Simulation Methods in Physics 1 - Report

Worksheet 5

Jens Jakschik, Fabio Oelschläger

February 1, 2021

Source code via Github
`https://github.com/bliepp/Simulation-Methods-in-Physics-Exercises`

Exercise 2: Simple sampling - Integration

The first exercise was to implement exact integration as well as Monte-Carlo integration in Python. The following function was to be integrated:

$$f(x) = (-2x^2 \sin(x)\cos(x) - 2x\sin^2(x)) \cdot \exp(-x^2\sin^2(x)). \tag{1}$$

First, the function was to be integrated exactly using sympy, which was done with the following code:

```python
from sympy import *
from sympy.utilities.lambdify import lambdify

x = Symbol('x')
expression = (-2*x*x * sin(x) * cos(x) - 2*x* sin(x)*sin(x))
            * exp(-x*x * sin(x) * sin(x))
f = lambdify(x, expression) # to use like a normal python function
result = N(integrate(expression, (x, 0.1, 50))) # calculated by sympy
```

Without calculating the result of the integration in the given boundaries, the following function was obtained:

$$F(x) = \exp(-x^2\sin^2(x)) \tag{2}$$

Then, afterwards, the same integration was to repeated using Monte-Carlo integration. Through the Monte-Carlo integration an approximation of the integration can be obtained by summing over random states in a defined range of the system, which is the so-called simple sampling method. In total, N random states of the system are generated and evaluated. To improve the accuracy of the Monte-Carlo integration, this is done for a total of M times (in our case M was chosen as 100) and then averaged over all the obtained results. This helps in minimizing random errors, as the mean will always be closer to the real result. In addition to this, the accuracy of the Monte-Carlo integration was tested for different sampling sizes N. Also, the error of the Monte-Carlo integration was determined by taking the variance of the results. This was done with the following code:

```python
def simple_sampling(func, a, b, N, M=100):
    results = np.empty((M), np.float32)
    for i in range(M):
        results[i] = np.sum(func(np.random.uniform(a, b, N)))*(b-a)/N
    mean_ = np.mean(results)
    variance = np.var(results)
    error = np.sqrt(variance)

    return np.array([mean_, error])
```

The results of these different steps can be seen below.

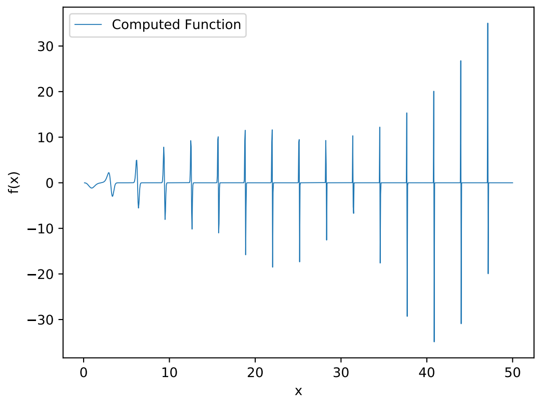

Figure 1: Results of the integration of the given function

Above, the result of the exact integration of the given function can be seen. There, it can be seen, that it is a function that is actually fairly difficult to determine using Monte-Carlo integration, as there are very sharp peaks. These peaks, given a low enough number of samples in the Monte-Carlo integration, can result in strong variance in the obtained results. This is confirmed by difference between the Monte-Carlo approximation and the actual results, which can be seen below. There, it can also be seen, that the Monte-Carlo integration exhibits a very strong variance for low sample sizes, while the variance becomes increasingly small for higher sample sizes.

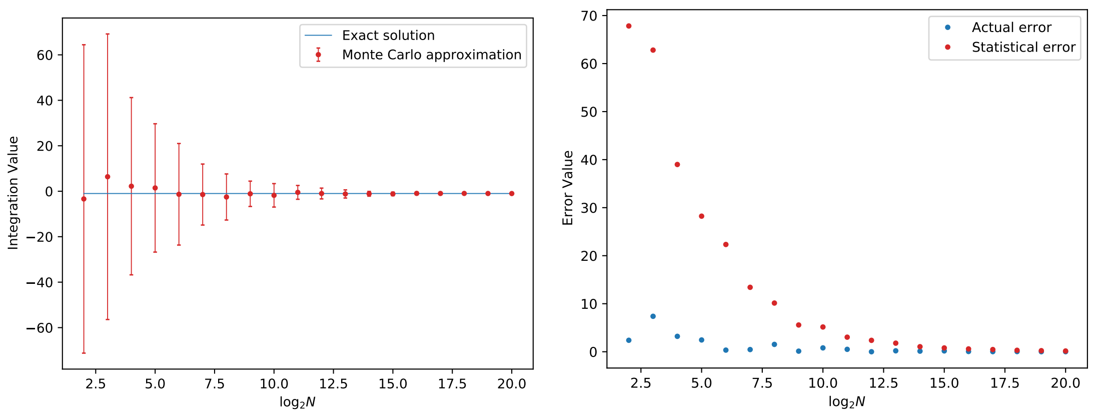

Figure 2: In the first picture, the results of the Monta-Carlo integration and their variance compared to the results of the actual integration can be seen. In the second picture, the actual error of the is compared to the statistical error of the Monte-Carlo integration.

In the second figure above, the actual error is compared to the statistical error of the Monte-Carlo integration. There it can be seen, that both errors approach zero for higher sampling sizes, which is to expected. Also, as expected, the statistical error is much higher than the actual error for low sample sizes. Normally, the actual error could also be much higher, but, as we averaged about 100 times, the actual error is reduced.

Exercise 3: Importance Sampling – Metropolis-Hastings Algorithm

For the next exercise, an often more applicable version than simple sampling was to be implemented. Here, the states of the system are not generated totally random like in simple sampling, but instead in accordance with a probability distribution. Goal of this exercise was to implement the Metropolis-Hastings-algorithm. In this case, a trial move was used which simply adds a uniformly distributed random number to the initial state. For this, different move ranges were implemented. To account for the influence of the different move ranges, each step only has a chance to be actually taken and the larger the step, the smaller the chance. All this was done with the following code:

```python
def metropolis(N, P, trial_move, phi0):
    #return "Superman"
    phi = phi0
    out, acceptance = list(), 0
    for i in range(N):
        phi_new = trial_move(phi)
        r = np.random.rand()

        # branchless if condition because we can
        condition = r < min(1, P(phi_new)/P(phi))
        phi = (condition)*phi_new + (not condition)*phi
        acceptance += condition

        out.append(phi)
    return np.array(out), acceptance/N
```

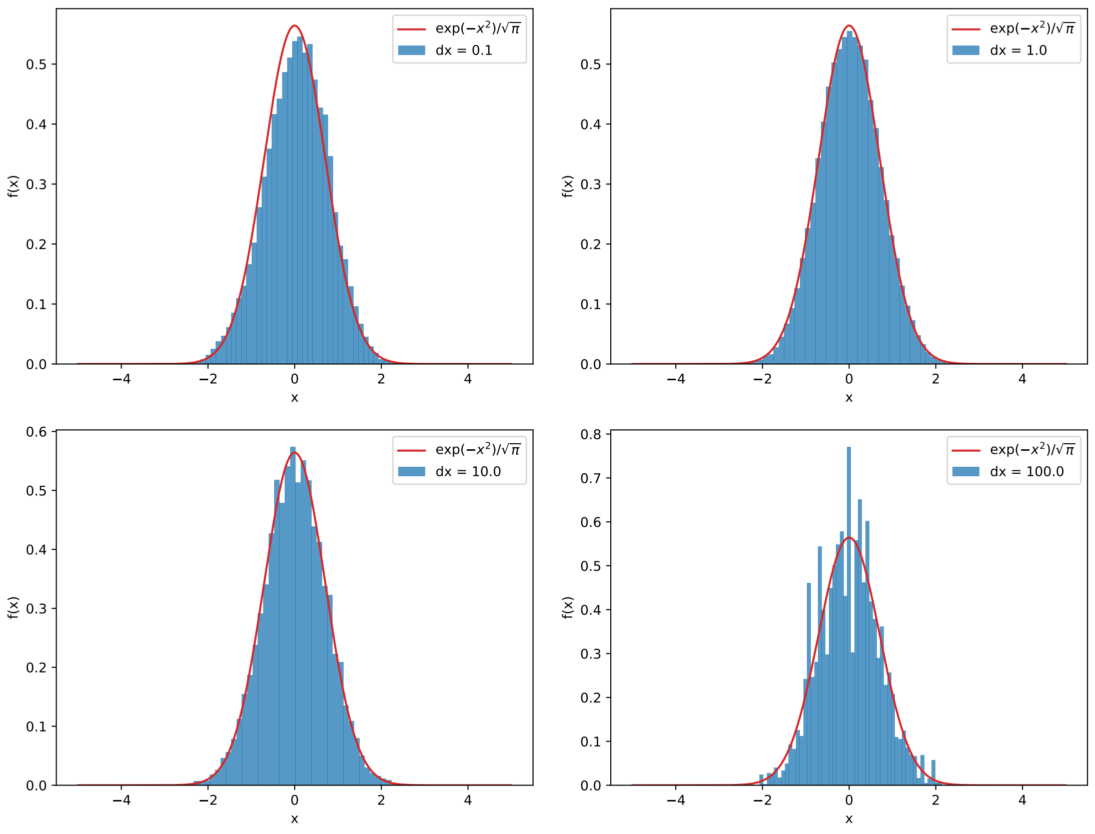

Figure 3: Results of the Metropolis-Hastings-algorithm with different step sizes applied to more or less create a random walk.

In the above pictures it can be seen, that the best fit to the theoretical distribution is obtained using a step size of 1 with an acceptance rate of 0.97 and all step sizes above result in progressively worse distributions. However, it can also be seen that the smallest step size of 0.1 also exhibited a poorer fit to the theoretical curve when compared to the step size of 1.0. This was not within our expectations, as we expected the smallest step size to result in the most accurate distribution, as the smallest step size also has the highest acceptance rate. The obtained acceptance rates are listed below.

Step Size dx	Acceptance Rate
0.1	0.97
1.0	0.73
10.0	0.11
100.0	0.01

Table 1: Acceptance rate in the Metropolis-Hastings-algorithm for different step sizes.

Exercise 4: Simulating the Ising model

The next exercise was to create a simulation for the Ising model. Here, a system with periodic boundary conditions on a (L x L) square lattice was used to calculate the total energy, the mean energy per site and the mean magnetization per site of this system. As the task was to do this for a 4 by 4 grid, 'some' C++ code was used to speed up the calculations (basically everything was written in C++). First, the Ising model was implemented in Python without implementing Monte-Carlo methods to determine an exact solution.

```python
def ising_exact_oop():
    TMP, ENG, MAG = list(), list(), list()

    from ising.cpp import Ising
    modell = Ising(args.L)

    for T in range(10, 51):
        kB, T = 1, T/10
        beta = 1 / (kB*T)

        energy_per_side, mag_per_side, partition_function = 0.0, 0.0, 0.0
        lattices = itertools.product(*[(-1,1)]*modell.L2)
        length = 1 << (args.L*args.L) # binary representation of lattice -> 2^(L*L) poss

        for lattice in tqdm.tqdm(lattices, total=length, desc=f"T = {T}"):
            modell.lattice = lattice
            energy = modell.energy
            magnetization = modell.magnetization

            # <A> = SUM A*exp(-beta H) / Z
            energy_per_side += energy * np.exp(-beta * energy)
            mag_per_side += abs(magnetization) * np.exp(-beta * energy)
            partition_function += np.exp(-beta * energy)

        TMP.append(T)
        ENG.append(energy_per_side / partition_function / TOTAL)
        MAG.append(mag_per_side / partition_function)

    return TMP, ENG, MAG
```

The results of this exact represantation of the Ising model are shown below.

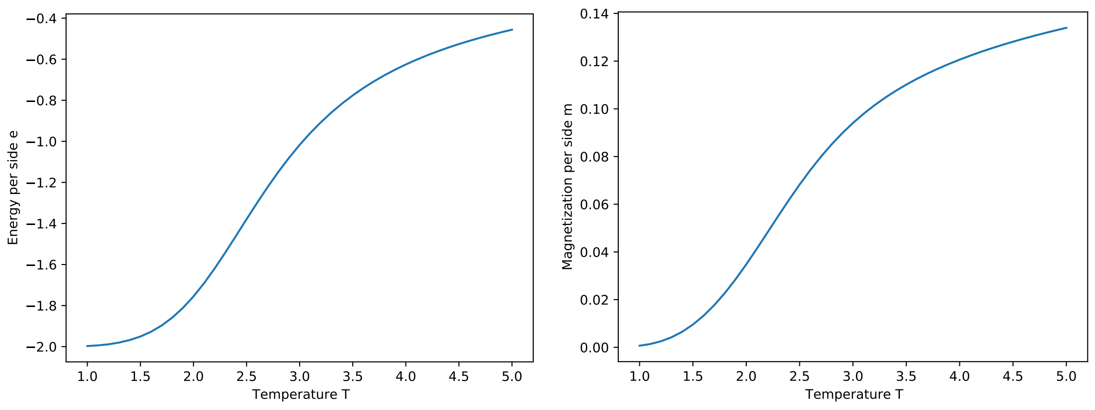

Figure 4: Local Energy and Magnetization in the Ising model in dependence of the Temperature

Here, it can be seen that both the Magnetization and the Energy show strong changes with rising temperature. The trend of these curves was to be expected, as an increased temperature results in a disturbed system where the spins have a higher chance to flip. The lower the temperature, the higher the ordering of the spins, meaning higher energy and higher magnetization.

To implement Monte-Carlo methods into the Ising model, the Python code from the previous exercise was implemented in class-based C++ and extended. This was done as we found the calculation times to scale extremely strongly with the lattice size. Especially in the initial version of our code many unnecessary calculations were done, resulting in a 4x4 grid which couldn't be simulated. By getting rid of unnecessary calculation and implementing it in C++, the calculation times were reduced by more than 65 times. To determine the probability of a spin flip, the Maxwell-Boltzmann statistic was used, which resulted in the following probability (with σ as the spin state of the Ising lattice):

$$P(\sigma) = \exp(-\beta\mathcal{H}(\sigma))/Z_\beta \tag{3}$$

$$P(\sigma_1)/P(\sigma_0) = \exp(-\beta(\mathcal{H}(\sigma_1) - \mathcal{H}(\sigma_0))) = \exp(-\beta\Delta E) \tag{4}$$

with $\mathcal{H}(\sigma_1) = E_1$ being the energy of the system after the spin flip and $\mathcal{H}(\sigma_0) = E_0$ the energy of the system before the spin flip.

```cpp
std::vector<double> Ising::metropolis(unsigned int steps, double beta){
    //std = sexually transmitted desease
    std::uniform_real_distribution<float> r_dist(0.0, 1.0);
    std::uniform_int_distribution<int> choose_element(0, this->L2);

    double accepted = 0.0, e = 0.0, m = 0.0, E = this->energy();
    for (unsigned int step = 0; step < steps; step++){
        float r = r_dist(*this->generator);
        int index = choose_element(*this->generator);
        int i = this->get_i(index), j = this->get_j(index);

        double dE = this->flip_spin(i, j);
        bool condition = r < std::min(1.0, std::exp(-beta*dE));

        accepted += condition;
        E += dE*condition;

        if (!condition){
            this->flip_spin(i, j); // flip back to previous state
        }

        // no exp(-beta * E), because p = exp(-beta*E)/Z is chosen
        e += E;
        m += std::abs(this->magnetization());
    }

    return std::vector<double>({accepted/steps,
                                e/steps/this->L2, m/steps});
}
```

The results of this algorithm were then input into a python script to be evaluated further. The code is fairly similar to the one used for the exact evaluation of the Ising model, which is why we chose not to include it again here. The mean energy and mean magnetization per side was again simulated in a temperature range from 1.0 to 5.0 and plot against the temperature.

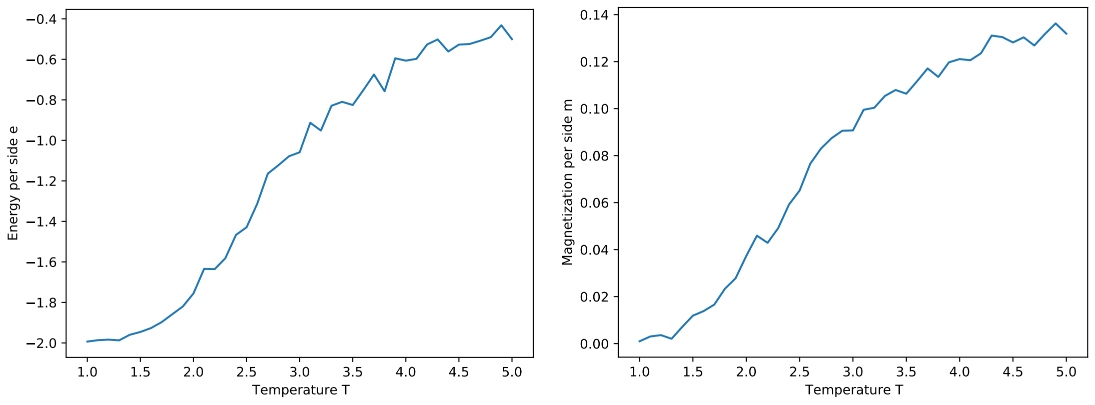

Figure 5: Local Energy and Magnetization in the Ising model in dependence of the Temperature created using Monte-Carlo Simulations

Here it can be seen that the trend of these graphs is extremely similar to those determined through exact summation. There is one major difference, which is the scale of the graphs. So far, we haven't found the cause for this difference between the graphs. But, as everything else is similar, it shows that the Monte-Carlo simulation has worked.