University *of Ljubljana*
Faculty *of Mathematics and Physics*

Department of Physics

Metropolis-Hastings Algorithm

8. Task for Model Analysis I, 2023/24

Author: Marko Urbanč
Professor: Prof. Dr. Simon Širca
Advisor: doc. dr. Miha Mihovilovič

Ljubljana, July 2024

Contents

1 Introduction

We're continuing our exploration into random numbers and their applications. Previously we had a look at Monte Carlo sampling. Today we're going to delve into the Metropolis-Hastings algorithm, which can be thought of as Monte Carlo sampling with a few extra steps. Since our end goal is to simulate the relaxation of a lattice of spins in a magnetic field, we'll take the Ising model as our physical context.

We know from statistical physics that the 2D Ising model relaxes to a state of minimum energy. We can simulate this relaxation by flipping spins at random and accepting or rejecting the new state based on the change in energy. We can have a negative change of energy which we can call a *good move* or a positive change of energy which we can call a *bad move*. The added twist with this algorithm is that while we always accept the new state if we make a good move, we also sometimes accept a new state after a bad move. This is the key to the Metropolis-Hastings algorithm. Given our system the probability of accepting a bad move is given by the Boltzmann factor and the temperature of the system:

$$P_{\text{bad accept}} = \exp\left(-\frac{\Delta E}{kT}\right) , \tag{1}$$

where ΔE is the change in energy, k is the Boltzmann constant and T is the temperature. Why exactly this works is a bit more involved and probably out of the scope of this report however a dedicated reader can find more information in this well written blog post by Gregory Gundersen of Princeton University [1]. We can define multiple different exit conditions for the algorithm, such as a fixed number of iterations, a fixed number of accepted moves or an ε tolerance for ΔE. Really that is all there is to this method from a theory standpoint.

2 Task at Hand

2.1 Molecular Chain

The instructions given demand that we first explore the Metropolis-Hastings algorithm on a simple molecular chain. The molecular chain is made up of 17 molecules that can go from a state of 0 to a depth of -18. The deeper a molecule is the lower its potential energy. But we also have a positive energy contribution if neighboring bonds are very stretched. The final result is determining the equilibrium energy as a function of the temperature.

2.2 Ising Model

As mentioned in the intro our main goal is to simulate the relaxation of a lattice of spins in a magnetic field. We can describe a ferromagnetic material with the following Hamiltonian:

$$\mathcal{H} = -J \sum_{\langle i,\, j \rangle} s_i s_j - H \sum_i s_i \,, \tag{2}$$

where s_i is the spin of the i-th site, H is the external magnetic field and J is the coupling constant, which is positive for ferromagnetic materials and negative for antiferromagnetic materials. In the absence of a magnetic field the critical temperature for the transition from a paramagnetic to a ferromagnetic state is given by the Onsager solution approximately as:

$$T_c \approx 2.269185\, \frac{J}{k_B} \,. \tag{3}$$

The instructions demand that we determine the average energy $\langle E \rangle$ and eigen-magnetization $\langle S \rangle$ as functions of temperature. The magnetization of the system is defined as:

$$S = \frac{1}{N} \sum_i s_i \,, \tag{4}$$

where N is the number of spins. We can also have a look at spin susceptibility χ and heat capacity c at different external magnetic field strengths. Spin susceptibility is defined as:

$$\chi = \frac{\langle S^2 \rangle - \langle S \rangle^2}{N k_B T} \,, \tag{5}$$

and heat capacity is defined as:

$$c = \frac{\langle E^2 \rangle - \langle E \rangle^2}{N k_B T^2} \,. \tag{6}$$

3 Solution Overview

This report marks a change in the way I approach these tasks for Model Analysis I. Up until now I've given heavy focus on the quality, optimization and performance of my code and have neglected the actual purpose of the subject, which is the analysis of physical models. The idea was that more advanced code or methods would allow me to do better analysis. However, the overhead of time needed to implement these methods was not worth the benefit. I don't think I've ever managed to show the results of my code since I've always been too focused on the code itself. It's time to change that. As much as it pains me I will try to employ a *good-enough* approach (aka. the *KISS* principle) and focus on the analysis of the models. I will still try to write clean and efficient code but I will not spend too much time on it, especially on paralelization and optimization. From a utilitarian perspective it's better to have a working solution that is not optimal than to have an optimal solution that took to long to implement.

3.1 Molecular Chain

I stared writing the code for the molecular chain quite a long time ago before I burnt out. I threw all that old code away and started anew. As has become a bit of a habit I created a 1D Metropolis-Hastings solver class `Metropolis1` which I can then also reuse for the Ising model with minimal adaptations. The advantage of a class-based approach is that I can more easily do parameter sweeps as I can just create a new instance of the class with different parameters. Since I wanted to get decent statistics for the molecular chain I had quite strict exit conditions and also many repeat runs for each temperature. This produced quite a massive amount of data which I conveniently stored in a `HDF5` file. I ♡ `HDF5`.

I started of by just running the code and plotting a few results to see if everything is working as expected. A nice result is displayed in Figure (1) where the average of 100 runs is shown for the final state of the molecular chain as well as the scatter of the final temperature and energy across runs where the initial state is fixed. The aim of this plot is to display the working of the Metropolis-Hastings algorithm and its convergence to a final state.

It was here that I also experimented slightly with the parameters of the chain, such as length or allowed molecule depth. For example I managed to find a state that optimized almost to a chain link but since one of the molecules was on the top level it tried to make two chain links, as displayed in Figure (2).

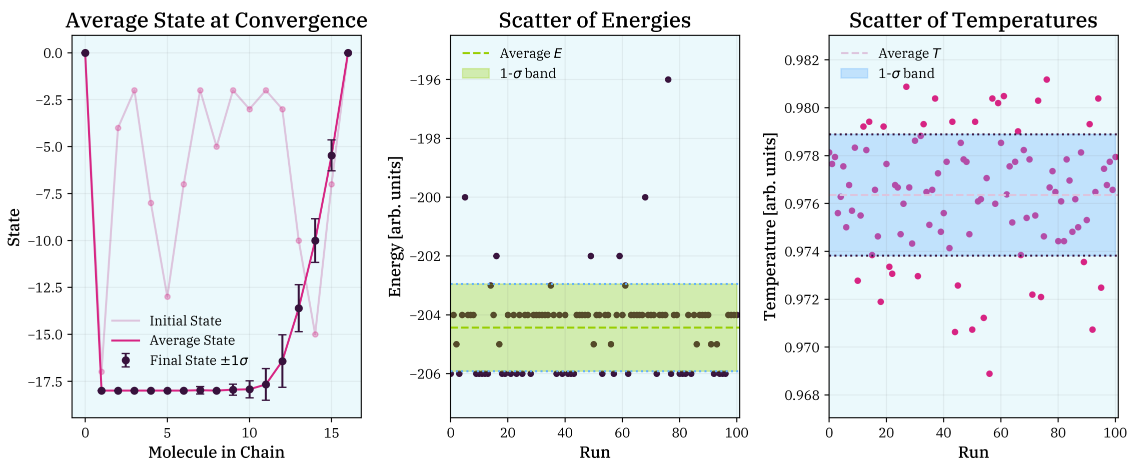

Figure 1: Final state, Temperature, Energy scatter over runs for the molecular chain at a fixed initial state.

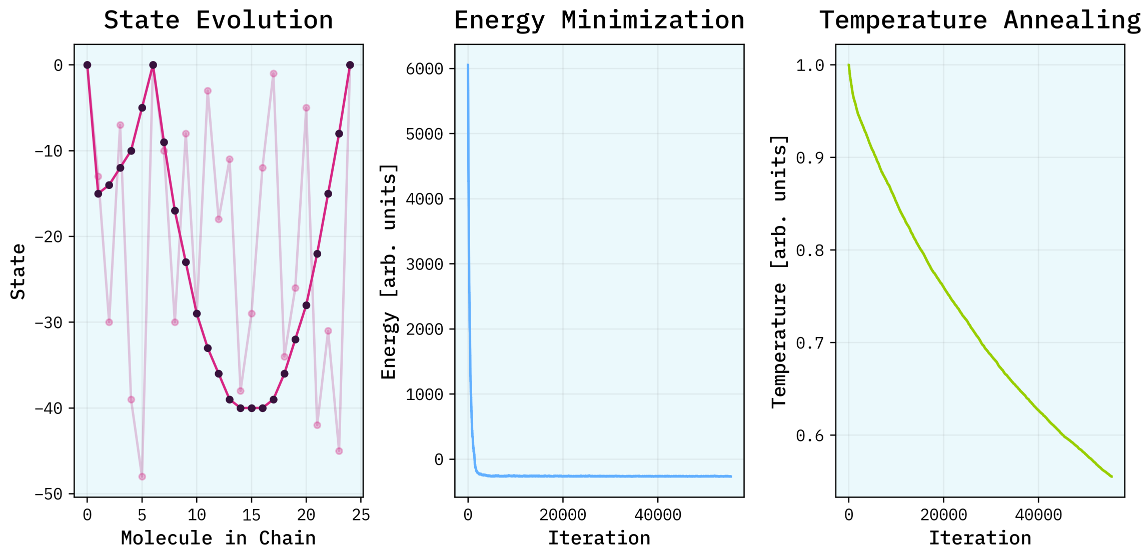

Figure 2: Optimized state of the molecular chain that almost forms a double chain link.

3.2 Ising Model

After (re)writing the code for the molecular chain I adapted it for the Ising model, creating a solver class `Metropolis2`. I had a few problems with the scale of energy and temperature but I think I managed to solve those in the end and the results seem to be reasonable. Data was also stored in a `HDF5` file, due to the sheer volume as I wanted a decent sample size for my statistics. The main challenge was the code being relatively slow per lattice and I wanted to simulate long times. As much as I promised not to focus on optimization I did to single-computer parallelization by utilizing the `ProcessPoolExecutor` from the `concurrent.futures` module of Python. As stated above, a class-based approach allowed me to easily spawn multiple instances of the solver class with different parameters on different processes which greatly sped up the gathering of data. I think this style of parallelization is reasonable as it takes very little time to implement and yields significant speedups. Here is a brief overview of how `ProcessPoolExecutor` is used:

```python
import ...

class Worker:
    def __init__(self,
                 temperature: float,
                 run: int,
                 ):
        self.T = temperature
        self.run = run

    def process(self):
        dim = (50, 50)
        m = Metropolis2(dim, self.T)
        # Set parameters for the simulation
        s_init, s_final, en = m.run()

        return self.run, s_init, s_final, en, m.temperatures

# Create function that'll get the workders to work
def worker_task(params):
    worker = Worker(*params)
    return worker.process()

# Run process pool and save on thread execution
with concurrent.futures.ProcessPoolExecutor(max_workers=16) as executor:
    futures = [executor.submit(worker_task, params) for params in param_list]

    # Step 4: Collect the results
    for future in concurrent.futures.as_completed(futures):
        run, s_init, s_final, en, temps = future.result()
        # Save data etc.
```

4 Results

4.1 Molecular Chain

5 Conclusion and Comments

References

[1] Gregory Gundersen. Why Metropolis-Hastings Works. https://gregorygundersen.com/blog/2019/11/02/metropolis-hastings/#bishop2006pattern, Nov 2019. [Accessed 24-07-2024].