Simulating a One-Dimensional Heisenberg Model

Nathan Halsey

Winter 2020

1 Introduction

1.1 Description of Model

The Heisenberg model is a way of simulating a ferromagnetic system, and is a simplified model of magnestism. It is closely related to the Ising model, with the major difference being that the spin states are represented by a three-dimensional vector rather than a simple up-down state as seen in the Ising model. It is seen as an extension of the Ising model, and has been used to solve various problems in physics, including physical problems dealing with quarks. In this paper I follow a paper written by C.G Windsor et al (Referenced in section "References") and try to recreate the same algorithm. I also do further testing on various data as outlined in the "Computational details" section of that paper.

1.2 Mathematical Formulation of Model

In this model, I simulate a one dimensional Heisenberg model, also sometimes called a Heisenberg Chain, or Heisenberg Spin Chain. The Hamiltonian of this one dimensional Heisenberg model is:

$$\hat{H} = -J \sum_{i=1}^{N} \mu_i \cdot \mu_{i+1} - h \sum_{i=1}^{N} \mu_i \tag{1}$$

where J is the coupling constant, and the sum is over the nearest neighbours of i. I also use the Metropolis algorithm, as described in the paper "Equation of State Calculations by Fast Computing Machines", by Metropolis et al. Specifically, we will be modelling the Heisenberg XXX model, with no magnetic field. Thus, we can simplify and expand our Hamiltonian to:

$$\hat{H} = -1/2 \sum_{i=1}^{N} (J_x \mu_i^x \mu_{i+1}^x + J_y \mu_i^y \mu_{i+1}^y + j_z \mu_i^z \mu_{i+1}^z) \tag{2}$$

And, since we are using the Heisenberg XXX model, this can be further simplified to:

$$\hat{H} = -1/2J \sum_{i=1}^{N} (\mu_i^x \mu_{i+1}^x + \mu_i^y \mu_{i+1}^y + \mu_i^z \mu_{i+1}^z) \tag{3}$$

This is because in the XXX model, $J_x = J_y = J_z$.

1.3 Aims of this Paper

The aim of this paper is to describe how to implement the model, and explore what changing various constants in the equations will do to our final result.

2 Implementation and Simulation

2.1 Formulation of Simulation

Consider a chain with n spin sites, with n being some finite number, and each spin site occupied by a spin μ^n. These spins are represented by a three-dimensional unit vector. To simplify things, we have no external magnetic field acting on this system, so our Hamiltonian can also be simplified to:

$$\hat{H} = -1/2J \sum_{i=1}^{N} \mu_i \cdot \mu_{i+1} \tag{4}$$

Initially, we choose a random spin configuration, and then begin making trial spins. The energy of our system is equal to our Hamiltonian, dictated in (2). If $\Delta E < 0$ (the change in energy of the system after the trial spin is performed) then the trial spin is accepted. If $\Delta E > 0$, then the trial spin is accepted if and only if some transition probability is satisfied.

Our choice of transition probability will follow $T(S_i \to S_j)$ given by

$$T(S_i \to s_j) = \frac{e^{-\Delta E/kT}}{1 + e^{-\Delta E/kT}} \tag{5}$$

We also need to change the spin direction carefully. We would like our model to be less volatile and be able to solve problems effectively. We want our spin change to be symmetrically distributed around the current spin direction to avoid this volatility.

2.2 The Algorithm

The algorithm for computing the total energy of our system can be executed in as little as 5 steps.

1. Create a random spin configuration for our chain. I did this by creating random x, y, z coordinates and then normalising the vector to a unit vector. This is done by:

$$\overrightarrow{V} = \frac{\overrightarrow{V}}{|\overrightarrow{V}|} \tag{6}$$

2. The energy of the system is calculated using our Hamiltonian in (2)

3. We compute a Monte Carlo move by generating a vector

$$(p_1, p_2, p_3) \tag{7}$$

where $p_n \in (-1, 1)$. In this case, I randomised these values, using a random() function. We then multiply this by a factor ΔS_{max}. We do this to limit the maximum change of any spin, and keep it symmetrically distributed around the current spin direction. If

$$|\Delta S| > \Delta S_{max} \tag{8}$$

we compute a new ΔS in the same way. I do this using an iterative process, calling the function again if (8) is not satisfied.

4. We do a trial spin on every node in the chain. This trial spin is only accepted if $\Delta E < 0$ or if $\Delta E > 0$ and the transition probability in (5) is satisfied.

5. We store the current move number, as well as the total energy of the system, as we're primarily interested in convergence rates in this paper.

Step 3 utilises the Metropolis Algorithm (Equation of State Calculations by Fast Computing Machines, Metropolis et al, 1953).

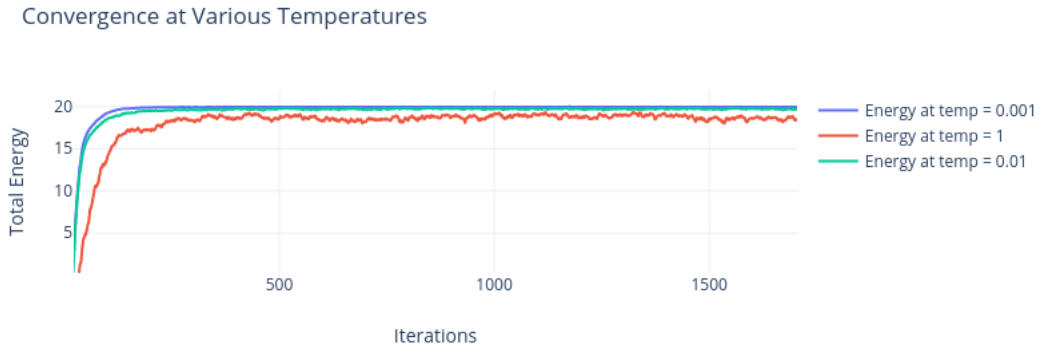
2.3 What our Model Solves

Our model solves a lot of practical and theoretical problems in statistical mathematics and physics. However, the objective of our model is instead to investigate what various properties the model has, and to play with some of the data. Particularly we are interested in seeing what a change in S_{max} does, as well as what changing the temperature will do to our model. S_{max} gives us mathematical insight, while changing temperatures gives us a physical insight. The mathematical insight is important to see why we have the stipulation on S_{max} and why it constitutes a Monte Carlo move.

2.4 Results

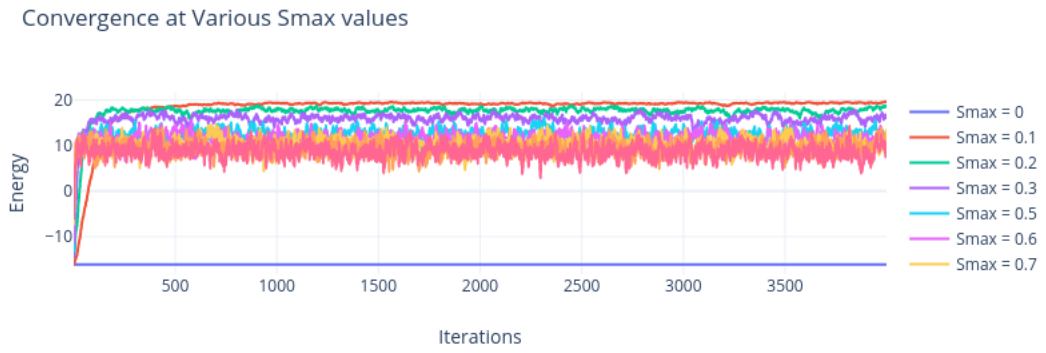
2.4.1 Temperature and Convergence

The total energy converges very quickly with a temperature between 0 and 1, and at high temperatures, converges more slowly.



2.4.2 Reducing ΔS_{max}

A more interesting thing to explore is reducing the value of ΔS_{max} used in equation (8), which will change the amount of spin our node is subject to during each iteration.

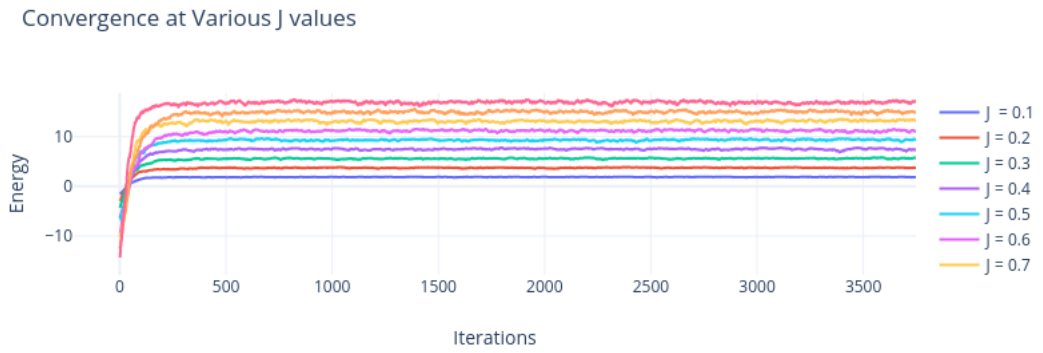


As we can see from the graph, with a low S_{max}, between $(0, 0.1)$, we have a less volatile model and it converges to a higher energy state.

This is an important part of this investigation, we have shown that setting an S_{max} as outlined in (8) is an important step in reducing the volatility of our model and having a better understanding of what we are modelling.

2.4.3 Changing our $J_x = J_y = J_z$

Changing our J value does nothing to change the convergence rate of our model. This only scales it to a higher/lower energy state.



3 Comparision to Other Simulations

In the paper "Computer simulation of the dynamics of onedimensional Heisenberg magnets, C G Windsor and JLocke-Wheaton 1976 J. Phys. C: Solid State Phys. 9 2749", comparing my simulation to the simulation used in the beginning stages of this paper, my program is exactly similar. They converge to similar rates. However, this paper uses a different transition probability as

$$T(S_i \to S_j) = e^{-\Delta E/kT} \tag{9}$$

I chose to use a different transition probability, as stated in (5) because it was the one used when we made an Ising model in class. I found that this transition probability would make my model less volatile and easier to manipulate that the transition probability given by the cited paper. The transition probability above is also more easily utilised by the Metropolis Algorithm defined in the paper "Equation of State Calculations by Fast Computing Machines, J. Chem. Phys. 21, 1087 (1953), Metropolis et al". The Metropolis Algorithm states that we should generate a candidate from an acceptence ratio

$$\mu = \frac{f'(x)}{f(x)} \tag{10}$$

which is not satisfied by the transition probability given in the paper, but is satisfied by the transition probability given in (5).

In the textbook, "An Introduction to Computer Simulation Methods Applications to Physical System Harvey Gould, Jan Tobochnik, andWolfgang Christian" very little is given in the way of the Heisenberg model, but it is also what I used to find out that we needed to limit the amount of spin of each node per iteration. If I had not found this resource, my model would have been very volatile, or may have never converged to an equillibrium state.

4 Conclusion

4.1 Final Conclusions

A model was created which utilised Monte Carlo moves and the Metropolis Algorithm to simulate a Heisenberg chain XXX model. The major difference between this model and the Ising model is that the Heisenberg model has spin directions in three-dimensions.

It was found that there is much importance on the step of the algorithm which constitutes how much the node is spun during a Monte Carlo move. We found that if we increase the maximum amount that this node can be spun, we increase the volatility of the system. By reducing the value of S_{max}, we reduce the volatility and get a better representation of our system.

It was also found that the choice of coupling constant has very little effect on the structure

of our model, except for scaling of our model. With a higher coupling, our model has a higher total energy, and with a lower coupling it has a lower total energy. This is as expected, since J is a constant.

Lastly, we found that changing the temperature of our system affected the total energy convergence rate. As the tempature approaches zero, we see the system converging to it's equillibrium much more quickly.

I would have liked to compared this to the Ising model and different rates of convergence, but in reality they solve two different things. The Ising model is usually good enough to solve problems of this type, and the Heisenberg model was developed to investigate phase transition properties.

4.2 Further Research

I would like to do more research on other Heisenberg models, including the XYY model, and the XYZ model. I am also interested in increasing the dimensions and moving from a Heisenberg chain to a lattice and seeing how this affects convergence rates, as well as looking at other problems that can be solved.

5 References

Computer simulation of the dynamics of onedimensional Heisenberg magnets, C G Windsor and J Locke-Wheaton 1976 J. Phys. C: Solid State Phys. 9 2749

Equation of State Calculations by Fast Computing Machines, J. Chem. Phys. 21, 1087 (1953),

Metropolis et al

An Introduction to Computer Simulation Methods Applications to Physical System Harvey Gould, Jan Tobochnik, andWolfgang Christian

Heisenberg Chain

```
#ifndef HEISENBERGHEADERDEF
#define HEISENBERGHEADERDEF

#include <iostream>

class Heisenberg
{
private:
    double** m_chain;
    double* m_TrialSpin;
    int m_nodes;
    int max_iter;
    double k_temp;
    double m_j;
    double m_smin;




public:
    Heisenberg(int nodes=40, double temp=1, double j=1, double s_min=0.2, int max_iter=4000);
    ~Heisenberg();
    double Transition(double d_energy);
    double CalculateEnergy();
    double max(double x, double y, double z);
    double deltaS();
    void MonteCarloMove(double s_max);
    double random(double m, double n);
    void setInitialState();
    void trialSpin(int element);
    void printLines();

};


#endif


#include "Heisenberg.h"
#include <cmath>
#include <fstream>

Heisenberg::Heisenberg(int nodes, double temp, double j, double s_min, int max_iter)
{
    m_nodes = nodes;
    k_temp = temp;
    m_j = j;
    m_smin = s_min;

    m_TrialSpin = new double[3];
    m_chain = new double*[m_nodes];
```

```
    for (int i = 0; i < m_nodes; i ++)
    {
        m_chain[i] = new double[3];
    }

}
Heisenberg::~Heisenberg()
{
    for (int i = 0; i < m_nodes; i ++)
    {
        delete[] m_chain[i];
    }
    delete[] m_chain;
    delete[] m_TrialSpin;
}

double Heisenberg::Transition(double d_energy)
{
    double value;
    double result;
    value = exp(-d_energy/k_temp);
    result = value/(1+value);
    return result;
}
double Heisenberg::CalculateEnergy()
{
    double sum = 0;
    for (int i = 0; i < m_nodes-1; i ++)
    {
        for (int k = 0; k < 3; k ++)
        {
            sum += m_chain[i][k]*m_chain[i+1][k];
        }
    }
    for (int j = 0; j < 3; j++)
    {
        sum += m_chain[m_nodes-1][j]*m_chain[0][j];
    }

    return -0.5*m_j*sum;
}
void Heisenberg::setInitialState()
{


    for (int i = 0; i < m_nodes; i ++)
    {
        double a,b,c;
        a = rand();
        b = rand();
        c = rand();
        double magnitude = 1/(sqrt(pow(a,2)+pow(b,2)+pow(c,2)));
        m_chain[i][0] = a*magnitude;
        m_chain[i][1] = b*magnitude;
        m_chain[i][2] = c*magnitude;
    }
```

```
}

void Heisenberg::MonteCarloMove(double s_max)
{

    double a = random(-1,1);
    double b = random(-1,1);
    double c = random(-1,1);

    a = s_max*a;
    b = s_max*b;
    c = s_max*c;

    if (sqrt(pow(a,2)+pow(b,2)+pow(c,2))>s_max)
    {
        MonteCarloMove(s_max);
    }
    else
    {
        m_TrialSpin[0] = a;
        m_TrialSpin[1] = b;
        m_TrialSpin[2] = c;
    }

}

double Heisenberg::random(double m, double n)
{
    return m + (rand() / ( RAND_MAX / (n-m)));
}
void Heisenberg::trialSpin(int element)
{
    double EnergyOld = CalculateEnergy();



    MonteCarloMove(m_smin);
    double a = m_chain[element][0] + m_TrialSpin[0];
    double b = m_chain[element][1] + m_TrialSpin[1];
    double c = m_chain[element][2] + m_TrialSpin[2];

    double magnitude = 1/sqrt(pow(a,2)+pow(b,2)+pow(c,2));

    a = a * magnitude;
    b = b * magnitude;
    c = c * magnitude;

    double holder_a = m_chain[element][0];
    double holder_b = m_chain[element][1];
    double holder_c = m_chain[element][2];

    m_chain[element][0] = a;
    m_chain[element][1] = b;
    m_chain[element][2] = c;
```

```
    double EnergyNew = CalculateEnergy();

    double deltaEnerg = EnergyOld - EnergyNew;
    double randomNum = random(0,1);

    if (((deltaEnerg) > 0)&&(Transition(deltaEnerg)<randomNum))
    {
        m_chain[element][0] = holder_a;
        m_chain[element][1] = holder_b;
        m_chain[element][2] = holder_c;
    }

}
void Heisenberg::printLines()
{

    setInitialState();
    std::ofstream myFile;
    myFile.open("Project.dat");
    for (int i = 0; i < max_iter; i ++)
    {
        for (int j = 0; j < m_nodes; j++)
        {
            trialSpin(j);
            myFile << i << "\t" << CalculateEnergy() << "\n";
        }
    }
    myFile.close();
}
int main(int argc, const char** argv)
{
    int i = 0;
    double j = 0;
    std::string s = "J";
    while (i < 10)
    {

        char str[10];
        sprintf(str,"%d.txt",i);
        std::string filename = s;
        filename.append(str);
        std::ofstream myFile;

        myFile.open(filename.c_str());

        Heisenberg myHeis = Heisenberg(40,1,j,0.15,4000);
        myHeis.setInitialState();


        for (int i = 0; i < 4000; i ++)
        {
            for (int j = 0; j < 40; j++)
            {
                myHeis.trialSpin(j);
            }
            myFile << i << "\t" << myHeis.CalculateEnergy() << "\n";
```

```
        }
        myFile.close();
        i += 1;
        j += 0.1;
    }
}
```
