plt.plot(X11,a11*X11**2+m11*X11 + b11,label='Susceptibility line of best fit before $T_c$')
a12,m12, b12 = np.polyfit(X12, Y12, 2)
ax.errorbar(X12, Y12,
            yerr=error4[27:52],
            capsize=4)#4)
plt.plot(X12,a12*X12**2+m12*X12 + b12,label='Susceptibility line of best fit after $T_c$')
plt.legend()
plt.show()

X13=(X5[0:28])
```

```
Y13=(Y5[0:28])
X14=(X5[27:52])
Y14=(Y5[27:52])
fig9, ax = plt.subplots()
plt.title('Verify critical temperature from Heat Capacity')
plt.xlabel('Temperature (1/(J))')
plt.ylabel('Heat Capacity $(C_V / k^2$)')
q13,a13,m13, b13 = np.polyfit(X13, Y13, 3)
ax.errorbar(X13, Y13,
            yerr=error5[0:28],
            capsize=4)#4)
plt.plot(X13,q13*X13**3+a13*X13**2+m13*X13 + b13,label='Heat Capacity line of best fit before $T_c$')
a14,m14, b14 = np.polyfit(X14, Y14, 2)
ax.errorbar(X14, Y14,
            yerr=error5[27:52],
            capsize=4)#4)
plt.plot(X14,a14*X14**2+m14*X14 + b14,label='Heat Capacity line of best fit after $T_c$')
plt.legend()
plt.show()
```

References

[1] C. Mcelfresh, "Monte Carlo Integration," Medium, 13-Jan-2021. [Online]. Available: https://cameron-mcelfresh.medium.com/monte-carlo-integration-313b37157852. [Accessed: 19-Oct-2021]. Wesley, Massachusetts, 2nd ed.

[2] P. Abeyasinghe, "Figure 12: Experimental Data and model ... - researchgate.net." [Online]. Available: https://www.researchgate.net/figure/Magnetization-susceptibility-and-Energy-as-a-function-of-temperature-for-the-Classical$_fig8_3$15730005. [Accessed: 22-Oct-2021].

[3] R. Fitzpatrick, The Ising model, 29-Mar-2006. [Online]. Available: https://farside.ph.utexas.edu/teaching/329/lectures/node110.html. [Accessed: 22-Oct-2021].