Exam Computational Physics
2020

Thorvald M. Ballestad

May 8, 2020

Abstract

Metropolis Monte Carlo simulation of 2D Ising model, using the Mon-Jasnow and extended Mon-Jasnow algorithm. This report is not intended as a scientific report, but as an answer to the exam. The goal of this report is to give a brief overview of how I have solved the problem, and then go into the parts of the solution that I think are the most interesting. Thus, this report consists of a brief documentation of the code and then explanations and thought on the interesting aspects of the solution.

During the exam I spoke to: Knut Andre G. Prestsveen, Niels Henrik Aase, and briefly with Herman Sletmoen.

1 Overview of the code

The simulation is written in Julia and the plots are generated in Python. As the Python code is just simple plotting, I will not give it much focus here. The code is separated into files, the main part being in `utils.jl`. Nothing should be run directly from `utils.jl`, it contains the machinery that can be used from other designated scripts. For each type of "operation" that we want to execute, there should exist a file, that extends `utils.jl`, for example `task1.jl` and `investigate_error.jl`.

Firstly, a *very* brief explanation of the Metropolis method. Given a 2D Ising lattice, select at random one lattice site. For this site, determine the change of energy in the system if one were to flip this spin. If the change in energy is negative, accept the flip. If it is positive, let the probability of flipping it be given by the Boltzmann distribution. A natural definition of a time step is a sweep, defined as N^2 flip attempts, where N is the dimension of the lattice. In the Mon-Jasnow and extended Mon-Jasnow algorithm, one applies the Metropolis method to certain lattice, that is such that there exists another lattice of close resemblance, but for which there exists an interface. It can be shown that the tension τ is a function of the difference in energy of the two systems.

Here is a quick overview of the code. The code is also well documented, so if anything is unclear I urge you to look at `utils.jl`. Then follows a discussion of the parts and solutions that I believe are the most interesting. The spin states are stored in a $N \times N$ array `H`, as $+1$ and -1. The core of the simulation is the flip attempt, represented with the function `step!`. This function selects a random lattice site and makes the flip attempt at that site. When deciding whether to flip the spin or not, the change in energy must be calculated. This is done by `neighbor_interaction_sum`. After the flip attempt, the change in energy, which might be zero, is returned. This energy is returned to `sweep!`, which carries out one sweep. `sweep!` sums up the energy change from all the steps, and finally returns the total change after one

sweep. `sweep!` is called from `simulate!`, the main function in the code. `simulate!` contains the main loop and also necessary setup and data treatment, such as calculating initial energy, allocating memory used inside the loop, and summing up the energy changes in each sweep after the simulation.

There are a number of tricks one might use to improve performance. Some of these come from insight to the problem itself and some are of more programmatic nature. As examples of tricks from insight into the problem are generating a table of possible $e^{\Delta H/T}$ values, that are used in the flip attempt, and how one calculates the energy difference. As a flip will only affect the nearest neighbors (the definition of Ising model), there are only a few possible values for the Boltzmann weights. In fact, as we always accept flips with negative energy change, the only values that needs calculating are $e^{-8/T}$ and $e^{-4/T}$. As exponents are computational heavy, this gives a big performance boost.

As for the energy calculation one has several options. The naive implementation would calculate the energy of the entire spin matrix for each measurement. However, a better approach is to measure only the initial energy, and then cumulatively add the energy change from each flip. In our problem, the energy is not really needed, except for debugging and determining when the system reaches equilibrium. The only quantity of importance, is the difference in energy between the two systems that the Mon-Jasnow algorithm considers – fixes positive and negative sides for original Mon-Jasnow, and torus and Klein flask for the extended algorithm. These energy differences have simple formulas: two time the magnetization of the rightmost column for the original, as stated in [3], and for the extended I found the expression

$$\sum_y S_{1,y}[S_{N_x,y} + S_{N_x,N_y+1-y}], \tag{1}$$

which is stated without proof. As Julia is column oriented in its arrays, this ends up being very efficient to calculate.

Now to the more programmatic aspects of my solution. There are some solutions in particular that I would like to emphasize. Here I will not discuss the performance in detail, for that see 3. Firstly, the code is written in a very module based manner. This makes it easy to change, both cosmetic changes to code and alterations to how one calculates or represents data. This gives a very general code, for which I am proud. When going from the original algorithm to the extended, only the input data must be changed, the code stays the same. This was also very helpful when performing benchmarks and improving performance.

One of the programmatic challenges in this problem is the boundary conditions. When calculating neighbor interaction one has to know what the neighbors are. Most implementations would probably, even though there are other possibilities, store the spin information in a two dimensional array, maintaining position information from the system in the data structure. A first naive approach to the boundary conditions would be to for each flip attempt check if one is at a boundary. Firstly, this could give more complicated code, and also it would have to be rewritten for each type of system. Secondly, it would involve checking several conditions, and if one is at the border, additions. In my solution, inspired by exercise 1 in this curse, the position information of the data structure is disregarded, at least when checking neighbor interaction. For each direction, right, left, up, down, there is a vector that gives the next index. For example, if you are at position (x, y), the next position to the right would be $(ir(x), y)$, where `ir` is the vector that gives the index to the right. Normally, `ir` at index i is simply $i+1$, likewise for down, `id` at index j is $j-1$. However, at the boundaries, the index is such that the boundary condition is met, for example the right index at the rightmost position N_x would be 1. A point of note is that the left and right position is only dependent on x and likewise up and

down is only dependent on y. Thus, the sites can be connected in whatever order one wishes, but only in x and y independently – the data are like two graphs, one for x and one for y. At one point, I had the structure so that it was only one graph, the right position was dependent on both x and y, and would give new values for x and y, allowing arbitrary geometries. This would be necessary for the Klein bottle geometry, for example, where going right may change the position in y. However, since one never has to do any calculations on the Klein system, only the torus, I chose to reduce the generality of my solution in favor of performance. These vectors, describing the structure of the graph, will from here on be referred to as index vectors.

The index vectors are central in my solution to setting up the fixed positive and negative columns of the original Mon-Jasnow algorithm. When selecting a lattice point at which to attempt a flip, I generate a random number between one and the number of lattice points in my system. In the original Mon-Jasnow the site to flip cannot, of course, be one of the fixed columns. In the interest of keeping the code general, my solution is somewhat less intuitive, but still highly efficient. Since we choose a random number between 1 and N^2, the first N^2 numbers must be non-fixed, and the fixed columns must be placed at the end of the array. As Julia is column based, this corresponds to columns after the Nth column. One could of course omit this by generating a number between a given number and some other given number, but in my opinion, that would be less clean code. The solution is to add a positive column at the end, ie. to the right, of the spin array. We do not need a negative column, as one never simulates on the positive/negative system, only the positive/positive. Using the index vectors, this column is placed *both* to the right and to the left of the system. The index right vector at position N_x gives N_{x+1} where the positive column is. The clever part is that the index left vector at position 1 is N_{x+1}! Finally, to make this solution work completely, energy calculation has to be considered. Given a spin configuration array, it is simple to calculate the total energy – simply sum over all neighbor interactions. To maintain general code, this sum would also run over the fixed column on the right (remember, I now talk about the position in actual memory, so the left column does not exist here. But for neighbor interaction in the calculation, the index vectors are used). The right column has no neighbor on the right, it is not a part of the system! Here, the solution is again the index vectors, because the index vector right at position N_{x+1} is N_{x+1}! This simply lead to a constant addition in the energy, which we of course are allowed to do.

Measurements are not trivial for Monte Carlo simulations. There are mainly two aspects of interest, the equilibration time and correlation time. When starting the simulation, it must be run for some period of time before the system has reached equilibrium. Before this point, it makes no sense to take samples for calculating expectation value. Also, after a measurement, one should wait some period of time before making a new measurement, to ensure that the two states are not correlated. One might do alot of work on these aspects. For example, for finding equilibration time, one solution is to simulate two different initial conditions, and see when the energy of the two systems stabilize at the same level. I did this in order to find an approximate value, but another solution, which would require some more computation time, would be to do this every time one ran the system, and use the collected data to infer when to begin measurements. Also, much more work could be done in finding a correlation time. I did this quite naively, simply measuring every three sweeps. [1] had quite a bit of discussion on this point, and concluded that it would often be sufficient to sample every sweep for Metropolis. It is possible that one could get better results if one had put more work into this, but I believe it would take quite alot more time and also would require spending more time understanding the physics fully.

For obtaining error estimates, I used the Bootstrap method, described in [1]. It is not perfect, for example it is heavily dependent on the number of resamples, I do 700, but it does at the

very least give an indication of relative uncertainty between simulations. The method is quite computational demanding, for low values of the lattice size N the Bootstrapping take more time than the simulation!

2 Results

Figure 1 shows τ as a function of T for some values of N, calculated with both the original and extended Mon-Jasnow algorithm, here referred to as the pp and torus systems respectively. Going from $N = 5$ to $N = 30$ it is very obvious that the phase transition is more apparent in both models. From $N = 30$ to $N = 80$ the difference is not as noticeable. All calculations overestimate τ relative to Onsager's solution. The torus system performs much better than the pp system, giving both sharper phase transition and values closer to the exact results.
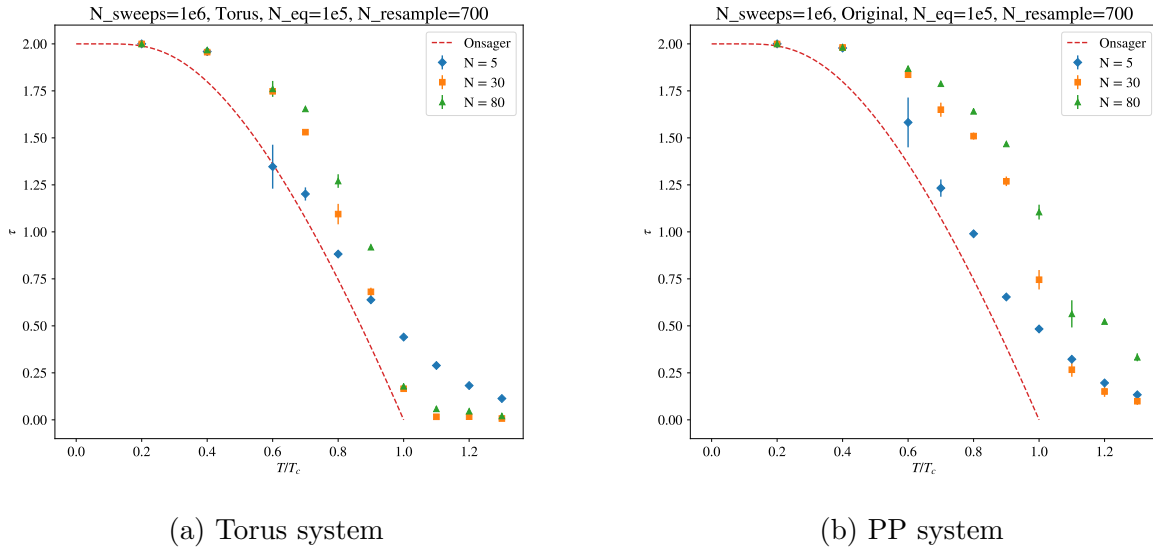

(a) Torus system   (b) PP system

Figure 1: τ as a function of T for various N Also shown is Onsager's exact result (dashed). The results were obtained with one million sweeps, an equilibration time of 1×10^5 and 700 resamples in the bootstrapping.

In figure 2 τ is plotted as a function of N for $T = T_c$. Note the logarithmic scales. From the exercise text we have that near T_c, where t is small,

$$\tau \propto N^{\nu/\mu} \tag{2}$$

so that in the log-log plot of $\tau(N)$ one should expect the trend to be a line of slope

$$\frac{\log \tau(N_2) - \log \tau(N_1)}{\log N_2 - \log N_1} = \frac{\log \tau(N_2)/\tau(N_1)}{\log N_2/N_1} \tag{3}$$

$$= \frac{\log N_2^{\nu/\mu}/N_1^{\nu/\mu}}{\log N_2/N_1} \tag{4}$$

$$= -\mu/\nu = -1. \tag{5}$$

From the figure it appears that the points fall on a line with about negative one slope. Note that for large N the uncertainty is very high. Running longer simulations would likely improve this. Also note that for larger N, the torus system is much more stable than the pp system.
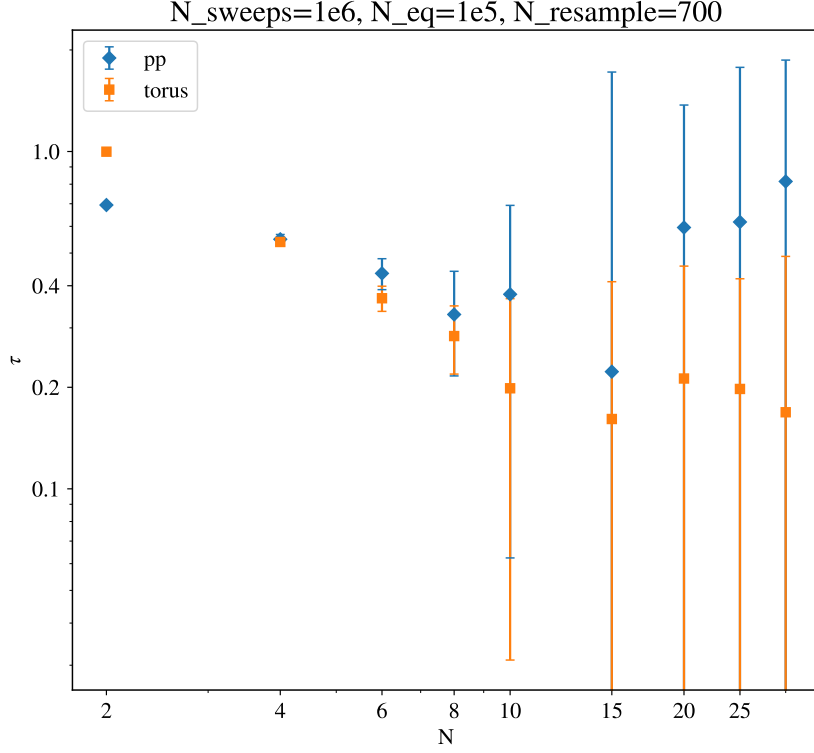
4

Figure 2: τ as a function of N for $T = T_c$. Note the logarithmic scales. The results were obtained with one million sweeps, an equilibration time of 1×10^5 and 700 resamples in the bootstrapping.

τ/t is plotted for both Nt and $1/Nt$ in figure 3. τ_0 is found using linear regression to be 3.06. An improved estimate could probably have been found by simulating for more values. Notice in 3b that the simulated points flatten out before the fitted curve, especially the $t = 0.1$ (arrow down) points. This indicates that one possibly could get a better estimate if one instead of doing linear regression had done it on this curve. The figure suggests that the true value of τ_0 is higher than the one I found here, which is correct as the exam text states that $\tau_0 \approx 3.99$.

(a) τ/t of $1/Nt$ (b) τ/t of Nt
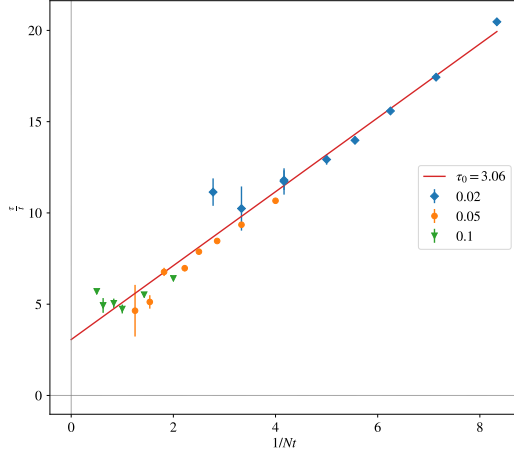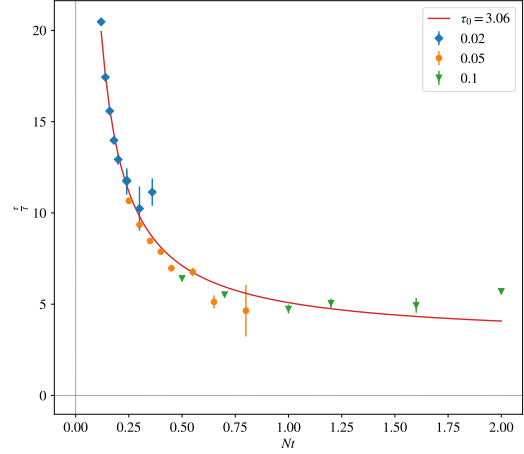
Figure 3: τ/t for various Nt used to estimate τ_0, found with linear regression.

3   Findings related to performance

During the development process, several benchmarks were made, and I discovered some small changes that had huge impact on performance. Especially pieces of code that are involved in the very core of the simulation loop, the flipping, are vital to streamline as they are executed several million, or even billion, times.

The most successful change I made with regards to performance, was also quite subtle. At the core of the simulation, in the flipping attempt, the function `step!` in my code, one has to calculate the interaction between neighbors. In the old version of the neighbor interaction calculation I had the following snippet, generating a vector of neighbor spins.

```
return [H[(y, x)] for (y,x) in [
    (y, ir[x]),   # Right
    (y, il[x]),   # Left
    (iu[y], x),   # Up
    (id[y], x),   # Down
]]
```

Here, `H` is the spin array and `ir, il, iu, id` are the index vectors. I changed this snippet to

```
return H(y, ir[x])+   # Right
       H(y, il[x])+   # Left
       H(iu[y], x)+   # Up
       H(id[y], x)    # Down
```

The difference between the two techniques is that the former makes a new a list, to which is applied the built in sum function(at another place in the code) while in the latter the sum is carried out directly, not having to allocate a list. The change in performance was huge – going from 34 seconds to 3.5 seconds in a benchmark with $N = 30$ and hundred thousand sweeps. That corresponds to cutting the computation time from one hour to six minutes! It is worth mentioning that the former solution is not only unnecessarily creating a list, the list is created through a loop! However, even a list written out, not created using a list comprehension, is slower than the direct sum, with a factor of about 2.

6

At one point, I changed the type of the index vectors from Int64 to UInt8, under the assumption that one would not simulate systems of $N = 255$. This would obviously save memory, but strangely it had a negative impact on speed, with a factor of about 3. I did not pursue this in depth, so I do not now the reason for this. My theory is that either there is something inherent with Julia or the computer that makes 64 bits faster, or that the compiler performs some hidden conversion from UInt8 to Int64 somewhere in the loop. The latter possibility could have been found by writing the code more strict in type, but I did not prioritize this.

One last point on performance is parallelization. The simulation is very CPU-demanding, but not very memory expensive, and should be fit for running on multiple threads. One simulation can not be multi threaded, as it has to be sequential, but running multiple simulations at once would be very beneficial in leveraging the full potential of the CPU. I did run my simulations in parallel without any issue, with as many as four simulations at a time! The only critical point then is to make sure that no part of the execution, for example the bootstrapping, requires too much memory and that the threads do not overwrite each others results in file. In fact, my first implementation of bootstrap was vectorized, using alot of memory for long simulations, and I had to rewrite it into using less memory in order to do this.

As a final note on parallelization, I will mention that I did very briefly attempt further optimization of the neighbor interaction calculation. The neighbor interaction consists of accessing four spin values and summing them. As mentioned, speeding up this operation is very impactful. Utilizing the SIMD instructions for multiplying these four values would probably give a considerable speedup. I did try this, but was not successful.

References

[1] M. E. J. Newman and G. T. Barkema, *Monte Carlo Methods in Statistical Physics* chap. 3, Oxford University Press, Oxford, (1999).

[2] L. Onsager, *Crystal statistics. I. A two-dimensional model with an order-disorder transition*, Phys. Rev. 65, 117 (1944).

[3] K.K. Mon and D. Jasnow, *Direct calculation of interfacial tension for lattice models by the Monte Carlo method*, Phys. Rev. A 30, 670 (1984).