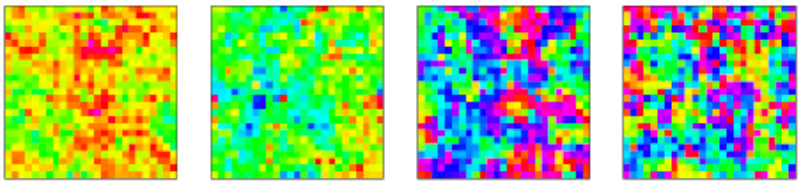
Monte Carlo Techniques

Timothy Budd

Oct 12, 2022

Contents



About this book

This interactive book collects the lecture notes for the course **NWI-NM042B Monte Carlo Techniques** taught in the Physics & Astronomy master program at Radboud University, Nijmegen, The Netherlands.

Please go ahead and browse the different chapters:

A **PDF version** of this book is also available: Book in PDF.

Instructions

Each chapter contains a variety of Python code snippets illustrating the material. In the preparation of this book the code has been executed and the output included statically in the HTML pages and the PDF version. But this does not mean that you cannot interact with the code. For each chapter there is a corresponding Jupyter notebook (in .ipynb

format) that can be downloaded with the links at the top of the page. Note that not all content will render identically in the notebook, but the code blocks will be identical.

If you have access to the JupyterHub server of the Science faculty at Radboud University, then you may use the following NBGitPuller link to synchronize the whole book including Jupyter notebooks to your science account:

https://jupyterhub.science.ru.nl/hub/user-redirect/git-pull?repo=https%3A%2F%2Fgitlab.science.ru.nl%2Ftbudd%2Fmonte-carlo-techniques&urlpath=tree%2Fmonte-carlo-techniques%2F_sources%2F&branch=main

Otherwise you can try launching the book on MyBinder:

https://mybinder.org/v2/git/https%3A%2F%2Fgitlab.science.ru.nl%2Ftbudd%2Fmonte-carlo-techniques.git/main?filepath=_sources

About the author

Timothy Budd is an assistant professor at High Energy Physics (HEP) department of the Institute for Mathematics, Astrophysics and Particle Physics (IMAPP) at Radboud University, Nijmegen.

Fig. 1: Timothy Budd

Part I

Lectures

Chapter 1

Introduction to the Monte Carlo method

Monte Carlo methods are those numerical approaches to any problem in which **random numbers** feature one way or another. Although the general idea is far from new, it came to fruition with the advent of computers in the 1940's and became increasingly capable with our ever-increasing computing power. Nowadays, it is employed all across the natural sciences, computing, mathematics, engineering, finance, computer graphics, artificial intelligence, social sciences.

Fig. 1.1: The Monte Carlo Casino in Monaco.

A little history

The Monte Carlo method in its modern form is often attributed to the work of the mathematician Stanislaw Ulam in the late forties at Los Alamos as part of the H-bomb development team. Calculations of the nuclear chain reactions were posing significant mathematical challenges and Ulam realized that the novel computer they had at the time could be employed more effectively by employing random numbers. Actually his idea was sparked not by his research but by his desire to figure out the chances of winning the card game of solitaire/patience: he had difficulty figuring out the math and philosophized that simulating the game many times on the (multi-million dollar) computer would give an accurate estimate. Because the project was classified at the time, Ulam together with his colleagues von Neumann and Metropolis had to think of a code name for the method, which they based on the famous Monte Carlo casino in Monaco.

Instead of jumping into the mathematics and probability theory underlying the technique or introducing the breadth of available Monte Carlo methods, we will start by looking at several very simple problems to get a feeling for what is ahead of us. This material is largely adapted from the introduction of [Kra06].

1.1 Direct sampling: pebble game 1.0

The general problem of **sampling**, i.e. obtaining random samples from a desired distribution, is in general very difficult. In fact, a significant part of this course is geared towards developing increasingly powerful methods to attack such problems. To grasp the idea we start by looking at a very simple game, which we can imagine children playing in school yard. They draw an exact square and a circle inscribed it and from a distance they start throwing pebbles at it.

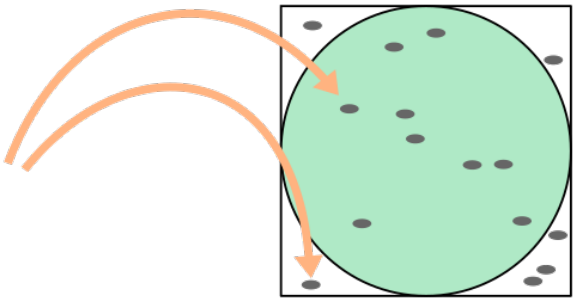


Fig. 1.2: Pebble game: estimating $\pi/4$ via direct sampling.

Each pebble landing in the square constitutes a *trial* and each pebble in the circle a *hit*. We assume that the children are not too apt at throwing, meaning that most pebbles will actually miss the square (which are collected and thrown again), but when they do land in the square the resulting position is completely random and uniformly so.

By keeping track of the number of trials and hits, the children perform a **direct-sampling** Monte Carlo computation of $\pi/4$, because

$$\frac{\text{hits}}{\text{trials}} \approx \frac{\text{area of circle}}{\text{area of square}} = \frac{\pi}{4}. \tag{1.1}$$

The reason this is called direct sampling is because the children have a purely operational way of producing trials independently with the desired distribution (the uniform distribution in the square). Let us write a python program to simulate the outcome of this game with 4000 trials.

```
import numpy as np
# we make a habit of using NumPy's pseudorandom number generator
rng = np.random.default_rng()

def random_in_square():
    """Returns a random position in the square [-1,1)x[-1,1)."""
    return rng.uniform(-1,1,2)

def is_in_circle(x):
    return np.dot(x,x) < 1

def simulate_number_of_hits(N):
    """Simulates number of hits in case of N trials in the pebble game."""
    number_hits = 0
    for i in range(N):
        position = random_in_square()
        if is_in_circle(position):
            number_hits += 1
    return number_hits

trials = 4000
hits = simulate_number_of_hits(trials)
print(hits , "hits, estimate of pi =", 4 * hits / trials )
```

```
3177 hits, estimate of pi = 3.177
```

This estimate is not too far off, but it is clear that the outcome still has some randomness to it. This is apparent when we look at the result of repeated experiment:

```
# build an array with the outcome of 10 repetitions
[simulate_number_of_hits(trials) for i in range(10)]
```

```
[3135, 3136, 3108, 3119, 3138, 3151, 3188, 3134, 3143, 3126]
```

Each of these outcomes gives an approximation of π and we will see shortly how to compute the statistical errors expected from this Monte Carlo simulation. For now we interpret the outcomes in terms of the main principle of the direct-sampling Monte Carlo method and many of its variants. Let $p(\mathbf{x})\mathrm{d}\mathbf{x}$ be a **probability distribution** on a continuous sample space Ω parametrized by $\mathbf{x} \in \Omega$, meaning that $p(\mathbf{x})$ is non-negative and $\int_\Omega p(\mathbf{x})\mathrm{d}\mathbf{x} = 1$. Furthermore we let $\mathcal{O}(\mathbf{x})$ be an **observable** or **random variable**, i.e. a real or complex function on Ω. Then the **expectation value** of $\mathcal{O}$ is given by

$$\mathbb{E}[\mathcal{O}] = \int_\Omega \mathcal{O}(\mathbf{x})\, p(\mathbf{x})\mathrm{d}\mathbf{x}.$$

Instead, we can look at a sampling $\mathbf{x}_1, \mathbf{x}_2, \ldots \in \Omega$ of points that are each distributed according to p and independent of each other. Central to Monte Carlo simulations is that we can approximate $\mathbb{E}[\mathcal{O}]$ via

$$\mathbb{E}[\mathcal{O}] \approx \frac{1}{N}\sum_{i=1}^{N} \mathcal{O}(\mathbf{x}_i).$$

Of course the approximation gets better with increasing number N of samples, how fast we will see in next week's lecture (but you will already perform an empirical study in the *Exercise sheet*). Note that the probability distribution p does not occur on the right-hand side anymore, so we have transfered the problem of integrating $\mathcal{O}$ against the probability distribution $p(\mathbf{x})\mathrm{d}\mathbf{x}$ to the problem of generating samples $\mathbf{x}_i$ and evaluating the observable on the samples.

How does our pebble game fit into this mathematical language? Here the sample space Ω is the square $\Omega = (-1, 1)^2$ and the probability distribution is the uniform distribution $p(\mathbf{x}) = 1/4$, which indeed satisfies $\int_\Omega p(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{-1}^{1}\mathrm{d}x \int_{-1}^{1}\mathrm{d}y\, \frac{1}{4} = 1$. What is the observable $\mathcal{O}$? We are interested in the ratio of hits to trials. If we denote the number of trials by N then we can express this as

$$\frac{\text{hits}}{\text{trials}} = \frac{1}{N}\sum_{\mathbf{x}_i} \mathcal{O}(\mathbf{x}_i),$$

where $\mathcal{O}(\mathbf{x}_i) = 1$ if $\mathbf{x}_i$ is inside the circle and it is equal to 0 otherwise. In other words $\mathcal{O}$ is the **indicator function** $\mathbf{1}_{\{x^2+y^2<1\}}$ on the set $\{x^2 + y^2 < 1\} \subset \Omega$. The expectation value of this observable is

$$\int_\Omega \mathcal{O}(\mathbf{x})p(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{-1}^{1}\mathrm{d}x \int_{-1}^{1}\mathrm{d}y\, \tfrac{1}{4}\, \mathbf{1}_{\{x^2+y^2<1\}} = \frac{\pi}{4},$$

in agreement with the calculation (1.1) above.

1.2 Markov-chain sampling: pebble game 2.0

The sampling problem in the pebble game is simple enough that we did not have to think twice how to design a direct sampling algorithm. For problems we will encounter later on this will be much more difficult and in many cases finding an efficient such algorithm is unfeasible. A more versatile technique is Markov chain sampling in which samples are not generated independently, as in the direct-sampling method, but via a **Markov chain**: samples are generated sequentially in a way that only depends on the previous sample (and not the full history of samples).

To introduce the principle we examine a variant of the pebble game to estimate the value of π. The goal is the same: the children wish to sample random positions in the square by throwing pebbles. This time however the square with

inscribed circle is much larger (let's say the children moved to the local park), and extends beyond the range of a child's throw. To simplify matters we assume a single child is playing the game. The procedure of the game is altered as follows. The child starts in some predetermined corner of the square with a bag full of pebbles. With closed eyes she throws the first pebble in a random direction and then she walks to where it has landed. From that position a new pebble is thrown and the procedure repeats. The goal is still to evenly sample from the square in order to estimate π. Now a problem arises when a pebble lands outside of the square. In the direct-sampling version, such an event was simply ignored and the pebble was just thrown again. In the new version of the game, however, a choice has to be made what to do with the out-of-range pebble and how to continue in such an event. Let us examine the consequences of different choices numerically.

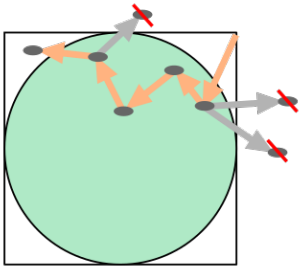
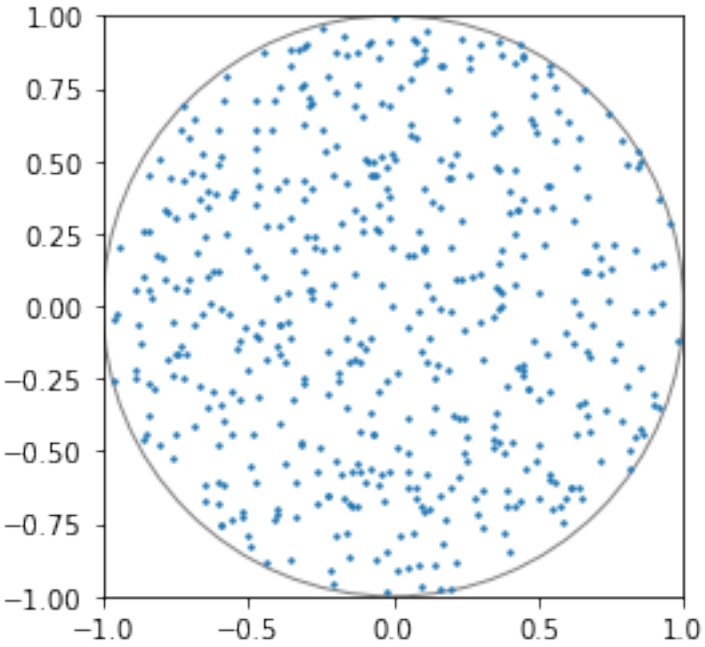


Fig. 1.3: Pebble game: estimating $\pi/4$ via a Markov chain.

We model the throw of a pebble by a random displacement within a disk of radius δ. Based on the previous game we already know a way of sampling a uniform random point in a disk, using direct sampling and rejection:

```
def random_in_disk():
    """Returns a uniform point in the unit disk via rejection."""
    position = random_in_square()
    while not is_in_circle(position):
        position = random_in_square()
    return position

# let us test this visually
import matplotlib.pylab as plt
# tell jupyter to display graphics inline
%matplotlib inline

# sample 500 points in NumPy array
testpoints = np.array([random_in_disk() for i in range(500)])
# make a plot
fig, ax = plt.subplots()
ax.set_xlim(-1,1)    # set axis limits
ax.set_ylim(-1,1)
ax.set_aspect('equal')    # preserve aspect ratio of the circle
ax.add_patch(plt.Circle((0,0),1.0,edgecolor='gray',facecolor='none'))
plt.scatter(testpoints[:,0],testpoints[:,1],s=2)
plt.show()
```



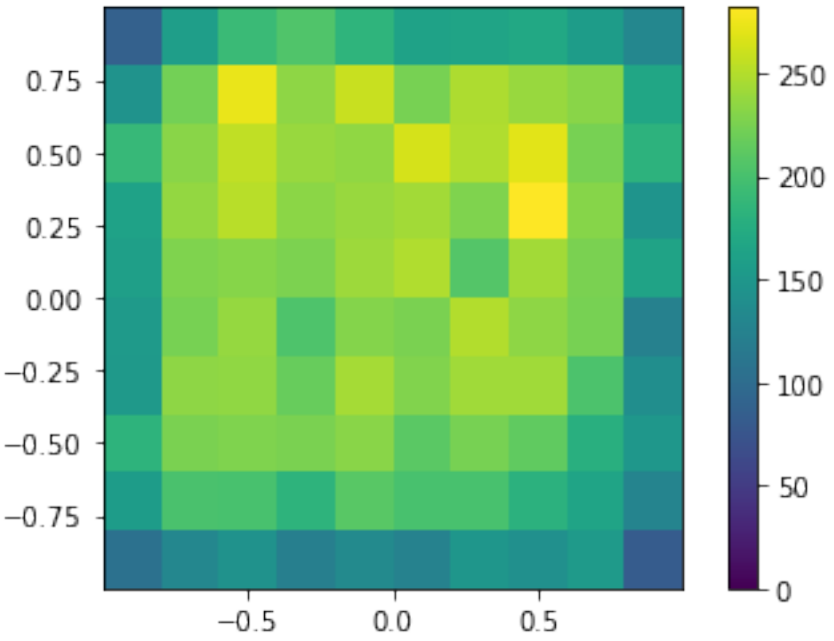
In our first version of this game the child discards any pebbles that end up outside of the square and stays in place before throwing the next. We can simulate this as follows.

```
def is_in_square(x):
    """Returns True if x is in the square (-1,1)^2."""
    return np.abs(x[0]) < 1 and np.abs(x[1]) < 1

def sample_next_position_naively(position,delta):
    """Keep trying a throw until it ends up in the square."""
    while True:
        next_position = position + delta*random_in_disk()
        if is_in_square(next_position):
            return next_position

def naive_markov_pebble(start,delta,N):
    """Simulates the number of hits in the naive Markov-chain version
    of the pebble game."""
    number_hits = 0
    position = start
    for i in range(N):
        position = sample_next_position_naively(position,delta)
        if is_in_circle(position):
            number_hits += 1
    return number_hits

trials = 20000
delta = 0.3
start = np.array([1,1])  # top-right corner
hits = naive_markov_pebble(start,delta,trials)
print(hits , "hits out of", trials, ", naive estimate of pi =", 4 * hits / trials␣
↪)
```

```
16872 hits out of 20000 , naive estimate of pi = 3.3744
```

This estimate does not look particularly good, reporting an estimate of π that is above target. Whether the deviation is significant or not, we can only decide after discussing the statistical errors. But comparing to the direct-sampling estimates we should be getting suspicious. This suspicion is confirmed when we look at a histogram of trials in the square:

```
def naive_markov_pebble_generator(start,delta,N):
    """Same as naive_markov_pebble but only yields the positions."""
    position = start
    for i in range(N):
        position = sample_next_position_naively(position,delta)
        yield position

# collect an array of points
trials = 20000
delta = 0.4
start = np.array([1,1])
testpoints = np.array(list(naive_markov_pebble_generator(start,delta,trials)))

# make a plot
fig, ax = plt.subplots()
ax.set_xlim(-1,1)
ax.set_ylim(-1,1)
ax.set_aspect('equal') # preserve aspect ratio of the square
plt.hist2d(testpoints[:,0],testpoints[:,1], bins=10, vmin=0)
plt.colorbar()
plt.show()
```



We clearly observe that the density of points near the boundary is lower compared to the central region. This means that the algorithm is not sampling properly from the uniform distribution on the square, but from some more complicated distribution. Can you think of a reason why this is the case?

A different choice in the case of an out-of-range throw is not to discard the pebble, but to have it retrieved and deposited at the location from where it was thrown before continuing to throw the next pebble. We call this a **rejection**. Note that there was already a pebble at that location from the previous throw, so for every rejection the pile of pebbles at the current location is increased with one. After the desired number of trials (i.e. pebbles deposited in the square), the number of hits within the circle can be counted. Note that the distribution of pebbles will be quite different compared to the direct sampling before: close to the boundary (more precisely, within distance δ) many of the pebbles will be part of piles, while in the central region no piles are to be found. Indeed, the chance that pebbles end up at the exact same position there is zero, just like in the direct sampling.

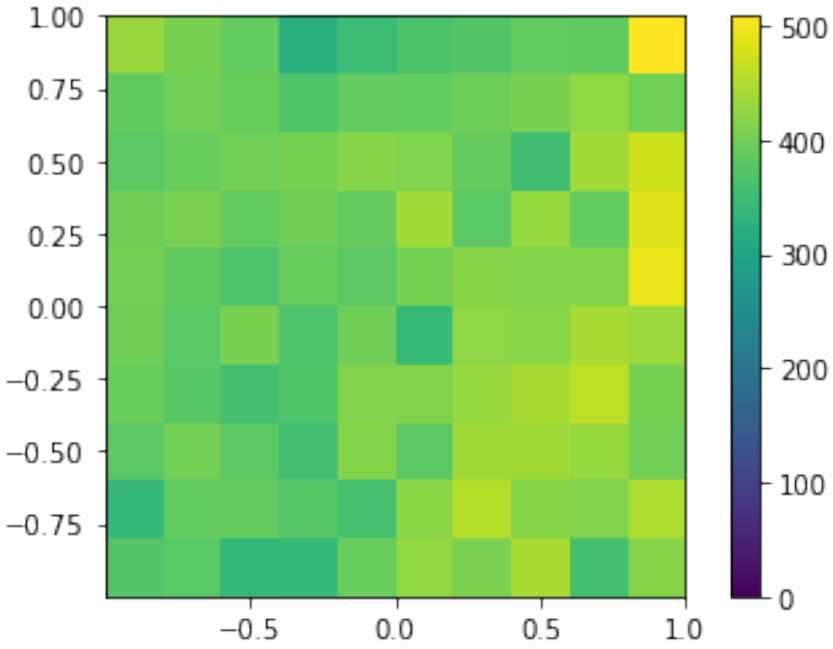
Let us simulate this process.

```
def sample_next_position(position,delta):
    """Attempt a throw and reject when outside the square."""
    next_position = position + delta*random_in_disk()
    if is_in_square(next_position):
        return next_position  # accept!
    else:
        return position  # reject!

def markov_pebble(start,delta,N):
    """Simulates the number of hits in the proper Markov-chain version of the␣
 ↪pebble game."""
    number_hits = 0
    position = start
    for i in range(N):
        position = sample_next_position(position,delta)
        if is_in_circle(position):
            number_hits += 1
    return number_hits

trials = 20000
delta = 0.3
start = [1,1]
hits = markov_pebble(start,delta,trials)
print(hits , "hits out of", trials, ", estimate of pi =", 4 * hits / trials )
```

```
15372 hits out of 20000 , estimate of pi = 3.0744
```

Judging by this single estimate we are already much closer to the exact value $\pi = 3.1415\ldots$, and we might be inclined to trust this algorithm. Let us also have a look at the histogram:

```
def markov_pebble_generator(start,delta,N):
    """Same as markov_pebble but only yields the positions."""
    position = start
    for i in range(N):
        position = sample_next_position(position,delta)
        yield position

# collect an array of points
trials = 40000
delta = 0.4
start = np.array([1,1])
testpoints = np.array(list(markov_pebble_generator(start,delta,trials)))

# make a plot
fig, ax = plt.subplots()
ax.set_xlim(-1,1)
ax.set_ylim(-1,1)
ax.set_aspect('equal') # preserve aspect ratio of the square
plt.hist2d(testpoints[:,0],testpoints[:,1], bins=10, vmin=0)
plt.colorbar()
plt.show()
```



Although the distribution over the square is not exactly uniform we do not observe a clear preference for the boundary versus center region anymore. Indeed we will convince ourselves later that with very many trials the distribution of pebbles approaches the uniform distribution, implying that the ratio of hits to trials approaches $\pi/4$. Note that the density of piles is identical to the previous algorithm, but apparently having multiple pebbles per pile in the boundary region exactly balances the reduced density of piles!

The method we see in action is a simple example of what is called the **Metropolis algorithm** for Markov-chain sampling: at each step of the Markov chain a random move is proposed and then accepted or rejected based on the outcome, and in case of a rejection you stay in place and wait. It should be clear that such a method is less efficient than direct sampling, because it can take many steps for the Markov chain to explore the full domain. This is particularly clear if we take the throwing range to δ to be very small. In this case it takes many throws even to leave the starting corner of the square. At the other extreme, if we take $\delta \gg 1$ almost all throws will result in rejections, leading to relatively few piles with many pebbles. Although in both cases we can prove convergence to the correct distribution, the rate of convergence depends on δ. A good rule of thumb is to choose the parameters in your Metropolis algorithm, in this case δ, such that the rejection rate is of order $1/2$, meaning that half of the proposed moves are rejected. You will test this for the pebble game in the *Exercise sheet*.

1.3 Markov-chain sampling: a discrete pebble game

To get a feeling for why the Metropolis algorithm with its rejection strategy works, let us simplify the problem even further. Instead of pebbles landing in a continuous set of locations in the unit square, let us make the domain discrete and only allow the pebbles to end up in a row of m sites. Accordingly we only allow Markov-chain moves to nearest-neighbor sites and rejections. The goal is to choose the move probabilities such that the distribution after many moves approaches the uniform distribution $p(i) = 1/m$ for $i = 1, \ldots, m$.

If the player is in position i we denote the probability of a move to $j \in \{i-1, i, i+1\}$ by $P(i \to j)$. It should be clear that the number of times it jumps $i \to i+1$ is the same as $i+1 \to i$ (ok, they can differ by 1). So after many moves these two events will happen with equal probability. For $i \to i-1$ to occur the player has to be in position i, which happens with probability $p(i)$, and select the move $i \to i+1$, with probability $P(i \to i+1)$. We thus obtain an equation relating the equilibrium distribution p to the move probabilities,

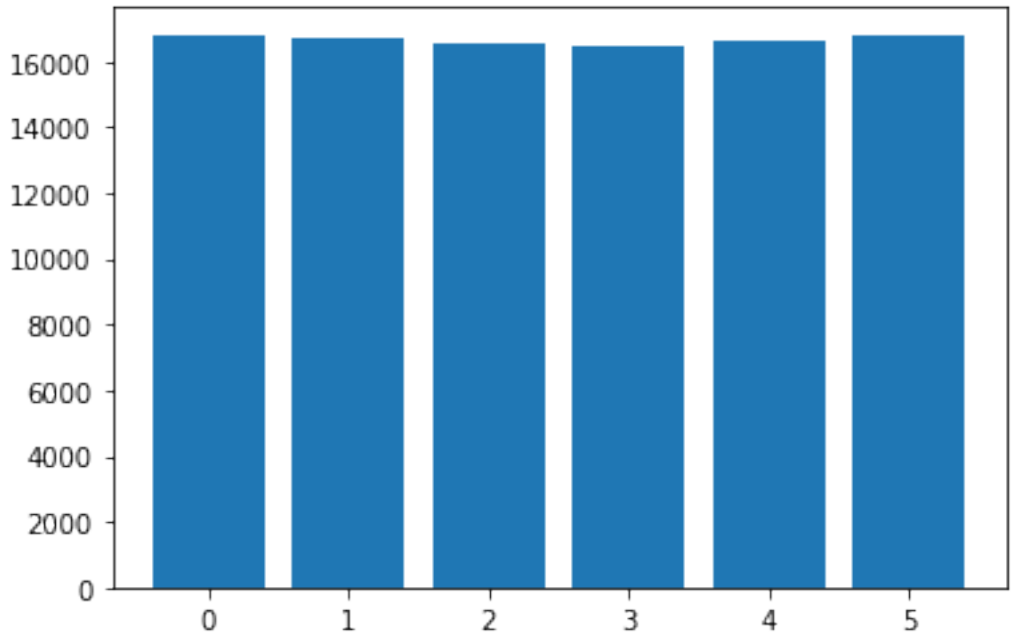
$$p(i)P(i \to i+1) = p(i+1)P(i+1 \to i). \tag{1.2}$$

For this to be satisfied for the uniform distribution $p(i) = 1/m$, we must have $P(i \to i+1) = P(i+1 \to i)$. In particular this implies we cannot choose $P(1 \to 2) = 1$, because then also $P(2 \to 1) = 1$ and the player would just oscillate back and forth between the first two sites. Although (1.2) does not have a unique solution, a simple solution

clearly is to take all $P(i \to i \pm 1) = \frac{1}{2}$. Since a player at site 1 cannot move to 0, we must therefore allow for a rejection, i.e. $P(1 \to 1) = \frac{1}{2}$, and similarly for site m. Let us verify the validity of this analysis with a simulation.

```
def simulate_discrete_pebble(m,N):
    """Simulate discrete pebble game of N moves on m sites,
    returning a histogram of pebbles per site."""
    histogram = np.zeros(m, dtype=int)
    position = 0
    for i in range(N):
        position += np.random.choice([-1,1])
        position = max(position, 0)
        position = min(position, m-1)
        histogram[position] += 1
    return histogram

histogram = simulate_discrete_pebble(6,100000)
print(histogram)
plt.bar(range(6),histogram)
plt.show()
```

```
[16816 16696 16569 16504 16619 16796]
```



Equation (1.2) is called **detailed balance**. In this simple example it is a necessary condition to reach the equilibrium distribution p. In general Markov chains this is not the case, but we will see (in the Section *Markov Chain Monte Carlo*) that imposing detailed balance is a convenient way to design simulations that have the desired equilibrium condition.

1.4 Further reading

For further reading on this week's material, have a look at Section 1.1 of [Kra06].

For a first-hand historical account of the birth of the Monte Carlo method have a look at the fascinating autobiography of Stanislaw Ulam, [Ula91].

Chapter 2

Probability theory & random variables

Last week was our first encounter with Monte Carlo simulations: the various incarnations of the pebble game used random numbers to estimate the ratio of the areas of a circle and a square. We have also witnessed that several factors affect the accuracy of the estimate: more samples means higher accuracy, while having independent samples like in the direct-sampling Monte Carlo method is better than the correlated samples of the Markov-chain approach. In order to reason properly about the simple examples and to be well equipped for the more involved problems we will encounter, it is essential to agree upon the probability theory background. This will be the focus of this week's lecture.

2.1 Elements of probability theory

2.1.1 Probability spaces, events, independence

The arena of probability theory is a **probability space** $(\Omega, \mathbb{P})$. Here Ω is the **sample space**, meaning the set of possible outcomes of an experiment, and $\mathbb{P}$ is a **probability measure** on Ω. A subset $A \subset \Omega$ is called an **event** and to any event $\mathbb{P}$ associates a real number $\mathbb{P}(A) \in [0, 1]$ telling us the probability of this event, i.e. the probability that the outcome of an experiment is in A. In particular it is required to satisfy $\mathbb{P}(\Omega) = 1$, $\mathbb{P}(\emptyset) = 0$, and for any sequence of disjoint events $A_1, A_2, \ldots$ (meaning $A_i \cap A_j = \emptyset$ whenever $i \neq j$),

$$\mathbb{P}(A_1 \cup A_2 \cup \cdots) = \mathbb{P}(A_1) + \mathbb{P}(A_2) + \cdots .$$

In words: the chance of any of a set of mutually exclusive events to happen is the sum of the chances of the individual events.

Example (direct-sampling pebble game)

The sample space of a single throw in the direct-sampling pebble game is the square $\Omega = (-1, 1)^2$ and the probability measure $\mathbb{P}$ is the measure on Ω determined by the probability density $p(\mathbf{x}) = 1/4$, i.e. for any subset $A \subset \Omega$ we had $\mathbb{P}(A) = \int_A p(\mathbf{x}) \mathrm{d}\mathbf{x}$. We were particularly interested in the event hit $= \{\mathbf{x} \in (-1, 1)^2 : \mathbf{x} \cdot \mathbf{x} < 1\}$ and we convinced ourselves that $\mathbb{P}(\text{hit}) = \pi/4$.

Example (die roll)

When rolling a 6-sided die, the sample space is $\Omega = \{1, 2, \ldots, 6\}$ and the probability measure is the one that satisfies $\mathbb{P}(\{i\}) = 1/6$ for any $i \in \Omega$. An example of an event is even $= \{2, 4, 6\} \subset \Omega$ which by formula (1) has probability $\mathbb{P}(\text{even}) = \mathbb{P}(\{1\}) + \mathbb{P}(\{2\}) + \mathbb{P}(\{3\}) = 1/2$.

Several properties of a probability space follow directly from the definition. If A and B are two events then

$$\begin{aligned} \mathbb{P}(\Omega \setminus A) &= 1 - \mathbb{P}(A) && \text{"probability of } A \text{ not happening"}, \\ \mathbb{P}(A \cup B) &= \mathbb{P}(A) + \mathbb{P}(B) - \mathbb{P}(A \cap B) && \text{"probability of } A \text{ or } B\text{"}, \\ \mathbb{P}(A \setminus B) &= \mathbb{P}(A) - \mathbb{P}(A \cap B) && \text{"probability of } A \text{ and not } B\text{"}, \\ A \subset B \quad \implies \quad \mathbb{P}(A) &\leq \mathbb{P}(B) && \text{"if } A \text{ implies } B \text{ then } B \text{ cannot have smaller probability"}. \end{aligned}$$

Side note: If we want to be mathematically rigorous we should be specifying what subsets $A \subset \Omega$ constitute proper events (the so-called σ-algebra), because some subsets of a continuous space can be just too wild to assign a probability to. We will not delve into these measure-theoretic issues as they will play little role in the probability spaces we consider.

An important concept in probability is that of **conditional probabilities**. If A and B are events such that B happens with non-zero probability $\mathbb{P}(B) > 0$ then the **conditional probability** of A given B is

$$\mathbb{P}(A|B) := \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)}.$$

You may check that $\mathbb{P}(\cdot|B)$ is a proper probability measure on Ω, which tells us what the probability of outcomes of an experiment is if you already know that event B happens.

Note that in general this measure is different from $\mathbb{P}$ itself, i.e. typically $\mathbb{P}(A) \neq \mathbb{P}(A|B)$, meaning that knowing that B happens changes the probability of A happening. Sometimes this is not the case: events A and B are said to be **independent** if

$$\mathbb{P}(A) = \mathbb{P}(A|B) \quad \iff \quad \mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B) \quad \iff \quad \mathbb{P}(B) = \mathbb{P}(B|A).$$

Example (die roll)

The events $\{2, 4, 6\}$ ("even") and $\{1, 2, 3, 4\}$ ("at most 4") are independent, but $\{1, 2, 3\}$ ("at most 3") is independent of neither of those.

The notion of independence can be extended to a sequence of events. Events $A_1, A_2, \dots, A_n$ are **independent** if

$$\mathbb{P}(\cap_{i \in I} A_i) = \prod_{i \in I} \mathbb{P}(A_i) \qquad \text{for all subsets } I \subset \{1, 2, \dots, n\}.$$

Example (direct-sampling pebble game)

When considering N throws in this game, the sample space is naturally given by the N-fold copy of the square $\Omega = \left((-1, 1)^2\right)^N = \{(\mathbf{x}_1, \dots, \mathbf{x}_N)\}$. Then the event of a hit in the kth trial is given by $\text{hit}_k = \{(\mathbf{x}_1, \dots, \mathbf{x}_N) : \mathbf{x}_k \cdot \mathbf{x}_k < 1\}$. The events $\text{hit}_1, \dots, \text{hit}_N$ are then independent: the probability of a hit in one trial is not affected by knowing the hits or misses in the $N - 1$ other trials.

2.1.2 Random variables

A (real) **random variable** X is a function from the sample space Ω to the real numbers $\mathbb{R}$. Sometimes one considers also functions into other spaces like $\mathbb{C}$ or $\mathbb{R}^n$, giving rise to complex random variables or random vectors, but for now let us concentrate on real random variables. Sometimes the sample space Ω is quite complicated (as it contains all details about an experiment) and we would like to limit ourselves to describing the information one gets from just looking at the random variable X. In other words we consider the **distribution** of X, which is a probability measure on $\mathbb{R}$ given by

$$\mathbb{P}(X \in B) := \mathbb{P}(\{x \in \Omega : X(x) \in B\}) \qquad \text{for every (decent) subset } B \subset \mathbb{R}.$$

A convenient way to characterize a distribution is via the **cumulative distribution function (CDF)**

$$F_X(x) := \mathbb{P}(X \leq x), \qquad x \in \mathbb{R}.$$

It exists for any random variable (discrete or continuous) and satisfies: $F_X(x)$ is non-decreasing and right-continuous whenever it jumps; $\lim_{x\to-\infty} F_X(x) = 0$; $\lim_{x\to\infty} F_X = 1$. This is really a 1-to-1 characterization: for any such real function there exists a random variable having that CDF. In fact we will see below an explicit way, called the **inversion method**, to construct such a random variable from the CDF. Two random variables (potentially on different probability spaces) are said to be **identically distributed** if they have the same distribution, or equivalently the same CDF.

2.1.3 Expectation value

Formally, given a random variable X, the **expectation value** or **mean** of X is

$$\langle X\rangle \equiv \mathbb{E}[X] := \int_\Omega X \mathrm{d}\mathbb{P},$$

but making sense of this requires abstract measure and integration theory. Luckily pretty much all random variables we will encounter are either **discrete** of **(absolutely) continuous** in which case the expectation value amounts to a sum or standard calculus integral (see below). In full generality it satisfies the important property of **linearity**: if X and Y are random variables on some probability space and $a, b \in \mathbb{R}$ then

$$\mathbb{E}[aX + bY] = a\,\mathbb{E}[X] + b\,\mathbb{E}[Y].$$

Note that this holds even when X and Y are highly dependent, making it an extremely powerful property that we will use again and again. Note also that if X is constant (or deterministic), i.e. $\mathbb{P}(X = x) = 1$ for some $x \in \mathbb{R}$, then $\mathbb{E}[X] = x$.

Example (Buffon's needle and Buffon's noodle)

A classic example in which the additivity of expectation values helps surprisingly is Buffon's needle experiment. Suppose the floor has parallel lines with an even spacing of distance 1 (think of a wooden floor with boards of width 1) and we drop a needle of length $\ell < 1$ from considerable height. What is the probability $\mathbb{P}(\text{across})$ that it will lie across a line? By parametrizing the position and orientation of the needle appropriately one may compute the appropriate integral to derive $\mathbb{P}(\text{across}) = 2\ell/\pi$. But there is a clever way to arrive at this answer by generalizing the problem somewhat. Let C be a 1-dimensional needle (or noodle) of any shape or size and denote by N_C the random variable counting the number of times C intersects a line when thrown. We can then subdivide C into smaller parts $C_1, \dots, C_k$ and note that the number of intersections just add up, $N_C = N_{C_1} + \cdots + N_{C_k}$. Even though $N_{C_1}, \dots, N_{C_k}$ are far from independent, linearity of expectation tells us that

$$\mathbb{E}[N_C] = \mathbb{E}[N_{C_1}] + \cdots + \mathbb{E}[N_{C_k}].$$

Suppose the total length of C is ℓ, and we divide C into $k = \ell/\epsilon$ parts that all approximately have the shape of a straight segment of small length ϵ up to rotation and translation. Then each of those parts contributes the same quantity to the right-hand side, giving $\mathbb{E}[N_C] = \frac{\ell}{\epsilon}\mathbb{E}[N_{C_1}] = c\ell$ for some universal constant c. In particular, $\mathbb{E}[N_C]$ only depends on the total length of the curve C. To determine the proportionality constant c, we consider a circular needle C of diameter 1 and length $\ell = \pi$, for which we know that always $N_C = 2$ and thus $\mathbb{E}[N_C] = 2$. This fixes $c = 2/\pi$. Returning to our straight needle C of length ℓ, we now know that $\mathbb{E}[N_C] = 2\ell/\pi$. But we also know that when $\ell < 1$, we can only have $N_C = 0$ or $N_C = 1$, meaning that the probability and expectation value agree

$$\frac{2\ell}{\pi} = \mathbb{E}[N_C] = 1 \cdot \mathbb{P}(\text{across}) + 0 \cdot \mathbb{P}(\text{across}) = \mathbb{P}(\text{across}).$$

Another important property holds for independent random variables. The notion of independence of random variables is similar to that of events: random variables $X_1, X_2, \dots, X_n$ are called **independent** if the distribution of one of the random variables is not affected by knowing any of the values of the other $n-1$ variables. Mathematically this is expressed as

$$\mathbb{P}(X_1 \in B_1, X_2 \in B_2, \dots, X_n \in B_n) = \prod_{i=1}^{n} \mathbb{P}(X_i \in B_i),$$

where the B_i are arbitrary subsets of $\mathbb{R}$. In this case

$$\mathbb{E}\Big[\prod_{i=1}^{n} X_i\Big] = \prod_{i=1}^{n} \mathbb{E}[X_i]. \qquad \text{(if } X_1, \dots, X_n \text{ are independent)} \tag{2.1}$$

In terms of the expectation value several other important quantities can be defined. The **variance** $\sigma_X^2 \equiv \mathrm{Var}(X)$ of X is defined as

$$\mathrm{Var}(X) = \mathbb{E}\Big[(X - \mathbb{E}[X])^2\Big].$$

Using the linearity of expectation this can be equivalently expressed as

$$\mathrm{Var}(X) = \mathbb{E}\Big[X^2 - 2\mathbb{E}[X]X + \mathbb{E}[X]^2\Big] = \mathbb{E}[X^2] - 2\mathbb{E}[X]\mathbb{E}[X] + \mathbb{E}[X]^2 = \mathbb{E}[X^2] - \mathbb{E}[X]^2.$$

An important property of the variance is that it is easily calculated for a sum of independent random variables,

$$\mathrm{Var}\Big(\sum_{i=1}^{n} X_i\Big) = \sum_{i=1}^{n} \mathrm{Var}(X_i). \qquad \text{(if } X_1, \dots, X_n \text{ are independent)}$$

We can check this in the case $n = 2$ (and conclude the general case by induction) as follows.

$$\begin{aligned}
\mathrm{Var}(X_1 + X_2) &= \mathbb{E}[(X_1 + X_2)^2] - \mathbb{E}[X_1 + X_2]^2 \\
&= \mathbb{E}[X_1^2] + 2\mathbb{E}[X_1 X_2] + \mathbb{E}[X_2^2] - \mathbb{E}[X_1]^2 - 2\mathbb{E}[X_1]\mathbb{E}[X_2] - \mathbb{E}[X_2]^2 \\
&= \mathbb{E}[X_1^2] + \mathbb{E}[X_2^2] - \mathbb{E}[X_1]^2 - \mathbb{E}[X_2]^2 \\
&= \mathrm{Var}(X_1) + \mathrm{Var}(X_2),
\end{aligned}$$

where we used (2.1) that $\mathbb{E}[X_1 X_2] = \mathbb{E}[X_1]\mathbb{E}[X_2]$ when X_1 and X_2 are independent. The **standard deviation** σ_X is of course just the square root of the variance $\sigma_X = \sqrt{\mathrm{Var}(X)}$.

2.1.4 Discrete random variables

A **discrete random variable** X is a random variable that takes on only a finite or countable number of values. In that case we can characterize its distribution by the **probability (mass) function** $p_X(x)$ given by

$$p_X(x) = \mathbb{P}(X = x).$$

In that case the expectation value of X is simply given by a sum

$$\mathbb{E}[X] = \sum_x x\, p_X(x).$$

If $g : \mathbb{R} \to \mathbb{R}$ is any function then we can also consider the expectation value of $g(X)$,

$$\mathbb{E}[g(X)] = \sum_x g(x)\, p_X(x).$$

Example (Bernoulli random variable)

This is probably the simplest discrete random variable. It has a single parameter $p \in [0, 1]$ and the random variable takes on only two values 0 and 1 with probability function

$$p_X(0) = 1 - p, \qquad p_X(1) = p.$$

Note that it has expectation value $\mathbb{E}[X] = p$ and variance $\mathrm{Var}(X) = p(1-p)$. Such random variables appear naturally when considering events on a probability space. If $A \subset \Omega$ is an event then the indicator function $\mathbf{1}_A : \Omega \to \mathbb{R}$ defines a random variable taking values 0 and 1, which is distributed as a Bernoulli random variable with $p = \mathbb{P}(A)$. We already encountered a random variable of this form in the direct-sampling pebble game, where we looked at an observable $\mathcal{O}$ that was indicator function on the event hit $= \{\mathbf{x} \cdot \mathbf{x} < 1\}$ with $p = \pi/4$.

It also makes sense to look at the joint distribution of several discrete random variables $X_1, \dots, X_n$. In that case one considers the **joint probability (mass) function**

$$p_{X_1,\dots,X_n}(x_1, \dots, x_n) = \mathbb{P}(X_1 = x_1, \cdots, X_n = x_n).$$

2.1.5 Continuous random variables

A random variable is **(absolutely) continuous** if there exists a **probability density function** $f_X : \mathbb{R} \to [0, \infty)$ such that

$$\mathbb{P}(X \leq s) = \int_{-\infty}^{s} f_X(x)\mathrm{d}x.$$

It follows directly that we have the properties

$$\begin{aligned} \mathbb{P}(X = x) &= 0 \text{ for } x \in \mathbb{R}, \\ \mathbb{P}(a \leq X \leq b) &= \int_a^b f_X(x)\mathrm{d}x, \\ \int_{-\infty}^{\infty} f_X(x)\mathrm{d}x &= 1, \\ f_X(x) &= F_X'(x). \end{aligned}$$

Now the expectation value is given by

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x\, f_X(x)\mathrm{d}x$$

and for any (decent) function $g : \mathbb{R} \to \mathbb{R}$,

$$\mathbb{E}[g(X)] = \int_{-\infty}^{\infty} g(x)\, f_X(x)\mathrm{d}x.$$

Example (uniform random variable)

The uniform random variable U on (a, b), $a < b$, has probability density function

$$f_U(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0, & \text{otherwise.} \end{cases}$$

Its expectation value is $\mathbb{E}[U] = (a + b)/2$ and variance is $\mathrm{Var}(U) = \frac{1}{12}(b - a)^2$.

Example (exponential random variable)

The exponential random variable X of rate λ has probability density function $f_X(x) = \lambda e^{-\lambda x}$. Its expectation value is $\mathbb{E}[X] = 1/\lambda$ and variance is $\mathrm{Var}(X) = 1/\lambda^2$.

As in the discrete case we may consider the **joint probability density functions** of several continuous random variables, e.g. $f_{X,Y}(x, y)$ such that

$$\mathbb{P}(X \leq s, Y \leq t) = \int_{-\infty}^{s} \mathrm{d}x \int_{-\infty}^{t} \mathrm{d}y f_{X,Y}(x, y).$$

In this case the random variables are independent if and only if the joint density function factorizes, $f_{X,Y}(x, y) = f_X(x)f_Y(y)$.

2.2 Constructing one random variable from another

2.2.1 Inversion method

Recall from the Section *Random variables* that knowing the cumulative distribution function fully characterizes the distribution of a random variable. The **inversion method** allows to explicitly construct such a random variable from a uniform random variable. To this end we consider the **(generalized) inverse distribution function**

$$F_X^{-1}(p) := \inf\{x \in \mathbb{R} : F_X(x) \geq p\}.$$

In the case of a continuous random variable with everywhere positive probability density this is simply the inverse function of F_X, but it is perfectly well-defined even when F_X has jumps, like in the case when X is discrete. If U is a uniform random variable, then we claim that $F_X^{-1}(U)$ and X are identically distributed, meaning that $F_X^{-1}(U)$ has CDF F_X. This follows from the fact that for any $p \in [0,1]$ we have $F_X^{-1}(p) \leq x$ if and only if $p \leq F_X(x)$, which directly implies

$$\mathbb{P}(F_X^{-1}(U) \leq x) = \mathbb{P}(U \leq F_X(x)) = F_X(x).$$

Example (triangular distribution)

Consider the continuous random variable X on $(0,1)$ with probability density function $f_X(x) = 2x$. Its CDF is given by $F_X(x) = \int_0^x f_X(y)\mathrm{d}y = x^2$. Therefore $F_X^{-1}(U) = \sqrt{U}$ and X are identically distributed.

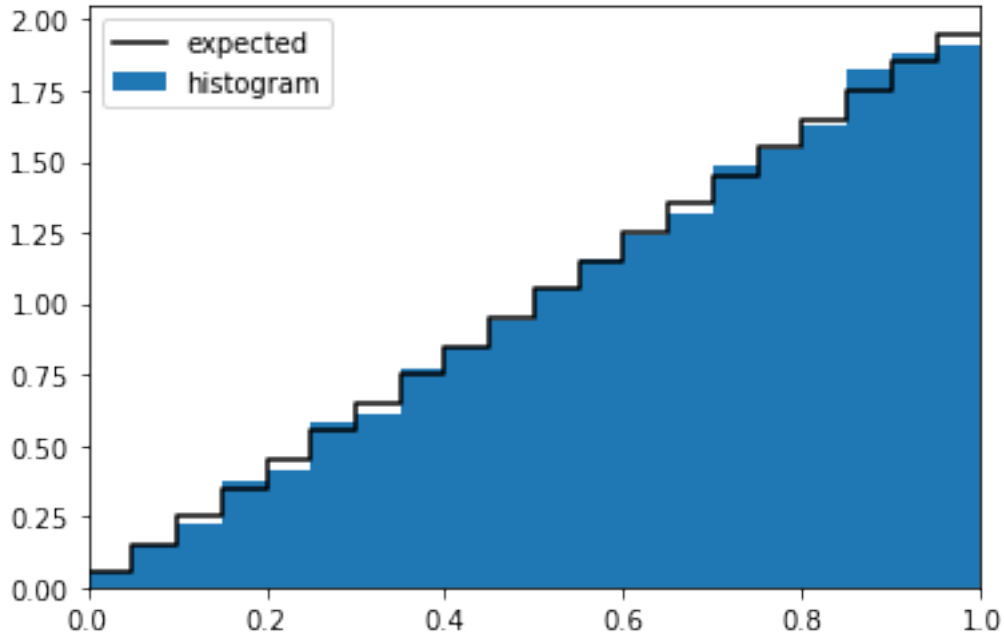
Since the uniform random variable U is very simple to sample on the computer, this gives a powerful method to sample from other distributions as long as the inverse distribution function is manageable. For instance, we can verify the previous example as follows.

```
import numpy as np
import matplotlib.pylab as plt
from scipy.integrate import quad

rng = np.random.default_rng()
%matplotlib inline

def inversion_sample(f_inverse):
    '''Obtain an inversion sample based on the inverse-CDF f_inverse.'''
    return f_inverse(rng.random())

def compare_plot(samples,pdf,xmin,xmax,bins):
    '''Draw a plot comparing the histogram of the samples to the expectation␣
 ↪coming from the pdf.'''
    xval = np.linspace(xmin,xmax,bins+1)
    binsize = (xmax-xmin)/bins
    # Calculate the expected numbers by numerical integration of the pdf over the␣
 ↪bins
    expected = np.array([quad(pdf,xval[i],xval[i+1])[0] for i in range(bins)])/
 ↪binsize
    measured = np.histogram(samples,bins,(xmin,xmax))[0]/(len(samples)*binsize)
    plt.plot(xval,np.append(expected,expected[-1]),"-k",drawstyle="steps-post")
    plt.bar((xval[:-1]+xval[1:])/2,measured,width=binsize)
    plt.xlim(xmin,xmax)
    plt.legend(["expected","histogram"])
    plt.show()

def f_inv_triangular(p):
    return np.sqrt(p)

def pdf_triangular(x):
```

(continues on next page)

(continued from previous page)

```
    return 2*x

compare_plot([inversion_sample(f_inv_triangular) for _ in range(10000)],pdf_
↪triangular,0,1,20)
```



In the *Exercise sheet* you will implement several examples of the inversion method.

2.2.2 Change of variables

If X is a random variable with range $D \subset \mathbb{R}$ and $g : D \to \mathbb{R}$ is strictly increasing and invertible, then $Y = g(X)$ is a random variable with CDF

$$F_Y(y) = \mathbb{P}(g(X) \leq y) = \mathbb{P}(X \leq g^{-1}(y)) = F_X(g^{-1}(y)).$$

If in addition X is (absolutely) continuous and g is differentiable, then because $f_Y(y) = F_Y'(y)$, we can take derivatives on both sides to relate the probability density functions via

$$f_Y(y) = f_X(g^{-1}(y))\frac{\mathrm{d}g^{-1}(y)}{\mathrm{d}y} \qquad \Longleftrightarrow \qquad f_Y(g(x))\frac{\mathrm{d}g(x)}{\mathrm{d}x} = f_X(x).$$

Note that this is nothing but a change of variables $y = g(x)$ in the density $f_X(x)\mathrm{d}x$, turning it into $f_Y(y)\mathrm{d}y$.

If you remember how to perform multivariate change of variables in integrals, it should be clear how to generalize this to joint probability distribution functions. For instance, if X and Y are continuous random variables and the range of (X, Y) is $D \subset \mathbb{R}^2$ and $T : D \to \mathbb{R}^2$ is an invertible differentiable mapping, we can consider the pair of random variables $(U, V) = T(X, Y)$. The joint probability densities $f_{U,V}(u, v)\mathrm{d}u\mathrm{d}v$ and $f_{X,Y}(x, y)\mathrm{d}x\mathrm{d}y$ should then agree, implying that they differ by the Jacobian of T,

$$f_{U,V}(T(x, y))\left|\frac{\mathrm{d}u}{\mathrm{d}x}\frac{\mathrm{d}v}{\mathrm{d}y} - \frac{\mathrm{d}v}{\mathrm{d}x}\frac{\mathrm{d}u}{\mathrm{d}y}\right| = f_{X,Y}(x, y),$$

where we used the notation $T(x, y) = (u(x, y), v(x, y))$.

Example (uniform from pair of exponentials)

Consider X and Y to be two independent exponential random variables of rate 1 and let $T : (0, \infty)^2 \to (0, 1) \times (0, \infty)$ be given by $T(x, y) = (\frac{x}{x+y}, x + y)$. Let us compute the PDF of $(U, V) = T(X, Y)$. The inverse mapping

is $T^{-1}(u,v) = (uv, (1-u)v)$, which has Jacobian $\left|\frac{\mathrm{d}x}{\mathrm{d}u}\frac{\mathrm{d}y}{\mathrm{d}v} - \frac{\mathrm{d}y}{\mathrm{d}u}\frac{\mathrm{d}x}{\mathrm{d}v}\right| = v$. Hence $f_{U,V}(u,v) = v f_{X,Y}(T^{-1}(u,v)) = ve^{-v}$. Since this is of factorized form $f_{U,V}(u,v) = f_U(u)f_V(v)$, we conclude that U is a uniform random variable in $(0,1)$ and V is independently distributed with density $f_V(v) = ve^{-v}$ on $(0,\infty)$.

Change of variables can be quite useful for simulation purposes. If a desired (univariate or multivariate) distribution is difficult to sample directly a clever change of variables can turn it into an simpler distribution, that may be amenable to sampling through the inversion method. We will encounter an example of this in the *Exercise sheet* in order to sample a pair of independent normal random variables (the Box-Muller transformation).

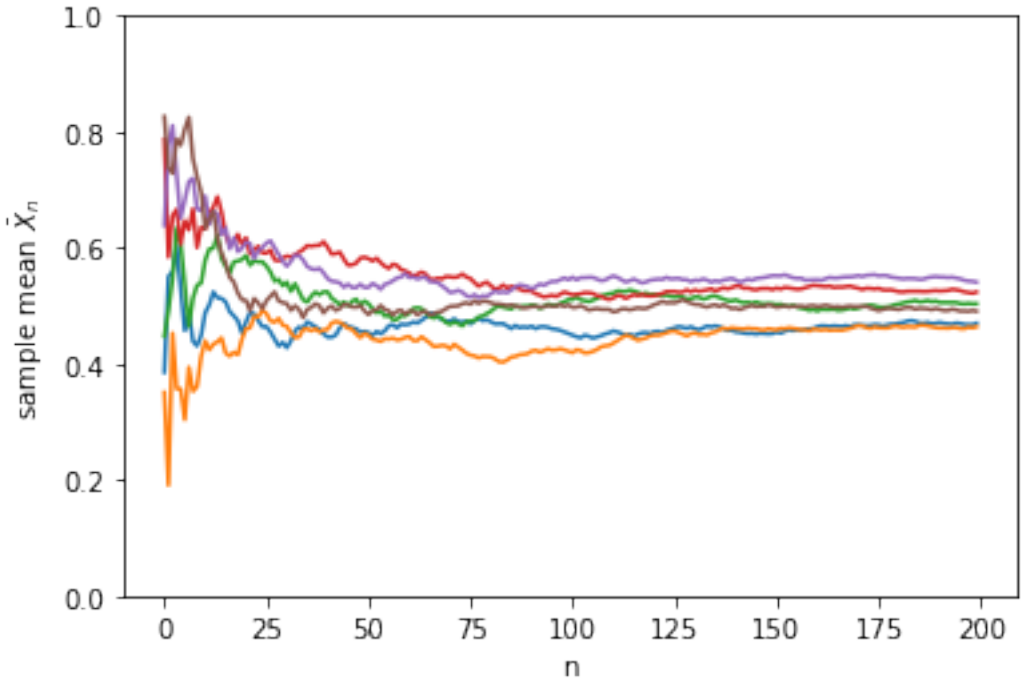
2.3 Laws of large numbers, central limit theorem

Let X be a real random variable and $X_1, X_2, \ldots$ a sequence of independent, identically distributed (abbreviated **i.i.d.**) random variables on some probability space with the same distribution as X. The **sample mean** is defined to be

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

The plot below illustrates the sample mean $\bar{X}_n$ as function of n in the case that X is a uniform random variable in $(0,1)$. Lines of different colors correspond to repetitions of the sampling.

```
length = 200
for _ in range(6):
    iid_sequence = rng.random(length)
    sample_mean = np.cumsum(iid_sequence) / [i+1 for i in range(length)]
    plt.plot(sample_mean)

plt.ylim(0,1)
plt.xlabel("n")
plt.ylabel(r"sample mean $\bar{X}_n$")
plt.show()
```



Clearly the sample mean $\bar{X}_n$ is still a random variable as its value varies from one sampling to another, but as $n \to \infty$ it becomes less random and apparently converges to $\mathbb{E}[X] = 1/2$. But what does it mean for a random variable to

converge to a real number? We could look at the expectation value, but this is unsatisfactory because

$$\mathbb{E}[\bar{X}_n] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X_i] = \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[X] = \mathbb{E}[X]$$

does not even depend on n. Perhaps it is better to look at the variance $\mathrm{Var}(\bar{X}_n)$. Using that the X_i are all independent we may use the sum rule for the variance to get

$$\mathrm{Var}(\bar{X}_n) = \frac{1}{n^2}\mathrm{Var}(\sum_{i=1}^{n}X_i) = \frac{1}{n^2}\sum_{i=1}^{n}\mathrm{Var}(X_i) = \frac{1}{n}\mathrm{Var}(X).$$

In particular we observe the fact, probably well-known to you, that the standard deviation of the sample mean decreases as

$$\sigma_{\bar{X}_n} = \frac{\sigma_X}{\sqrt{n}}.$$

We can combine this with the Bienaymé–Chebyshev inequality

$$\mathbb{P}(|Y - \mathbb{E}[Y]| \geq \epsilon) \leq \frac{\sigma_Y^2}{\epsilon^2} \qquad \text{for } \epsilon > 0$$

that holds for any random variable Y with finite variance to derive the **(weak) law of large numbers** for the sample mean

$$\mathbb{P}(|\bar{X}_n - \mathbb{E}[X]| > \epsilon) \leq \frac{\sigma_X^2}{n\epsilon^2} \qquad \text{for } \epsilon > 0. \tag{2.2}$$

In other words, $\bar{X}_n$ is very likely to approximate $\mathbb{E}[X]$ to an accuracy ϵ if we take the sample size $n \gg \sigma_X^2/\epsilon^2$.

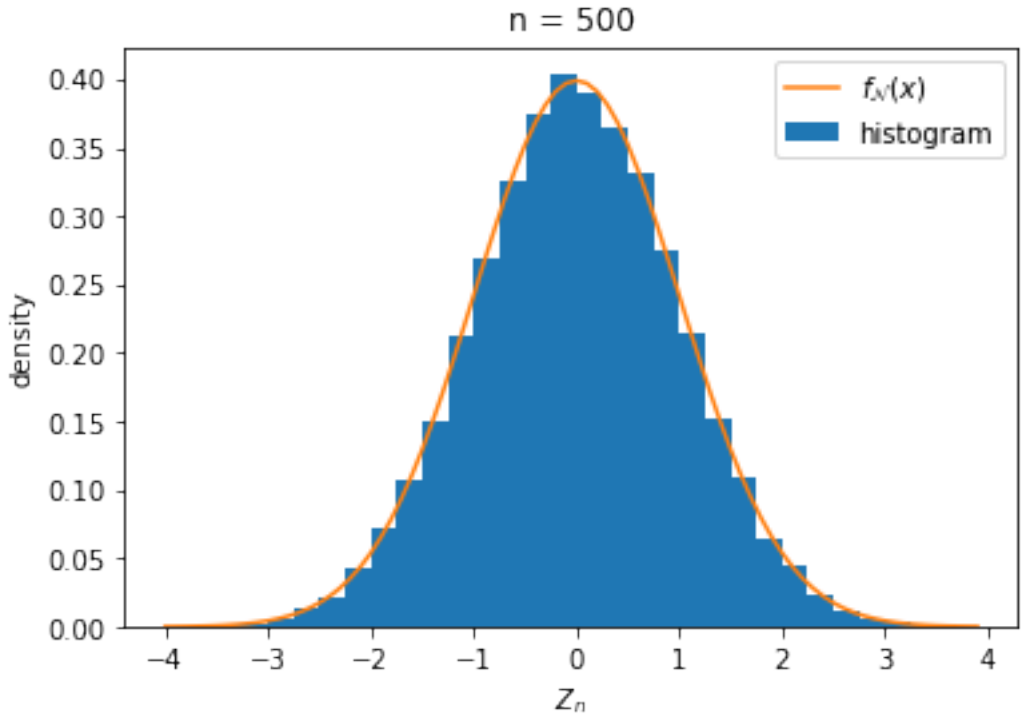
2.3.1 Central limit theorem

The law of large numbers tells us that the sample mean approximates the mean up to an error of order $1/\sqrt{n}$. We can be more precise and ask what is the distribution of the error in units of $\sigma_X/\sqrt{n}$ when n becomes large,

$$Z_n := \frac{\sqrt{n}}{\sigma_X}(\bar{X}_n - \mathbb{E}[X]).$$

In the case when X is uniformly distributed on $(0,1)$ (with mean $\mathbb{E}[X] = 1/2$ and standard deviation $\sigma_X = 1/\sqrt{12}$) the distribution of Z_n looks like this:

```
def gaussian(x):
    return np.exp(-x*x/2)/np.sqrt(2*np.pi)

length = 500
sample_means = np.array([np.mean(rng.uniform(0,1,length)) for _ in range(40000)])
# the mean of X is 0.5 and the standard deviation is 1/np.sqrt(12)
normalized_sample_means = (sample_means - 0.5) * np.sqrt(12*length)
plt.hist(normalized_sample_means,bins=np.arange(-4,4,0.25),density=True)
xrange = np.arange(-4,4,0.1)
plt.plot(xrange,gaussian(xrange))
plt.xlabel(r"$Z_n$")
plt.ylabel("density")
plt.title("n = {}".format(length))
plt.legend([r"$f_{\mathcal{N}}(x)$","histogram"])
plt.show()
```



The orange curve is the standard Gaussian, i.e. the probability density function of a **normal** random variable $\mathcal{N}$ of mean 0 and standard deviation 1,

$$f_{\mathcal{N}}(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}.$$

To be precise the **central limit theorem (CLT)** states that, provided $\mathbb{E}[|X|]$ and $\mathrm{Var}(X)$ are finite, the CDF of Z_n converges to that of $\mathcal{N}$ as $n \to \infty$,

$$\lim_{n\to\infty} F_{Z_n}(x) = F_{\mathcal{N}}(x) \qquad \text{for all } x \in \mathbb{R}.$$

Such a convergence of random variables is called **convergence in distribution**.

It is instructive to see how the central limit theorem can be proved. To this end we introduce the **characteristic function** $\varphi_X(t)$ associated to any random variable X as

$$\varphi_X(t) = \mathbb{E}[e^{itX}] \qquad \text{for } t \in \mathbb{R}.$$

Due to absolute convergence of the sum or integral appearing in the expectation value, it is a continuous function satisfying $|\varphi_X(t)| \leq 1$ and it completely characterizes the distribution of X. If X has finite mean and finite variance, which we assume, then $\varphi_X(t)$ is twice continuous differentiable. Set $\hat{X} = (X - \mathbb{E}(X))/\sigma_X$ to be the normalized version of X, such that $\mathbb{E}[\hat{X}] = 0$ and $\mathrm{Var}(\hat{X}) = \mathbb{E}[\hat{X}^2] = 1$ and $Z_n = \frac{1}{\sqrt{n}} \sum_{i=1}^n \hat{X}_i$. Then we can Taylor expand the characteristic function of $\hat{X}$ around $t = 0$ to obtain

$$\varphi_{\hat{X}}(t) = \mathbb{E}[1 + it\hat{X} - \tfrac{1}{2}t^2\hat{X}^2 + \cdots] = 1 - \tfrac{1}{2}t^2 + \cdots.$$

On the other hand the characteristic function of Z_n is

$$\varphi_{Z_n}(t) = \mathbb{E}\Big[\exp(it\tfrac{1}{\sqrt{n}}\sum_{i=1}^n \hat{X}_i)\Big] = \prod_{i=1}^n \mathbb{E}\Big[\exp(it\tfrac{1}{\sqrt{n}}\hat{X}_i)\Big] = \varphi_{\hat{X}}(\tfrac{1}{\sqrt{n}}t)^n,$$

where we used (2.1).

Plugging in the Taylor expansion gives

$$\lim_{n\to\infty} \varphi_{Z_n}(t) = \lim_{n\to\infty} \left(1 - \tfrac{t^2}{2n} + \cdots\right)^n = e^{-t^2/2}.$$

But the right-hand side we recognize as the characteristic function

$$\varphi_{\mathcal{N}}(t) = \mathbb{E}[e^{it\mathcal{N}}] = \int_{-\infty}^{\infty} \mathrm{d}x e^{itx} \frac{1}{\sqrt{2\pi}} e^{-x^2/2} = e^{-t^2/2}$$

of the standard normal distribution. Convergence of characteristic functions can be shown (Lévy's continuity theorem) to be equivalent to convergence in distribution and therefore this proves the central limit theorem.

Note that the central limit theorem does **not** hold when $\mathrm{Var}(X) = \infty$ and a variety of other limit distributions (called **stable distributions**) can arise in the large-n limit. You will investigate this further in the *Exercise sheet*.

Chapter 3

Direct-sampling Monte Carlo integration

After last week's recap of probability theory and random variables, it is time to apply this knowledge to understand the applicability and shortcomings of Monte Carlo methods. Before heading in that direction let us add one more method to our toolbox to sample random variables.

3.1 Acceptance-Rejection sampling

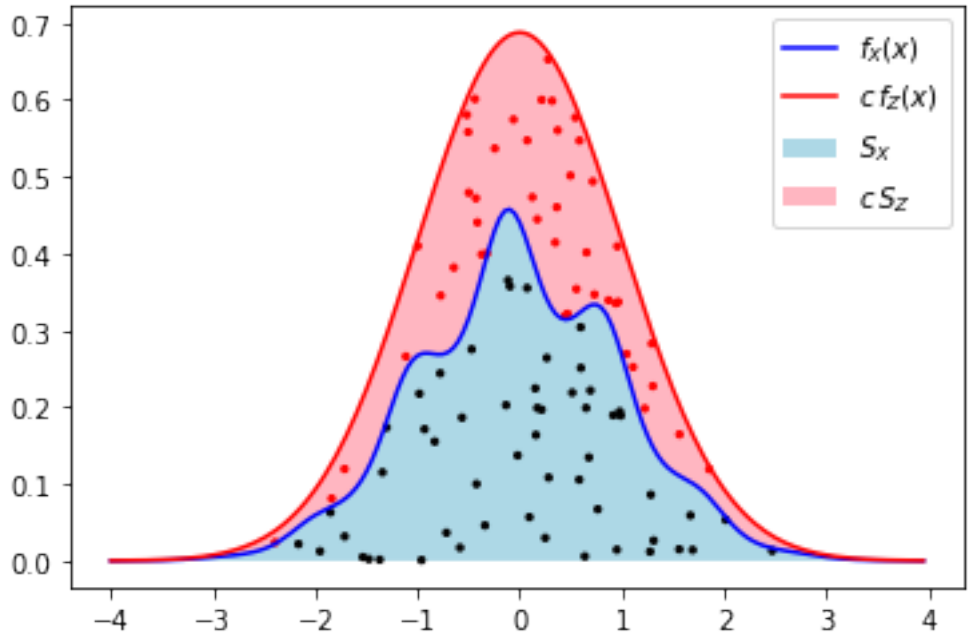
Suppose we wish to sample a continuous random variable X with probability density function $f_X(x)$. Last week we have seen that one way to achieve this is via the *Inversion method* by applying the inverse cumulative distribution function F_X^{-1} to a uniform random variable U on $(0,1)$. But this requires that we have an efficient way to calculate $F_X^{-1}(p)$ for any real number $p \in (0,1)$. In many case a closed-form expression for F_X is not available, let alone for its inverse, so one has to resort to costly numerical approximations. Applying the inversion method to joint distributions of several random variables becomes even more challenging. We also discussed *Change of variables* as a method to generate particular distributions, but discovering suitable transformations requires some luck and ingenuity.

A more generally applicable method, that may or may not be efficient, is **acceptance-rejection sampling**. Suppose there is another random variable Z with density $f_Z(x)$ that we already know how to sample efficiently and that $f_Z(x)$ and $f_X(x)$ are not too dissimilar. More precisely, suppose we can find a $c \geq 1$ such that $f_X(x) \leq c\, f_Z(x)$ for all $x \in \mathbb{R}$. Then we can sample X using the following simple algorithm. Here `sample_z` is a function sampling Z and `accept_probability` is the function $x \to f_X(x)/(cf_Z(x))$.

```
import numpy as np
rng = np.random.default_rng()

def sample_acceptance_rejection(sample_z,accept_probability):
    while True:
        x = sample_z()
        if rng.random() < accept_probability(x):
            return x
```

Why does this work? It is useful to assign a geometric interpretation to this sampling procedure (see the figure below). Let U be uniform random in $(0,1)$ independent of X. The point $(X,Y) = (X, Uf_X(X))$ is then uniform in the region $S_X = \{(x,y) : x \in \mathbb{R},\, 0 < y < f_X(x)\}$ enclosed by the graph of f_X and the x-axis. To see this, we note that $(x,u) \to (x,y) = (x, uf_X(x))$ is a differentiable and invertible mapping (at least where $f_X(x) \neq 0$) and we can therefore relate the joint densities by change of variables: $f_X(x)\mathrm{d}x\mathrm{d}u = \mathrm{d}x\mathrm{d}y$ for $(x,y) \in S_X$, so $f_{X,Y} = \mathbf{1}_{S_X}$. One way to sample (X,Y) is by sampling from a larger but simpler region containing S_X and rejecting any sample that is not in S_X. If we have another random variable Z and constant $c \geq 1$ satisfying $f_X(x) \leq c\, f_Z(x)$ then a natural region is $cS_Z = \{(x,cy) : x \in \mathbb{R}, 0 < y < f_Z(x)\}$, because $S_X \subset cS_Z$. If we sample Z and an independent U' uniform in $(0,1)$, then the conditional distribution of $(Z, U'cf_Z(Z))$ given the event $(Z, U'cf_Z(Z)) \in S_X$ is uniform in S_X. On that event Z is therefore identically distributed to X. The natural way to sample a random variable conditionally on an event, is simply to repeat the sampling until the event occurs. Once we know $Z = z$ this event

occurs with probability $\mathbb{P}(U'cf_Z(Z) < f_X(Z)|Z = z) = f_X(z)/(cf_Z(z))$, so we should repeatedly sample Z and then accept the value with this probability. This is precisely what the algorithm above does.



How efficient is this algorithm? At each iteration the acceptance probability is

$$\mathbb{P}(U'cf_Z(Z) < f_X(Z)) = \int_{-\infty}^{\infty} \mathrm{d}z f_Z(z)\mathbb{P}(U'cf_Z(Z) < f_X(Z)|Z = z) = \int_{-\infty}^{\infty} \mathrm{d}z f_X(z)/c = 1/c.$$

This means that the number of iterations needed is a random number $N \geq 1$ with probability mass function $p_N(n) = \frac{1}{c}\left(1 - \frac{1}{c}\right)^{n-1}$, i.e. a **geometric random** variable with parameter $1 - \frac{1}{c}$. The expected number of samples required is then simply $\mathbb{E}[N] = c$. So we should make sure c is not too large, since the average runtime of a simulation that relies heavily on sampling X scales linearly with c.

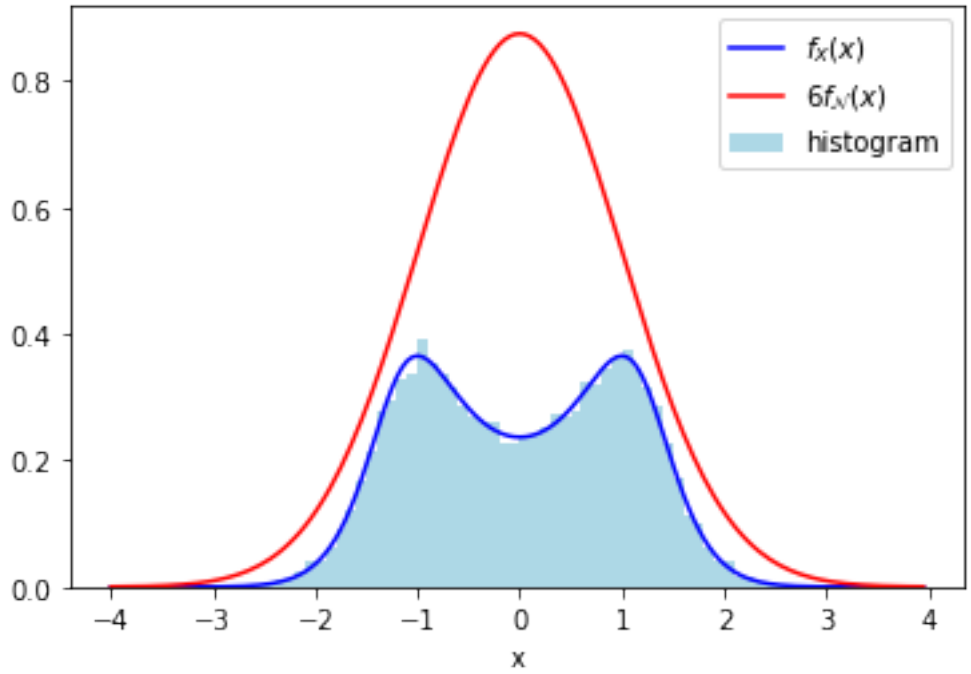
A bonus of this method is that you only really need to know the probability density $f_X(x)$ up to an overall constant, the conditioning in the method takes care of the proper normalization. As an example, suppose we wish to sample X with distribution $f_X(x) = C/\cosh(x^2 - 1)$, but we do not know the normalization constant. However, we can convince ourselves that $1/\cosh(x^2 - 1) \leq 6f_{\mathcal{N}}(x)$, so we can apply acceptance-rejection to a standard normal random variable $\mathcal{N}$ with $c = 6C$. Luckily C cancels out in the acceptance probability $f_X(x)/(cf_{\mathcal{N}}(x))$, so we can proceed as follows.

```
import matplotlib.pylab as plt
%matplotlib inline

def difficult_pdf(x):
    return 1/np.cosh(x**2-1)

def gaussian(x):
    return np.exp(-x**2/2)/np.sqrt(2*np.pi)

def accept(x):
    return difficult_pdf(x)/(6*gaussian(x))

samples = [sample_acceptance_rejection(rng.normal,accept) for _ in range(20000)]

# for plotting purposes let's compute C
from scipy.integrate import quad
normalization_C = 1/quad(difficult_pdf, -20, 20)[0]

xrange = np.arange(-4,4,0.05)
```

(continues on next page)

(continued from previous page)

```
plt.plot(xrange,normalization_C*difficult_pdf(xrange),'-b')
plt.plot(xrange,normalization_C*6*gaussian(xrange),'-r')
plt.hist(samples,bins=np.arange(-4,4,0.1),density=True,color="lightblue")
plt.legend([r"$f_X(x)$",r"$6f_{\mathcal{N}}(x)$","histogram"])
plt.xlabel("x")
plt.show()
```



The method generalizes straightforwardly to discrete random variables (where one needs a bound $p_X(x) \leq c\, p_Z(x)$ on the probability mass functions) and to multivariate distributions. Note however that as the number of dimensions increases it will become increasingly difficult to keep the constant c relatively small.

3.2 Monte Carlo integration: direct-sampling

3.2.1 Integral as an expectation value

A straightforward application of Monte Carlo methods is the evaluation of integrals of the form

$$I = \int_{-\infty}^{\infty} g(y) f_Y(y) \mathrm{d}y,$$

where $f_Y(y)$ is some function integrating to $\int_{-\infty}^{\infty} f_Y(y)\mathrm{d}y = 1$. As a special case we can consider $f_Y(y) = \frac{1}{b-a}\mathbf{1}_{\{a<y<b\}}$ to be the probability density of the uniform random variable on (a, b), such that the integral simply becomes the definite integral

$$I = \frac{1}{b-a}\int_a^b g(y)\mathrm{d}y.$$

The basic idea of Monte Carlo integration is to interpret the integral as the expectation value of the random variable $X = g(Y)$,

$$I = \mathbb{E}[g(Y)] = \mathbb{E}[X].$$

3.2.2 Error estimation

We know from last week's considerations how to approximate this expectation value by considering the **sample mean**

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i.$$

It is an **unbiased estimator** of $\mathbb{E}[X]$ in the sense that $\mathbb{E}[\bar{X}_n] = \mathbb{E}[X]$. According to the **weak law of large numbers** the sample mean $\bar{X}_n$ indeed approaches $\mathbb{E}[X]$ in the sense of convergence in probability. If we know the variance σ_X^2 one also obtains the upper-bound (2.2) on the error ϵ as

$$\mathbb{P}(|\bar{X}_n - \mathbb{E}[X]| > \epsilon) \leq \frac{\sigma_X^2}{n\epsilon^2} \qquad \text{for } \epsilon > 0,\ n \geq 1.$$

Equivalently

$$\mathbb{P}\left(\bar{X}_n - \frac{\sigma_X}{\sqrt{\delta n}} \leq \mathbb{E}[X] \leq \bar{X}_n + \frac{\sigma_X}{\sqrt{\delta n}}\right) \geq 1 - \delta \qquad \text{for any } \delta > 0,\ n \geq 1.$$

For example, after an experiment with n samples we can say that with probability **at least** 0.682 the mean will lie within distance $1.77\sigma_X/\sqrt{n}$ of our estimate $\bar{E}_n$. This bound holds for any n, even $n = 1$, but is quite conservative when n is large, because it does not take into account that the error approaches a normal distribution. In that limit the *Central limit theorem* gives a more tight bound, because we know

$$\mathbb{P}(|\bar{X}_n - \mathbb{E}[X]| > \frac{\sigma_X}{\sqrt{n}}\epsilon) \quad \xrightarrow{n\to\infty} \quad \mathbb{P}(|\mathcal{N}| > \epsilon) = \frac{2}{\sqrt{2\pi}}\int_{\epsilon}^{\infty} e^{-x^2/2}\mathrm{d}x.$$

This gives the more accurate **1σ-confidence level**, stating that in the limit of large n with **precisely** $\mathbb{P}(|\mathcal{N}| > 1) = 0.682\ldots$ probability we have that

$$\bar{X}_n - \frac{\sigma_X}{\sqrt{n}} \leq \mathbb{E}[X] \leq \bar{X}_n + \frac{\sigma_X}{\sqrt{n}}. \qquad (1\sigma \text{ or } 68.2\% \text{ confidence when CLT holds})$$

So knowing that the central limit theorem holds allows us to decrease the error interval by a factor 1.77 in this case.

Of course, we are cheating a bit here, because generally we do not know the variance σ_X^2, so it has to be estimated as well. An **unbiased estimator** for the variance is given by **sample variance**

$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)^2,$$

which differs from the usual variance of the sample sequence $X_1, \ldots, X_n$ by a factor $n/(n-1)$. This additional factor ensures that

$$\mathbb{E}[s_n^2] = \sigma_X^2,$$

as you may check by expanding the parentheses and using linearity of expectation. A variation of the weak law of large numbers tells us that s_n really converges in probability to σ_X as $n \to \infty$. We may thus substitute the variance by the sample variance in the error estimate without changing the central limit theorem. Hence with the **1σ-confidence level** can also be written as

$$\bar{X}_n - \frac{s_n}{\sqrt{n}} \leq \mathbb{E}[X] \leq \bar{X}_n + \frac{s_n}{\sqrt{n}}. \qquad (1\sigma \text{ or } 68.2\% \text{ confidence when CLT holds})$$

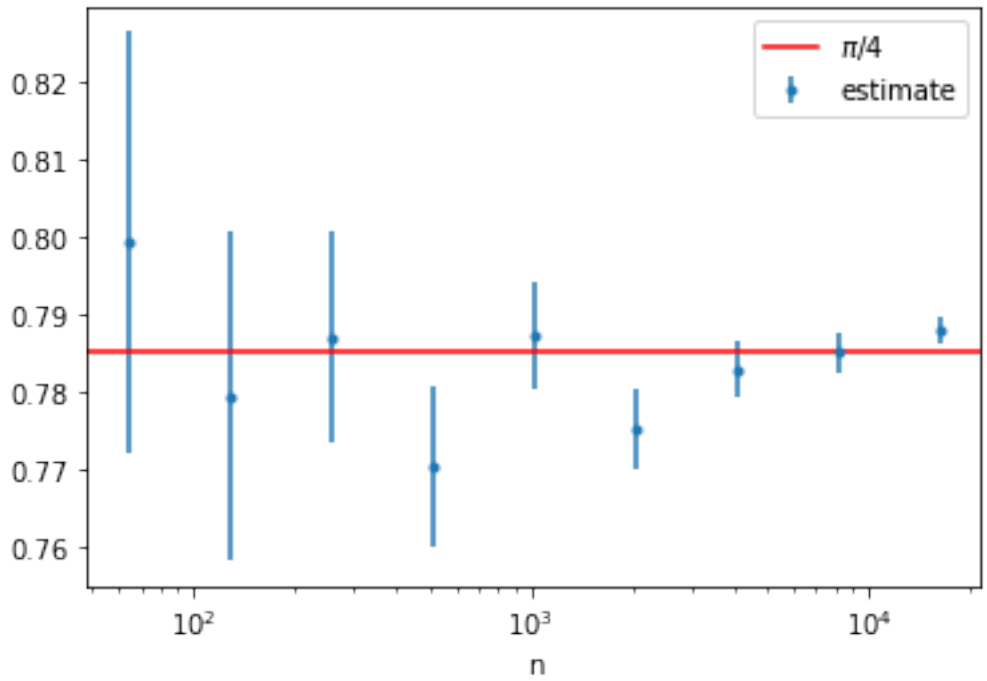
The result of an experiment is thus generally reported as $\mathbb{E}[X] = \bar{X}_n \pm \frac{s_n}{\sqrt{n}}$.

Let's consider an example that is directly related to the direct-sampling pebble game from the first chapter by considering the function $g(x) = \sqrt{1-x^2}$ on $(0,1)$ and estimating $\pi/4$ by Monte Carlo integrating

$$\frac{\pi}{4} = \int_{-\infty}^{\infty} \sqrt{1-x^2}\,\mathbf{1}_{0<x<1}\mathrm{d}x = \mathbb{E}[g(U)], \qquad (3.1)$$

where U is a uniform random variable on $(0,1)$.

```python
def estimate_expectation(sampler,n):
    '''Compute beste estimate of mean and 1-sigma error with n samples.'''
    samples = [sampler() for _ in range(n)]
    return np.mean(samples), np.std(samples)/np.sqrt(n-1)

def sample_x():
    return np.sqrt(1-rng.random()**2)

n_range = [2**i for i in range(6,15)]
estimates = np.array([estimate_expectation(sample_x,n) for n in n_range])
plt.errorbar(n_range,estimates[:,0],yerr=estimates[:,1],fmt='.')
plt.axhline(np.pi/4,c='r')
plt.xscale('log')
plt.legend([r"$\pi/4$","estimate"])
plt.xlabel("n")
plt.show()
```



3.2.3 One-pass algorithm

Note that here we generated all samples, stored them in a list, and then calculated the sample mean and variance. This can become inconvenient when dealing with a very large number of samples. Is there a way to calculate these quantities without storing the samples? The first approach that comes to mind is to keep track of the partial sums $\sum_{i=1}^{k} X_i$ and $\sum_{i=1}^{k} X_i^2$, since then at the end we could calculate

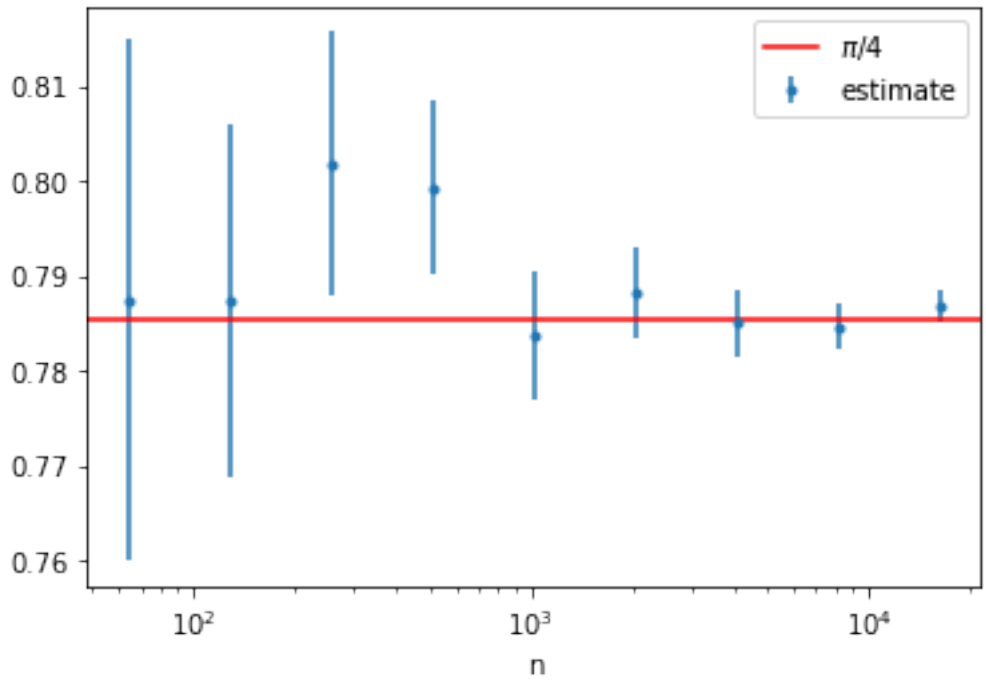
$$s_n^2 = \frac{1}{n-1}\sum_{i=1}^{n} X_i^2 - \frac{1}{n(n-1)}\left(\sum_{i=1}^{n} X_i\right)^2 .$$

Both terms however can be large and almost equal in magnitude, leading to large cancellations and depending on your programming language and machine precision to round-off errors dominating the answer. A **numerically stable one-pass algorithm** is obtained by keeping track of $\bar{X}_k = \frac{1}{k}\sum_{i=1}^{k} X_i$ and $S_k = \sum_{i=1}^{k}(X_i - \bar{X}_k)^2$, which satisfy

$$\bar{X}_k = \bar{X}_{k-1} + \frac{1}{k}\delta_k, \qquad S_k = S_{k-1} + \frac{k-1}{k}\delta_k^2, \qquad \delta_k := X_k - \bar{X}_{k-1}, \qquad \text{for } k \geq 1,$$

with the convention $\bar{X}_0 = S_0 = 0$. In particular, the following algorithm produces the same results (up to tiny rounding errors) as the previous method, without storing any samples.

```
def estimate_expectation_one_pass(sampler,n):
    sample_mean = sample_square_dev = 0.0
    for k in range(1,n+1):
        delta = sampler() - sample_mean
        sample_mean += delta / k
        sample_square_dev += (k-1)*delta*delta/k
    return sample_mean, np.sqrt(sample_square_dev / (n*(n-1)))

estimates = np.array([estimate_expectation_one_pass(sample_x,n) for n in n_range])
plt.errorbar(n_range,estimates[:,0],yerr=estimates[:,1],fmt='.')
plt.axhline(np.pi/4,c='r')
plt.xscale('log')
plt.legend([r"$\pi/4$","estimate"])
plt.xlabel("n")
plt.show()
```



3.2.4 Efficiency

To summarize, we have described an algorithm to approximate a definite integral which requires n steps to get an error of order $n^{-1/2}$. Should we be cheering? Not exactly. An error of $O(n^{-1/2})$ is actually pretty bad for a 1-dimensional integral. For instance, if the integrand has a continuous second-order derivative, then numerical Riemann integration using the midpoint method on n evenly spaced points yields an error of order $O(n^{-2})$. In situations where the integrand is not smooth, Monte Carlo integration can become competitive. But the most important point is that our derivation of the error $\sigma_X/\sqrt{n}$ in the Monte Carlo integration was **independent of the dimensionality** of the integral. Extending numerical Riemann integration to d-dimensional integrals requires d-dimensional grids of points to perform the appropriate interpolations, resulting in an error of order $O(n^{-2/d})$ in the number n of points. The great value of Monte Carlo thus lies in attacking high-dimensional integrals. The dependence on the dimension can be illustrated by calculating the d-dimensional generalization of the integral (3.1) (which computes 2^{-d} times the volume of $(d+1)$-dimension unit ball) using quadrature and Monte Carlo integration.

```
from scipy.integrate import nquad
def sphere_function(*args):
    '''Compute the function √(1-x^2) where the coordinates of the vector x are␣
 ↪provided as arguments.'''
```

(continues on next page)

(continued from previous page)

```
    norm = np.dot(args,args)
    return 0 if norm > 1 else np.sqrt(1-norm)

for d in range(1,5):
    print("In dimension {}:".format(d))
    result_quad = nquad(sphere_function,[[0.0,1.0] for _ in range(d)],opts={
↪'epsabs':[1e-10,1e-8,1e-4,1e-1][d-1]},full_output=True)
    print(" quadrature integration (n = {}): {} ± {}".format(result_quad[2]['neval
↪'],result_quad[0],result_quad[1]))
    num_mc = [2000,10000,100000,100000][d-1]
    result_mc = estimate_expectation_one_pass(lambda : sphere_function(*rng.
↪random(d)),num_mc)
    print(" Monte Carlo integration (n = {}): {} ± {}".format(num_mc,*result_mc))
```

```
In dimension 1:
 quadrature integration (n = 231): 0.7853981633974481 ± 8.833911380179416e-11
 Monte Carlo integration (n = 2000): 0.7829646471925696 ± 0.0050567317846304085
In dimension 2:
 quadrature integration (n = 13629): 0.5235987761827239 ± 1.1162598087397921e-08
 Monte Carlo integration (n = 10000): 0.5251332493181627 ± 0.0034392921532605376
In dimension 3:
 quadrature integration (n = 748587): 0.3084270213141045 ± 9.997878748733979e-05
 Monte Carlo integration (n = 100000): 0.30721660982150995 ± 0.
↪0010687255353164977
In dimension 4:
 quadrature integration (n = 745605): 0.1645870024457981 ± 0.09982235818747615
 Monte Carlo integration (n = 100000): 0.1652651648885213 ± 0.
↪0008717728127836131
```

Beyond 4 dimensions SciPy's quadrature starts struggling to provide any sensible answer, while the Monte Carlo integration is only starting to stretch its legs.

3.2.5 Variance reduction: importance sampling

It is all good and well that an error of $O(n^{-1/2})$ can be competitive compared to other approaches (especially in higher dimensions), but if the multiplicative factor σ_X in front is huge we are still in trouble. If $g(x)$ is close to zero in a large part of the domain, then intuitively we expect the Monte Carlo integration to be wasteful, because most of the samples will yield little contribution to the sample mean $\bar{X}_n$. To make this quantitative we can calculate the relative variance as

$$\sigma_X^2 = \int_{-\infty}^{\infty} (g(y) - \mathbb{E}[X])^2 f_Y(y) \mathrm{d}y.$$

It vanishes when $g(y)$ is constant (and thus equal to $\mathbb{E}[X]$) and can be very large if $g(y)$ is very localized. The situation can be improved by changing the distribution of the random variables used. In particular, if Z is a random variable with density $f_Z(y)$ that vanishes only when $f_Y(y)$ vanishes, then we can write

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} g(y) f_Y(y) \mathrm{d}y = \int_{-\infty}^{\infty} \frac{g(y) f_Y(y)}{f_Z(y)} f_Z(y) \mathrm{d}y = \mathbb{E}\left[\frac{g(Z) f_Y(Z)}{f_Z(Z)}\right].$$

What we have done is replacing the random variable Y by another random variable Z and the function $g(x)$ by $g(x) f_Y(x)/f_Z(x)$ without changing the expectation value, $\mathbb{E}[X] = \mathbb{E}[X']$ where $X' = \frac{g(Z) f_Y(Z)}{f_Z(Z)}$. The variance however has changed, because now

$$\sigma_{X'}^2 = \int_{-\infty}^{\infty} \left(\frac{g(z) f_Y(z)}{f_Z(z)} - \mathbb{E}[X]\right)^2 f_Z(z) \mathrm{d}z.$$

The strategy is then to choose Z such that $f_Z(z)$ is as close to being proportional to $g(z) f_Y(z)$ as possible.

As an example let us compute the probability $\mathbb{P}(Y > 3)$ for a normal random variable Y by setting $g(y) = \mathbf{1}_{\{y>3\}}$, i.e.

$$I = \int_{-\infty}^{\infty} \mathbf{1}_{\{y>3\}} f_Y(y) \mathrm{d}y = \int_4^{\infty} \frac{1}{2\pi} e^{-y^2/2} \mathrm{d}y.$$

```
def sample_x_naive():
    return 1 if rng.normal() > 3 else 0

estimate_expectation_one_pass(sample_x_naive,10000)
```

```
(0.0015999999999999988, 0.00039969985739001684)
```

The standard error is still quite large despite many samples. Instead, let us consider the random variable to be Z with PDF $f_Z(x) = e^{-(x-3)}\mathbf{1}_{\{x>3\}}$ and compute $\mathbb{E}[X] = \mathbb{E}[e^{Z-3} f_Y(Z)]$.

```
def random_exponential():
    return - np.log(rng.random())

def sample_x_faster():
    z = random_exponential() + 3
    return np.exp(z-3-z*z/2)/np.sqrt(2*np.pi)

estimate_expectation_one_pass(sample_x_faster,10000)
```

```
(0.001345952613432873, 1.356616409585813e-05)
```

This has reduced our error by a factor of about 30.

Note that it can happen that $\mathbb{E}[X]$ is finite but that one of σ_X^2 and $\sigma_{X'}^2$ is infinite. In last week's exercises we have seen that having infinite variance can lower the convergence rate of X_n to $\mathbb{E}[X]$ in an undesirable way. Importance sampling often provides a useful method to cure such infinite variances. For example, let us try to compute the integral

$$I = \int_0^1 x^{-\alpha} e^{-x} \mathrm{d}x$$

for $1/2 < \alpha < 1$. If we take $I = \mathbb{E}[X]$ with $X = U^{-\alpha} e^{-U}$ for a uniform random variable U then the variance $\sigma_X^2 = \infty$. Using importance sampling with $f_Z(x) = (1-\alpha)x^{-\alpha}\mathbf{1}_{\{0<x<1\}}$, we have

$$I = \frac{1}{1-\alpha} \mathbb{E}\left[e^{-Z}\right],$$

which is the expectation value of a bounded random variable $X' = e^{-Z}/(1-\alpha)$ and hence with finite variance.

```
def estimate_unbounded_integral(alpha,n):
    def sample_x_prime():
        exponent = 1/(1-alpha)
        z = rng.random()**exponent
        return exponent*np.exp(-z)

    return estimate_expectation_one_pass(sample_x_prime,n)

for alpha in np.arange(0.5,1,0.1):
    estimate,error = estimate_unbounded_integral(alpha,10000)
    print("alpha = {:.1f}, I = {:.5f} +- {:.5f}".format(alpha,estimate,error))
```

```
alpha = 0.5, I = 1.48985 +- 0.00402
alpha = 0.6, I = 1.95933 +- 0.00498
alpha = 0.7, I = 2.73756 +- 0.00645
alpha = 0.8, I = 4.35263 +- 0.00894
alpha = 0.9, I = 9.28230 +- 0.01452
```

3.3 Further reading

A thorough resource for this week's material is Chapter 2 of [Owe13].

Chapter 4

Markov Chain Monte Carlo (MCMC)

So far we have looked at direct-sampling Monte Carlo methods, i.e. various ways by which to draw independent random samples from a desired distribution. This works well for low-dimensional distributions and for problems in which the distribution takes a very simple or structured form. For many problems, however, these algorithms do not finish in reasonable time or simply fail. We have seen a cartoon problem of this sort in the first lecture, in which the kids were to deposit their pebbles uniformly on a square patch that extends far beyond their throwing range. The solution they devised was to not try to produce independent samples $X_1, X_2, ...$, but by sampling a Markov chain $X_1, X_2, ...$ in which X_{i+1} was dependent on X_i, the next pebble was thrown from the position of the former, in such a way that the distribution of X_i approaches the desired uniform distribution as $i \to \infty$. Of course, in this example the need for Markov Chain Monte Carlo (MCMC) was rather artificial. So before focusing on the mathematics, let us describe two statistical physics problems that really display this need.

4.1 The need for MCMC

4.1.1 Disks packed in a square

This is the statistical physics model that motivated Metropolis, Rosenbluth, Rosenbluth, Teller and Teller [MRR+53] in 1953 to propose the MCMC algorithm. Consider a collection of N disks of radius 1 packed into a square of side length L with periodic boundary conditions (hence, a torus). The model is that the disks are placed uniformly at random with the restriction that there are no overlaps. In mathematical terms the system is described by a random vector $\mathbf{X} = (x_1, y_1, \dots, x_N, y_N) \in [0, L)^{2N}$ of coordinates of the centers of the N disks with constant probability density function $f_{\mathbf{X}}(x_1, y_1, ...) = 1/Z$. Here the normalization Z, called the **partition function**, is the $2N$-dimensional volume of the set of coordinates without overlaps,

$$Z = \iint_{[0,L)^{2N}} \mathrm{d}x_1 \mathrm{d}y_1 \cdots \mathrm{d}x_N \mathrm{d}y_N \mathbf{1}_{\{\text{no overlaps}\}}.$$

If we have very few disks or a very large square then the model allows for a very simple direct-sampling Monte Carlo approach: simply place the disks uniformly in the square and reject the configuration in case of overlaps. What is the probability of a configuration being accepted? We can easily give an upper-bound: each of the last $N/2$ disks has a probability less than $1 - 2\pi N/L^2$ of being in a non-overlapping position, because the first $N/2$ disks already exclude an area of 4π each. Hence, the probability of success is less than $(1 - 2\pi N/L^2)^{N/2} < e^{-\pi N^2/L^2}$. In other words, the average computation time to sample a configuration with fixed packing fraction $\rho = \pi N/L^2$ grows at least as fast as $e^{N\rho}$ when the number N of disks increases! The solution Metropolis et al. came up with was to start with an acceptable configuration constructed by hand and then to perturb the positions of the individual disks via a Markov chain.

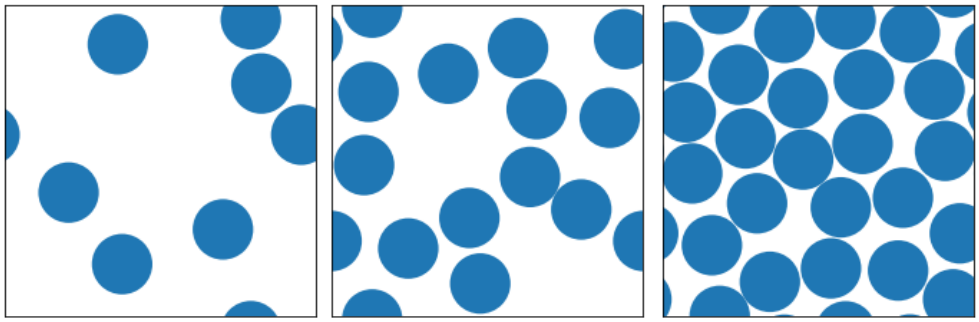


Fig. 4.1: Disks of radius 1 packed in a square with periodic boundary conditions.

4.1.2 Two-dimensional Ising model

Another famous problem showing the necessity of Markov Chain Monte Carlo simulations is the **ferromagnetic Ising model on a regular two-dimensional lattice** of size $w \times w = N$, again with periodic boundary conditions, at inverse temperature $\beta = 1/(k_\mathrm{B}T)$ and interaction energy $J > 0$. This time the sample space is discrete, described by a random configuration $\mathbf{s} \in \{-1, 1\}^N$ with probability mass function

$$p_\mathbf{s}(s) = \frac{1}{Z} e^{-\beta H(s)}, \qquad H(s) = -J \sum_{i \sim j} s_i s_j, \qquad s \equiv (s_1, \dots, s_N) \in \{-1, 1\}^N,$$

where in the **Ising Hamiltonian** $H(s)$ the sum runs over all pairs of nearest-neighbor sites i and j on the lattice. The normalization is the **Ising partition function**

$$Z = \sum_{s \in \{-1,1\}^N} e^{-\beta H(s)}.$$

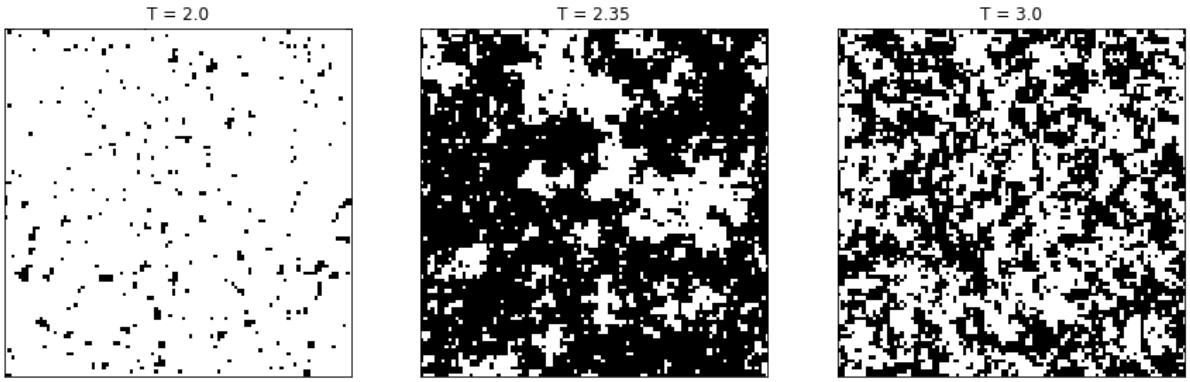


Fig. 4.2: Three configurations of the Ising model at different temperatures (where $\beta = J = 1$).

An important real random variable is the **(absolute) total magnetization** $|M(s)| = |\sum_{i=1}^N s_i|$ as its expectation value

$$\mathbb{E}[|M(\mathbf{s})|] = \sum_{s \in \{-1,1\}^N} |M(s)|\, p_\mathbf{s}(s)$$

shows an interesting dependence on the inverse temperature β (or the interaction J, since they only appear in the pair βJ). A brute-force evaluation of $\mathbb{E}[|M(\mathbf{s})|]$ involves summing over 2^N terms, infeasible for any interesting lattice. Direct sampling on the other hand would require either knowing a way to sample directly from $p_\mathbf{s}(s)$, which we don't, or sampling $\mathbf{s}'$ uniformly on $\{-1, 1\}^{2N}$ and measuring the expectation value $\mathbb{E}[|M(\mathbf{s})|] = 2^{2N}\mathbb{E}[|M(\mathbf{s}')|p_\mathbf{s}(\mathbf{s}')]$. However, for reasonably-sized lattices $p_\mathbf{s}(\mathbf{s}')$ varies astronomically from $e^{-2N\beta J}/Z$ up to $e^{2N\beta J}/Z$, meaning that the variance in the sample mean will be astronomical as well. The solution here again will be to construct a Markov

chain that only locally changes the Ising configuration in such a way that the distribution eventually converges to that of **s**.

Both examples have in common that we only really know the desired probability density or mass function up to a unknown normalization constant, which is the partition function Z of the model at hand. We should therefore aim our algorithms to only depend on ratios of probabilities, such that the dependence on the partition function drops out.

4.2 Markov chains

Let $(\Omega, \mathbb{P})$ be a probability space and consider a sequence of random variables $X_0, X_1, ... \in \Gamma$ on a **state space** Γ, i.e. X_i is a function $\Omega \to \Gamma$ for all $i \geq 1$. The sequence $X_1, X_2, ...$ is called a **Markov chain** if the random variables X_i satisfy the **Markov property**

$$\mathbb{P}(X_{i+1} \in A | X_j = x_j,\, 0 \leq j \leq i) = \mathbb{P}(X_{i+1} \in A | X_i = x_i)$$

for all $A \subset \Gamma$ and $i \geq 0$, meaning that the distribution of X_{i+1} only depends on the previous X_i and not on how the chain got to X_i. Informally, new samples in the state space are produced in a **memoryless** fashion, only depending on the current state. We will almost exclusively deal with **time-homogeneous** Markov chains which are the Markov chains that satisfy

$$\mathbb{P}(X_{i+1} \in A | X_i = x) = \mathbb{P}(X_1 \in A | X_0 = x),$$

meaning that at every step the transition probabilities are the same.

For now let us focus on the case of a finite state space $\Gamma = \{\omega_1, ..., \omega_M\}$ for some integer M. (An example would be the state space $\Gamma = \{-1, 1\}^N$ of the Ising model which has size $M = 2^N$.) The law of a time-homogeneous Markov chain is completely determined by specifying the probability mass function of the initial state X_0 and the **transition matrix**

$$P_{jk} \equiv P(\omega_j \to \omega_k) := \mathbb{P}(X_1 = \omega_k | X_0 = \omega_j).$$

Let's look at an example with $M = 6$ states and a transition matrix chosen by hand, making sure that the rows sum to 1.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

num_states = 6
transition_P = np.array([[0.2,0.4,0.4,  0,  0,  0],
                         [0.6,  0,  0,0.4,  0,  0],
                         [0.4,  0,0.4,  0,0.2,  0],
                         [  0,0.2,0.2,0.3,0.1,0.2],
                         [  0,  0,0.3,0.4,0.3,  0],
                         [  0,  0,  0,0.6,  0,0.4]])

print("Row sums =",np.dot(transition_P, np.ones(num_states)))
```

```
Row sums = [1. 1. 1. 1. 1. 1.]
```

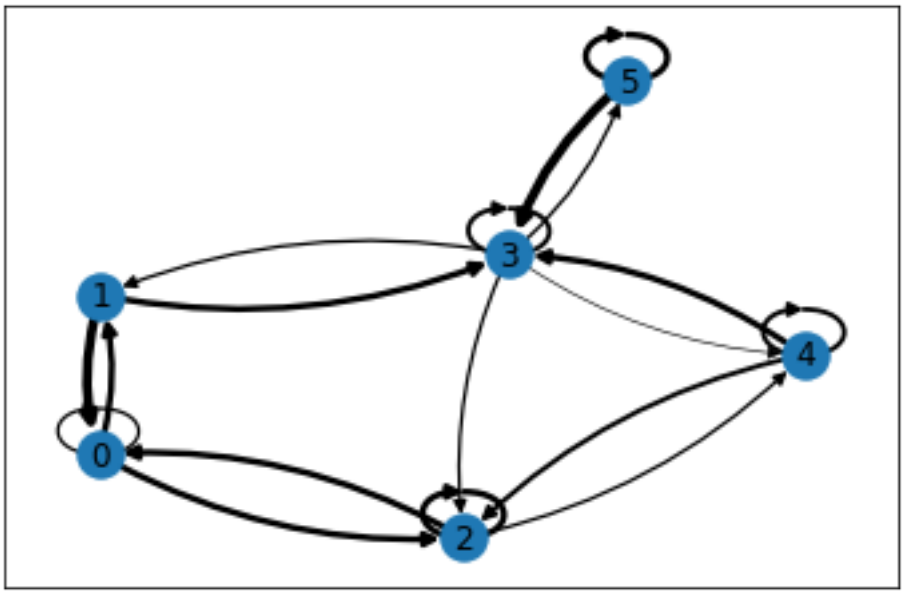
It is convenient to visualize the transition probabilities in a directed graph, which can be easily accomplished using the `networkX` package. Note that the widths of the arrows are proportional to the transition probabilities and that transitions from a node to itself may or may not be shown (depending on the version of networkX).

```
import networkx as nx
def draw_transition_graph(P):
    # construct a directed graph directly from the matrix
```

(continues on next page)

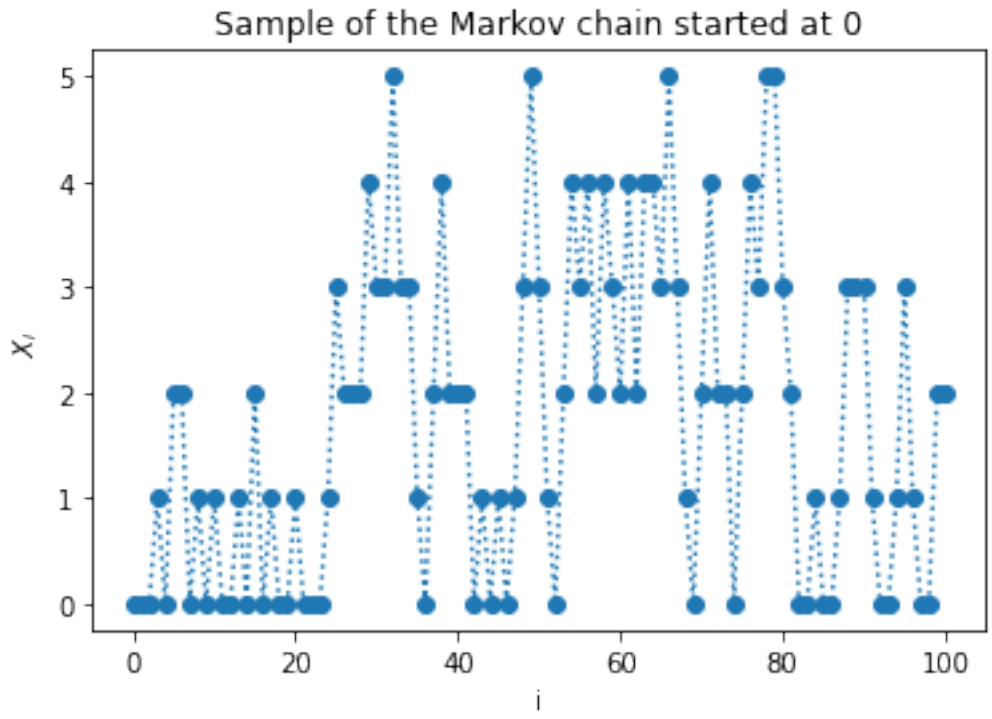
(continued from previous page)

```
    graph = nx.DiGraph(P)
    # draw it in such a way that edges in both directions are visible and have␣
↪appropriate width
    nx.draw_networkx(graph,connectionstyle='arc3, rad = 0.15',width=[6*P[u,v] for␣
↪u,v in graph.edges()])

draw_transition_graph(transition_P)
```



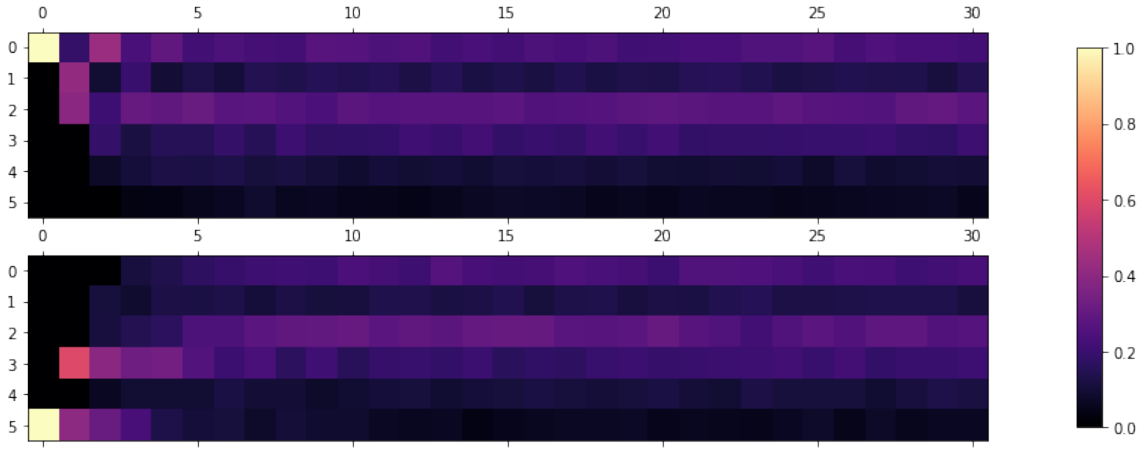
If we take the initial distribution to be deterministic, say $X_0 = 0$, we can sample the Markov chain as follows.

```
def sample_next(P,current):
    return rng.choice(len(P),p=P[current])

def sample_chain(P,start,n):
    chain = [start]
    for _ in range(n):
        chain.append(sample_next(P,chain[-1]))
    return chain

plt.plot(sample_chain(transition_P,0,100),linestyle=':', marker='o')
plt.xlabel("i")
plt.ylabel(r"$X_i$")
plt.title("Sample of the Markov chain started at 0")
plt.show()
```



By sampling the Markov chain many times we can numerically estimate the probability mass functions $p_{X_i}(\omega_j)$. The following plots illustrate this for $i = 0, \ldots, 30$ with two different initial conditions, $X_0 = 0$ respectively $X_0 = 5$.

```
fig, ax = plt.subplots(2,1, figsize=(12, 5))

num_samples = 500
length = 30
starting_points = [0,5]

for i in range(2):
    chains = [sample_chain(transition_P,starting_points[i],length) for _ in␣
↪range(num_samples)]
    distribution = np.apply_along_axis(lambda col: np.bincount(col,minlength=num_
↪states)/num_samples, 0, chains)
    im = ax[i].matshow(distribution,cmap=plt.cm.magma)

cbar_ax = fig.add_axes([0.95, 0.15, 0.02, 0.7])
fig.colorbar(im, cax=cbar_ax)
plt.show()
```



We observe that the distributions stabilize after ~10 steps and look remarkably similar for both initial conditions. Let us examine this mathematically.

Denoting the probability mass functions of X_i by the row-vector $\mathbf{p}_i = (p_{X_i}(\omega_1), \dots, p_{X_i}(\omega_M)) \in \mathbb{R}^M$, we find that

$$p_{X_{i+1}}(\omega_k) = \mathbb{P}(X_{i+1} = \omega_k) = \sum_{j=1}^{M} \mathbb{P}(X_i = \omega_j)\mathbb{P}(X_{i+1} = \omega_k | X_i = \omega_j) = \sum_{j=1}^{M} p_{X_i}(\omega_j) P_{jk} \quad \iff \quad \mathbf{p}_{i+1} = \mathbf{p}_i P.$$

More generally we thus have $\mathbf{p}_n = \mathbf{p}_0 P^n$ for $n \geq 0$. Note also that for some function $f : \Gamma \to \mathbb{R}$, we can compute the expectation value of the random variable $f(X_n)$ via

$$\mathbb{E}[f(X_n)] = \sum_{j=1}^{M} \mathbb{P}(X_n = \omega_j) f(\omega_j) = \mathbf{p}_0 P^n \mathbf{f}, \qquad \mathbf{f} = \begin{pmatrix} f(\omega_1) \\ \vdots \\ f(\omega_M) \end{pmatrix}.$$

What we see is that all these probabilistic concepts translate to linear algebra on matrices and vectors in $\mathbb{R}^M$.

4.2.1 Stationary distribution

A probability mass function $\pi : \Gamma \to [0, 1]$ is called a **stationary distribution** if

$$\pi(\omega_k) = \sum_{j=1}^{M} \pi(\omega_j) P_{jk},$$

or equivalently the row-vector $\pi = (\pi(\omega_1), \dots, \pi(\omega_M))$ is a right-eigenvector of P with eigenvalue 1, i.e. $\pi = \pi P$. We can make two observations:

- If $\mathbf{p}_n = \mathbf{p}_0 P^n$ converges as $n \to \infty$, the limit π must be a stationary distribution, since

$$\pi(\omega_k) = \lim_{n\to\infty} \mathbf{p}_n(\omega_k) = \lim_{n\to\infty} \sum_{j=1}^{M} \mathbf{p}_{n-1}(\omega_j) P_{jk} = \sum_{j=1}^{M} \pi(\omega_j) P_{jk}.$$

- P has an eigenvalue equal to 1, because $P\mathbf{1} = \mathbf{1}$ where $\mathbf{1} = (1, \dots, 1)^T$ is the vector with all ones, and all other eigenvalues have absolute value at most 1. Since P is a square matrix with non-negative entries, it then follows from the Perron-Frobenius Theorem that P admits at least one right-eigenvector of eigenvalue 1 and only non-negative entries. After normalization this gives at least one stationary distribution.

In the example above we have exactly one stationary distribution:

```
def stationary_distributions(P):
    eigenvalues, eigenvectors = np.linalg.eig(np.transpose(P))
    # make list of normalized eigenvectors for which the eigenvalue is very close
 ↪to 1
    return [eigenvectors[:,i]/np.sum(eigenvectors[:,i]) for i in
 ↪range(len(eigenvalues))
            if np.abs(eigenvalues[i]-1) < 1e-10]


print("Eigenvalues: ",np.linalg.eig(transition_P)[0])
for pi in stationary_distributions(transition_P):
    print("Stationary distribution: ",pi)
```

```
Eigenvalues:  [-0.55028838  1.          0.65425693 -0.08143647  0.42245768  0.
 ↪15501024]
Stationary distribution:  [0.23585973 0.13235294 0.27319005 0.19004525 0.
 ↪10520362 0.06334842]
```

But this is not true in general as can be seen in the following simple example.

```
disconnect_P = np.array([[0.5,0.5,0,0],[0.5,0.5,0,0],[0,0,0.5,0.5],[0,0,0.5,0.5]])
draw_transition_graph(disconnect_P)
print("Eigenvalues: ",np.linalg.eig(disconnect_P)[0])
for pi in stationary_distributions(disconnect_P):
    print("Stationary distribution: ",pi)
```

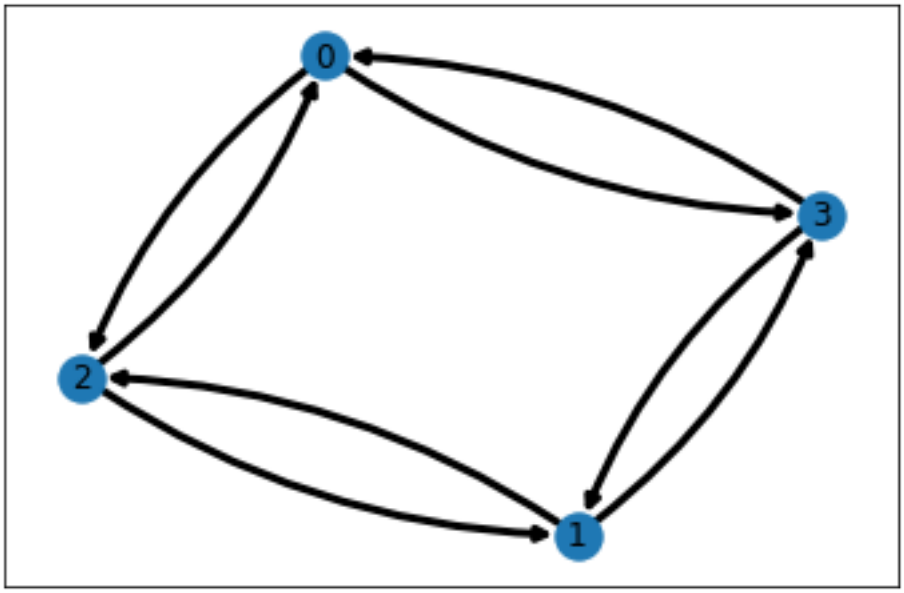
```
Eigenvalues:  [1.00000000e+00 1.11022302e-16 1.00000000e+00 1.11022302e-16]
Stationary distribution:  [0.5 0.5 0.  0. ]
Stationary distribution:  [0.  0.  0.5 0.5]
```

We see that there are two linearly independent stationary distributions, due to the fact that the state space consists of two components that have no transitions among each other. To prevent this phenomenon from happening it is useful to consider the property of irreducibility. The transition matrix P is **irreducible** if for every pair of states $x, y \in \Gamma$ there is a positive chance for the Markov chain started at x to eventually reach y. In the pictorial representation as a directed graph this property is equivalent to the graph being **strongly connected**, meaning that every vertex is reachable along an oriented path from every other vertex. It can be shown that an irreducible transition matrix P always possesses a unique invariant distribution.

Does this imply that every irreducible Markov chain approaches the invariant distribution? Almost, but not quite! There is one more thing that can go wrong: the Markov chain can be **periodic** with period $t = 2, 3, ...$, meaning that there exists a state $x \in \Gamma$ such that the Markov chain can only return to x after a number a transitions that is a multiple of t. It is not difficult to convince yourself that if the Markov chain is irreducible and this holds for one state x, then it holds for every state in Γ. An example of a periodic Markov chain with period 2 is the following.

```
disconnect_P = np.array([[0,0,0.5,0.5],[0,0,0.5,0.5],[0.5,0.5,0,0],[0.5,0.5,0,0]])
draw_transition_graph(disconnect_P)
print("Eigenvalues: ",np.linalg.eig(np.transpose(disconnect_P))[0])
```

```
Eigenvalues:  [-1.00000000e+00  1.00000000e+00 -8.80288308e-17 -2.61425503e-48]
```



In this case the Markov chain alternates between the states $\{0, 1\}$ and $\{2, 3\}$, so if the Markov chain start at a deterministic state, say $X_0 = 0$, the distribution of X_n cannot converge as $n \to \infty$, because $\mathbb{P}(X_{2n} \in \{0, 1\}) = 1$ while $\mathbb{P}(X_{2n+1} \in \{0, 1\}) = 0$ for all n. The periodicity is reflected in the eigenvalues: there are exactly t eigenvalues on the unit circle in the complex plane. If the Markov chain is not periodic, it is called **aperiodic**. Note that in practice it is not a particularly strong assumption on an irreducible Markov chain: $P_{ii} > 0$ for some state ω_i already guarantees the Markov chain is aperiodic.

If P is irreducible and aperiodic, then P has a unique right-eigenvector of eigenvalue 1, corresponding to its unique stationary distribution, while all other eigenvalues λ have absolute value $|\lambda| < 1$. If we assume that P is diagonalizable (which is true for all the transition matrices satisfying detailed balance, see next section), we can easily prove convergence, but the following result can be shown to hold in general. If P is diagonalizable, we can expand the initial distribution $\mathbf{p}_0 = \sum_i c_i v_i$ in terms of the right-eigenvectors v_i with eigenvalues λ_i, where $\lambda_1 = 1$ and $v_1 = \pi$ is the stationary distribution. Then it is clear that

$$\lim_{n\to\infty} \mathbf{p}_o P^n = \lim_{n\to\infty} \sum_i c_i \lambda_i^n v_i = c_1 \pi.$$

We necessarily have $c_1 \pi \mathbf{1} = c_1 = 1$ because $\mathbf{p}_o P^n \mathbf{1} = \mathbf{p}_o \mathbf{1} = 1$. To summarize, we have established the following important theorem.

Theorem: convergence to stationary distribution

If the transition matrix P on a finite state space is irreducible and aperiodic, then it has a unique stationary distribution π and $\lim_{n\to\infty} \mathbb{P}(X_n = x) = \pi(x)$ irrespective of the distribution of X_0.

If X is a random variable on Γ with probability density function π, then this statement is equivalent to X_n converging in distribution to X as $n \to \infty$, which we denote as $X_n \xrightarrow[n\to\infty]{(d)} X$. What can we say about the **sample mean** $\overline{f(X)}_n := \frac{1}{n} \sum_{i=1}^n f(X_i)$? In the case of i.i.d. sample $X_1, X_2, \ldots$ the law of large numbers told us that it converges in probability to $\mathbb{E}[f(X)]$, but now the samples are very much correlated with each other. Nevertheless it is possible to prove the following analogue, known as the ergodic theorem.

Theorem: ergodic theorem

If the transition matrix P on a finite state space is irreducible, then for any function $f : \Gamma \to \mathbb{R}$ the sample mean $\overline{f(X)}_n := \frac{1}{n} \sum_{i=1}^n f(X_i)$ converges to $\mathbb{E}[f(X)]$ with probability 1.

The way the statement is phrased is as an **almost sure convergence** ("almost sure" means with probability 1), which implies convergence in probability but is slightly stronger. Note that the transition matrix in this case is not required

to be aperiodic, because the sample mean properly averages over any periods P may have. Here is an example for our previous transition matrix with function $f(\omega_i) = i$.

```
def markov_sample_mean(P,start,function,n):
    total = 0
    state = start
    for _ in range(n):
        state = sample_next(P,state)
        total += function[state]
    return total/n

# an example of a function specified as a vector
function = [i+1 for i in range(num_states)]
exact_expectation = np.dot(stationary_distributions(transition_P)[0],function)
print("E[f(X)] =",exact_expectation)
for n in [2**i for i in range(8,16)]:
    print("sample mean for n =",n,":",markov_sample_mean(transition_P,0,function,
 ↪n))
```

```
E[f(X)] = 2.9864253393665163
sample mean for n = 256 : 2.84375
sample mean for n = 512 : 2.990234375
sample mean for n = 1024 : 3.0673828125
sample mean for n = 2048 : 2.96337890625
sample mean for n = 4096 : 3.037841796875
sample mean for n = 8192 : 3.0185546875
sample mean for n = 16384 : 2.9957275390625
sample mean for n = 32768 : 2.972900390625
```

4.3 Markov Chain Monte Carlo

It should now be clear what the general strategy of the MCMC technique is: given a desired distribution π on the sample space Γ, find a (preferably aperiodic) irreducible transition matrix P that has π as stationary distribution, and apply the ergodic theorem to estimate expectation values. Such a transition matrix is far from unique: if the sample space has M states, the space of transition matrices is $M(M-1)$-dimensional, while there is only an $(M-1)$-dimensional space of probability distributions on Γ, meaning that there are many transition matrices that share the same stationary distribution. So we need some guiding principle in constructing transition matrices. A particularly convenient principle is that of detailed balance.

4.3.1 Detailed balance

Note that the stationarity condition $\pi P = \pi$ is equivalent to requiring the following identity for all $y \in \Gamma$:

$$\sum_{x\in\Gamma} \pi(x)P(x \to y) = \pi(y) = \pi(y)\sum_{x\in\Gamma} P(y \to x) = \sum_{x\in\Gamma} \pi(y)P(y \to x).$$

This can be interpreted as a balance condition. The left-hand side gives the total flow of probability from any state x into y, while the right-hand side gives the flow out of y to any state x. In equilibrium, i.e. in the stationary distribution, these two flows must be equal.

The simplest way to ensure this balance condition is to require **detailed balance**,

$$\pi(x)P(x \to y) = \pi(y)P(y \to x),$$

meaning that not just the total flow in and out of state y is balanced, but that the flow between any two states x and y is the same in both directions.

4.3.2 Metropolis-Hastings algorithm

There is a general method to construct a transition matrix $P(x \to y)$ satisfying detailed balance for a desired distribution π, developed by Metropolis et al in the 50s and generalized by Hastings in the 70s. It takes as input an irreducible transition matrix $Q(x \to y)$ on the state space, whose stationary distribution may be quite different from π, and derives from it a new irreducible and aperiodic transition matrix $P(x \to y)$ with the desired stationary distribution. The way it does this is very much analogous to the acceptance-rejection sampling of real random variables, that we discussed last week. If the state of the Markov chain after i steps is X_i, then the next state is chosen as follows. A random state Y is sampled according to the transition matrix $Q(x \to y)$, meaning with **proposal probability** $\mathbb{P}(Y = y|X_i = x) = Q(x \to y)$. With a to-be-determined **acceptance probability** $A(x \to y) \in [0, 1]$ this state is accepted ($X_{i+1} = Y$) and otherwise rejected ($X_{i+1} = X_i$), i.e. we set

$$X_{i+1} = \begin{cases} Y & \text{with probability } A(X_i \to Y) \\ X_i & \text{otherwise} \end{cases}.$$

The resulting transition matrix $P(x \to y) = \mathbb{P}(X_{i+1} = y|X_i = x)$ satisfies

$$P(x \to y) = Q(x \to y)A(x \to y) \qquad \text{for } x \neq y.$$

It satisfies detailed balance precisely when

$$\pi(x)Q(x \to y)A(x \to y) = \pi(y)Q(y \to x)A(y \to x) \qquad \text{for all } x, y \in \Gamma.$$

We only need to consider the case $\pi(x)Q(x \to y) > 0$, because if it were zero the transition $x \to y$ would never be proposed anyway. What are the maximal values we can choose for the acceptance probabilities $A(x \to y)$ and $A(y \to x)$? Let us write $A(y \to x) = s\,\pi(x)Q(x \to y)$ for some $s \geq 0$. Detailed balance then implies $A(x \to y) = s\,\pi(y)Q(y \to x)$. Since both probabilities have to be smaller or equal to 1, the optimal choice is

$$s = \frac{1}{\max\left(\pi(y)Q(y \to x),\, \pi(x)Q(x \to y)\right)}.$$

This results in the **Metropolis-Hastings acceptance probability**

$$A(x \to y) = \min\left(1, \frac{\pi(y)Q(y \to x)}{\pi(x)Q(x \to y)}\right).$$

Note that if Q is irreducible and $Q(y \to x) > 0$ whenever $Q(x \to y) > 0$, then P is automatically irreducible as well because with non-zero probability a sequence of transitions proposed by Q is fully accepted. Note also the important property that the acceptance probability depends only on the ratio $\pi(y)/\pi(x)$, so it fulfills the wish we had that the algorithm **does not rely on knowing the normalization** of the desired probability distribution!

4.3.3 Application to Ising model

Let's see how this can be applied to the two-dimensional Ising model described above. The state space is $\Gamma = \{-1, 1\}^N$ and the desired distribution is $\pi(s) = e^{-\beta H(s)}/Z$ for $s \in \Gamma$. Probably the simplest irreducible transition matrix $Q(x \to y)$ we can think of is the one that uniformly selects a site i on the lattice and flips the spin, i.e.

$$Q(x \to y) = \frac{1}{N}\mathbf{1}_{\{x \text{ and } y \text{ differ by exactly one spin}\}}.$$

It is irreducible because with enough single spin flips we go from any state to any other state. Note also that by its definition $Q(x \to y) = Q(y \to x)$. The Metropolis-Hastings algorithm prescribes that we accept such a spin flip from state x to y with probability

$$A(x \to y) = \min\left(1, \frac{\pi(y)Q(y \to x)}{\pi(x)Q(x \to y)}\right) = \min\left(1, e^{\beta H(x) - \beta H(y)}\right) = \begin{cases} 1 & \text{if } H(y) \leq H(x) \\ e^{-\beta(H(y) - H(x))} & \text{if } H(y) > H(x) \end{cases}.$$

So if the spin flip lowers the energy it is always accepted, while if it increases the energy it is only accepted with a probability that decreases exponentially with increasing energy difference. The energy difference is easily calculated to be

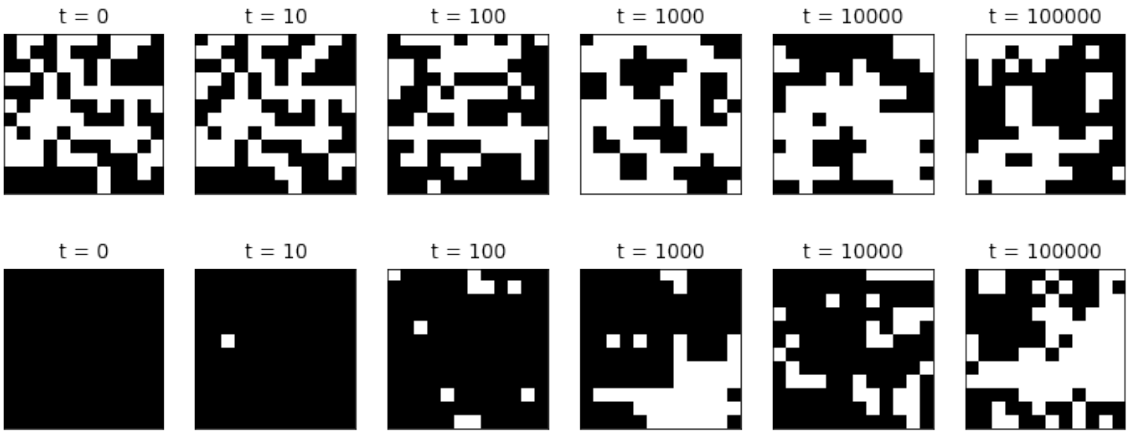
$$H(y) - H(x) = 2Js_i \sum_{j:i\sim j} s_j,$$

where the sum is over the 4 neighbors j of i. A single transition of the Markov chain can therefore be simulated with the following code, where `config` is a square array with values ± 1 representing the current state and `boltzmannfactor` is $e^{-2\beta J}$.

```
def attempt_spin_flip(config,boltzmannfactor):
    w = len(config)
    i,j = rng.integers(0,w,2)
    neighbour_sum = config[i,j] * (config[(i+1)%w,j] + config[(i-1)%w,j] +
                                   config[i,(j+1)%w] + config[i,(j-1)%w])
    if neighbour_sum <= 0 or rng.random() < boltzmannfactor**neighbour_sum:
        config[i,j] = -config[i,j]
```

As an example the following code illustrates the Markov chain of the Ising spins on a 12×12 grid with $\beta J = 0.3$ and two different initial conditions: a completely random configuration and a completely aligned configuration.

```
def plot_config(config,ax,title):
    ax.matshow(config, vmin=-1, vmax=1, cmap=plt.cm.binary)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])

def illustrate_evolution(config,plot_times):
    fig, ax = plt.subplots(1,len(plot_times),figsize=(12,10))
    for t in range(plot_times[-1]+1):
        if t in plot_times:
            plot_config(config,ax[plot_times.index(t)],"t = {}".format(t))
        attempt_spin_flip(config,boltzmannfactor)
    plt.show()

betaJ = 0.3
width = 12
boltzmannfactor = np.exp(-2*betaJ)
plot_times = [0,10,100,1000,10000,100000]

config = 2*rng.integers(2,size=(width,width))-1
illustrate_evolution(config,plot_times)

config = np.ones((width,width),dtype=int)
illustrate_evolution(config,plot_times)
```



4.3.4 Continuous state space

When the state space Γ is not finite but a compact subset of $\mathbb{R}^m$, most of the results above adapt naturally. A continuous (time-homogeneous) Markov chain does not have a transition matrix but a **transition density** $p(x, y)$ which gives the probability density in y of the state $X_i = x \in \mathbb{R}^m$ jumping to $X_{i+1} = y$. It satisfies

$$\int_\Gamma \mathrm{d}^m y \, p(x, y) = 1.$$

If X_0 is a continuous random variable with density $f_{X_0}(x)$, then

$$f_{X_1}(y) = \int_\Gamma \mathrm{d}^m x f_{X_0}(x) p(x, y), \qquad f_{X_2}(y) = \int_\Gamma \mathrm{d}^m x f_{X_1}(x) p(x, y), \qquad \text{etc.}$$

A **stationary distribution** for this Markov chain is a probability density function $\pi(x)$ satisfying

$$\pi(y) = \int_\Gamma \mathrm{d}^m x \pi(x) p(x, y).$$

If the Markov chain possesses a stationary distribution, then under mild conditions on $p(x, y)$ it is unique: it should satisfy a continuous analogue of irreducibility and **recurrence**, which essentially says that no matter the starting point it should visit any subset of Γ infinitely often with probability 1. In this case, the Markov chain will converge to its unique stationary distribution. We may thus again design a Markov chain to approach a desired distribution by requiring **detailed balance**

$$\pi(x)p(x, y) = \pi(y)p(y, x) \qquad \text{for } x, y \in \Gamma.$$

The Metropolis-Hastings algorithm works completely analogously, but this time with **proposal transition density** $q(x, y)$. The proposed next state y is accepted with probability

$$A(x \to y) = \min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right).$$

4.3.5 Application to the disk model

The state space of N disks of radius 1 in the torus $[0, L)^2$ is given by the N positions $\Gamma = [0, L)^{2N}$ and the desired distribution is the density $\pi(x) = \mathbf{1}_{\{\text{all pairwise distance } \geq 2\}}(x)/Z$. A simple proposal density is to select a uniform disk and move it by a reflection-symmetric two-dimensional distribution. The symmetry implies that $q(y, x) = q(x, y)$, so the acceptance probability of a proposed move $x \to y$ simply becomes

$$A(x \to y) = \min\left(1, \frac{\pi(y)q(y, x)}{\pi(x)q(x, y)}\right) = \mathbf{1}_{\{\text{all pairwise distance } \geq 2\}}(y).$$

In other words we should simply accept the proposed configuration whenever it is valid and reject otherwise. Implementing this model is one of this week's exercises.

4.4 Further reading

A detailed account of MCMC methods can be found in Chapters 11 of [Owe13].

For a more mathematical but still accessible treatment of Markov chains including self-contained proofs of the quoted results, you could have a look at Chapter 1 of [Nor97].

Chapter 5

MCMC in practice

Last week we have seen that Markov Chain Monte Carlo simulations work in theory. We can summarize our findings as follows. Given a discrete or continuous **state space** Γ, a **desired probability distribution** π on Γ, and an **observable** $f : \Gamma \to \mathbb{R}$, the problem is to compute the expectation value of f evaluated on a random variable X on Γ with probability mass function or probability density functions given by π,

$$\mathbb{E}[f(X)] = \int_\Gamma \mathrm{d}x\, \pi(x) f(x) \quad \text{(continuous)}, \qquad \mathbb{E}[f(X)] = \sum_{x\in\Gamma} \pi(x) f(x) \quad \text{(discrete)}.$$

In the discrete case this problem can be addressed by designing a Markov chain **transition matrix** $P(x \to y)$ that is irreducible and has π as its **stationary distribution**, i.e. $\pi P = \pi$. One then considers the Markov chain $X_0, X_1, X_2, ...$ with arbitrary distribution for X_0 and transition probability dictated by $P(x \to y)$. The **ergodic theorem** then assures us that we can approximate $\mathbb{E}[f(X)]$ via the **time average**

$$\frac{1}{n}\sum_{i=1}^{n} f(X_i) \xrightarrow[n\to\infty]{\text{almost surely}} \mathbb{E}[f(X)].$$

The same holds in the continuous case for an irreducible transition density, under a technical additional assumption of **Harris recurrence**.

We have also seen how to obtain a suitable transition matrix $P(x \to y)$ from a proposal transition matrix $Q(x \to y)$ using the **Metropolis-Hastings algorithm**. Thus with infinite computer power, our problem of computing $\mathbb{E}[f(X)]$ is solved. In practice, however, we can only sample the time average for a finite number n of states in the Markov chain, so we require quantitative control over the error $\frac{1}{n}\sum_{i=1}^{n} f(X_i) - \mathbb{E}[f(X)]$. In the case of **i.i.d.** random variables this error was easily related to the (estimated) variance of $f(X)$, but this becomes significantly more difficult now that the states X_i are correlated.

Warning: There is no universally applicable recipe for performing Markov Chain Monte Carlo simulations and their error analysis! Every problem comes with its own challenges and it is our job to figure our which method works best in those circumstances.

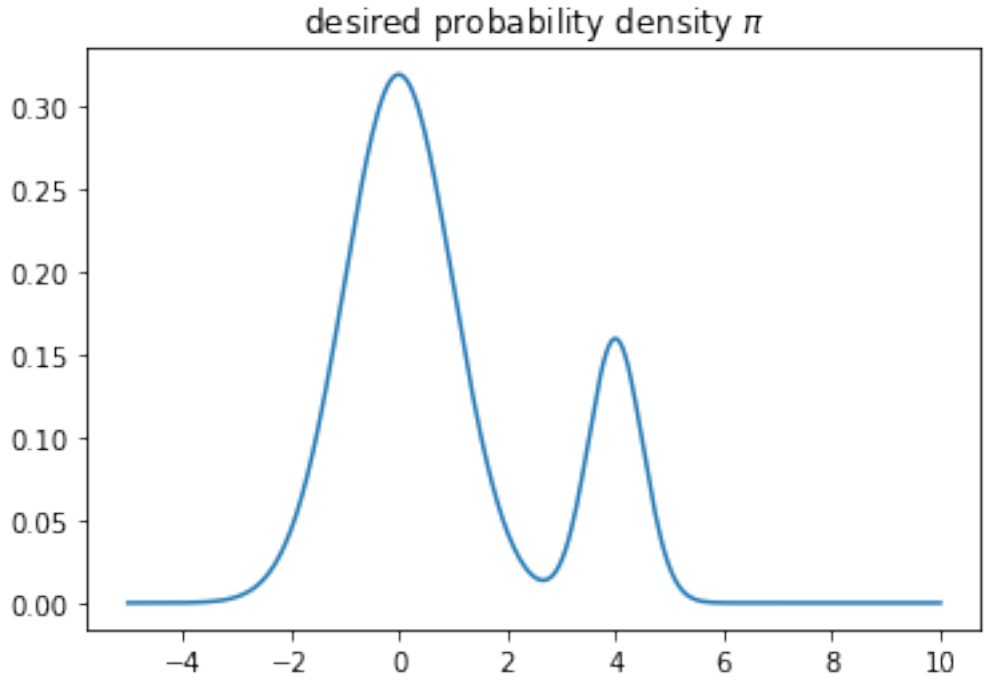
5.1 A toy model: MCMC on the real line

To see where the difficulties in estimating the error come from, let us have a look at a rather artificial toy problem in which the state space is $\mathbb{R}$ and the desired distribution of the random variable X displays two peaks,

$$\pi(x) = \frac{4}{5}\frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} + \frac{1}{5}\frac{2}{\sqrt{2\pi}}e^{-2(x-4)^2}.$$

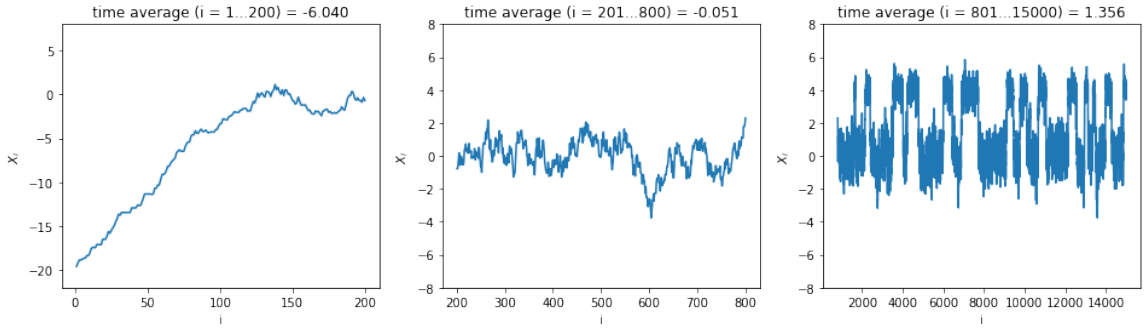
Suppose we wish to estimate the expectation value of the observable $f(x) = x$, which is $\mathbb{E}[f(X)] = \mathbb{E}[X] = 4/5$, using a MCMC.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def gaussian(mu,sigma,x):
    return np.exp(-(x-mu)*(x-mu)/(2*sigma*sigma))/(sigma*np.sqrt(2*np.pi))

def distribution_pi(x):
    return 0.8*gaussian(0,1,x) + 0.2*gaussian(4,0.5,x)

xrange = np.linspace(-5,10,300)
plt.plot(xrange,distribution_pi(xrange))
plt.title(r"desired probability density $\pi$")
plt.show()
```



To design a Markov chain $X_0, X_1, \ldots$ with the desired stationary distribution, we apply the Metropolis-Hastings acceptance to the proposal $X_i + U$ for the next state X_{i+1}, where U is a uniform random variable in $(-\delta, \delta)$ with $\delta = 0.7$. Since U has a symmetric distribution, the proposal density $q(x, y)$ is symmetric as well, and therefore the appropriate acceptance probability is $A(x \to y) = \min(1, \pi(y)/\pi(x))$ with $x = X_i$ and $y = X_i + U$. As initial state we choose a rather atypical value $X_0 = -20$. The following code samples the first 15000 steps of the Markov chain and displays plots of X_n (sometimes called **traces**) for three different ranges of the time n, together with the estimates of $\mathbb{E}[f(X)]$ stemming from the time average in the corresponding range.

```
def MH_step(current,delta,pi):
    proposal = current + rng.uniform(-delta,delta)
    return proposal if rng.uniform() < pi(proposal)/pi(current) else current

def sample_chain(start,delta,pi,n):
    chain = np.zeros(n+1)
    chain[0] = start
    for i in range(n):
        chain[i+1] = MH_step(chain[i],delta,pi)
    return chain

delta = 0.7
start_x0 = -20
chain = sample_chain(start_x0,delta,distribution_pi,15000)
ranges = [range(1,201),range(201,801),range(801,15001)]
fig, ax = plt.subplots(1,len(ranges),figsize=(16,4))
for i in range(len(ranges)):
    ax[i].plot(ranges[i],chain[ranges[i]])
    ax[i].set_ylim(-22 if i==0 else -8,8)
    time_average = np.mean(chain[ranges[i]])
    title = "time average (i = {}...{}) = {:.3f}".format(ranges[i][0],ranges[i][-
 ↪1],time_average)
    ax[i].title.set_text(title)
    ax[i].set_xlabel("i")
    ax[i].set_ylabel("$X_i$")
```



The first two estimates are (most likely) not particularly close to the exact value $\mathbb{E}[X] = 0.8$, but for two different reasons.

5.1.1 Equilibration

It is clear that the time average $\frac{1}{n}\sum_{i=1}^{n} X_i$ for $n = 200$ is very much biased by the atypical starting point $X_0 = -20$. This may seem artificial, since we chose deliberately to start in an atypical point, but in most applications of MCMC the situation is as bad or worse: any starting point X_0 we are able construct by hand in a high-dimensional sample space is almost guaranteed to be very atypical, in the sense that there are observables $f : \Gamma \to \mathbb{R}$ such that $f(X_0)$ is many sigmas away from $\mathbb{E}[f(X)]$. What we are seeing is that the probability density $f_{X_i}(x)$ for $i \leq 200$, is not yet close to the desired distribution $\pi(x)$. The time τ_{eq} it takes to approach $\pi(x)$ is called the **equilibration time**, which appears to be at least $\tau_{\text{eq}} \geq 200$ in this case. Of course, the contribution of the first 200 terms to the time average $\frac{1}{n}\sum_{i=1}^{n} f(X_i)$ will become negligible when $n \to \infty$, but can very much affect the estimate when n is limited, because the atypical configurations X_i may contribute excessive values $f(X_i)$ to the sum.

A common practice is therefore to discard the first b configurations in the time average, where b is chosen to ensure $b \geq \tau_{\text{eq}}$ to the best of our knowledge. This means that we take

$$\frac{1}{n-b}\sum_{i=b+1}^{n} f(X_i)$$

as our best estimate of the $\mathbb{E}[f(X)]$. The first b steps in a MCMC simulation, in which we discard the observable $f(X_i)$ (and sometimes not measure $f(X_i)$ at all), is often called the **equilibration**, **thermalization**, or **burn-in** phase of the simulation. In effect, what equilibration exploits is that, although the limit of the time average is independent of the distribution of X_0, the convergence will be faster if the distribution of X_0 is closer to π. This is achieved by taking the random state X_b as a new effective starting point X_0 of the chain (which by the convergence theorem is close to π for b large). We will see a technique to gauge τ_{eq} below in the case of the Ising model.

5.1.2 Autocorrelation

Suppose we know for our example that at time $i \geq b = 200$ the distribution of X_i is already sufficiently close to $\pi(x)$. Why is our estimate based on the time average $i = 201, \ldots, 800$ still not great, even though we have a decent 600 data points? The reason should be intuitively clear: the states X_i are far from independent, meaning that each new state only gives us a tiny bit of extra information on the distribution π. There is dependence in a trivial sense because our Markov chain is setup such that $|X_{i+1} - X_i| < \delta$, so the chain cannot wander far in a few steps. More serious is that the distribution π has two peaks, in the vicinity of which the Markov chain likes to stay, separated by a barrier where $\pi(x)$ is very small. This means that it takes a sequence of probabilistically unlikely steps for the chain to transition between the two regions, so it happens only rarely. The dependence in this example therefore persists over a large number of steps, e.g. you can imagine from looking at the plot that when X_i is in the vicinity of the first peak the probability that X_{i+200} is also in this vicinity is larger than if X_i was in the vicinity of the second peak, i.e. $\mathbb{P}(X_{i+200} \in (3,5)|X_i = 4)$ is significantly larger than $\mathbb{P}(X_{i+200} \in (3,5)|X_i = 0)$. For a good approximation of the desired distribution π we need many transitions between the two regions to have occurred, showing the necessity of performing many steps in the Markov chain (the last plot shows ± 20 transitions in the first 15000 steps).

Let us try to get a quantitative grasp on the error. To this end we assume that $\text{Var}(f(X)) < \infty$ and that X_0 has exactly the stationary distribution $\pi(x)$ (perhaps obtained by performing equilibration for a long time b and then resetting the label i of the state X_i to 0). By construction of the stationary distribution, the same holds for all X_i with $i \geq 0$. In this case $\overline{f(X)}_n = \frac{1}{n}\sum_{i=1}^n f(X_i)$ is an unbiased estimator for $\mathbb{E}[f(X)]$ since

$$\mathbb{E}\left[\overline{f(X)}_n\right] = \frac{1}{n}\sum_{i=1}^n \mathbb{E}[f(X)] = \mathbb{E}[f(X)].$$

For the variance we have

$$\begin{aligned}
\text{Var}\left(\overline{f(X)}_n\right) &= \mathbb{E}\left[\left(\overline{f(X)}_n - \mathbb{E}[f(X)]\right)^2\right] \\
&= \frac{1}{n^2}\sum_{i=1}^n\sum_{j=1}^n \mathbb{E}\Big[(f(X_i) - \mathbb{E}[f(X)])(f(X_j) - \mathbb{E}[f(X)])\Big] \\
&= \frac{1}{n^2}\sum_{i=1}^n \Bigg(\mathbb{E}\Big[(f(X_i) - \mathbb{E}[f(X)])^2\Big] \\
&\qquad + 2\sum_{t=1}^{n-i} \mathbb{E}\Big[(f(X_i) - \mathbb{E}[f(X)])(f(X_{i+t}) - \mathbb{E}[f(X)])\Big]\Bigg) \\
&= \frac{\text{Var}(f(X))}{n}\left(1 + 2\sum_{t=1}^{n-1}\frac{n-t}{n}\rho(t)\right),
\end{aligned} \tag{5.1}$$

where $\rho(t)$ is the **autocorrelation**

$$\rho(t) = \text{Corr}(f(X_i), f(X_{i+t})) := \frac{\mathbb{E}\Big[(f(X_i) - \mathbb{E}[f(X)])(f(X_{i+t}) - \mathbb{E}[f(X)])\Big]}{\text{Var}(f(X))}.$$

We can estimate the latter via the **sample autocovariance**,

$$\bar{\gamma}(t) = \frac{1}{n-t}\sum_{i=1}^{n-t}(f(X_i) - \overline{f(X)}_n)(f(X_{i+t}) - \overline{f(X)}_n), \tag{5.2}$$

from which one obtains the **sample autocorrelation** as

$$\bar{\rho}(t) = \frac{\bar{\gamma}(t)}{\bar{\gamma}(0)}$$

by normalizing $\bar{\gamma}(t)$ by the sample variance $\bar{\gamma}(0)$. For completeness we should mention that variations of the formula (5.2) are used in the literature, for instance one in which the factor $1/(n-t)$ is replaced by $1/n$. In each case the result agrees in the limit of large n when t is kept fixed, and differences only become noticeable when t is of the same order as n. The formula (5.2) is sometimes called the *unbiased* sample autocorrelation, but this is a bit of a misnomer because it is only unbiased if we replace $\overline{f(X)}_n$ by the known mean $\mathbb{E}[f(X)]$. The formula with $1/n$ is often found to be more stable numerically when t approaches n, so keep that in mind.

One may be tempted to take the sample autocorrelation $\bar{\rho}(t)$ and plug it in (5.1), to get an error estimate. However, in practice this rarely works, because the statistical errors in $\bar{\rho}(t)$ grow large when t becomes of the order of n. It is often more convenient to estimate an **autocorrelation time** τ_f under the assumption that $\bar{\rho}(t)$ decays exponentially, i.e. $\bar{\rho}(t) \approx e^{-t/\tau_f}$. This can be done either by fitting an exponential curve or, slightly less accurately, by taking τ_f to be the first time t that $\bar{\rho}(t)$ drops below $1/e$,

$$\tau_f \approx \inf\{t : \bar{\rho}(t) < 1/e\}.$$

If $1 \ll \tau_f \ll n$, then

$$\left(1 + 2\sum_{t=1}^{n-1} \frac{n-t}{n}\bar{\rho}(t)\right) \approx 2\sum_{t=1}^{\infty} e^{-t/\tau_f} \approx 2\tau_f.$$

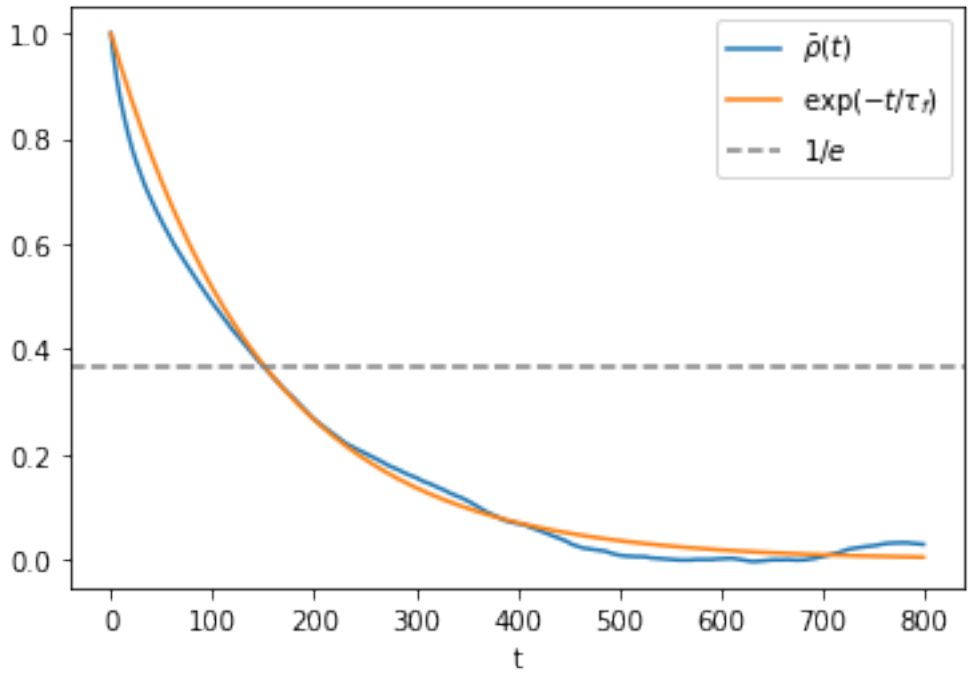
Therefore a rough 1σ-estimate becomes

$$\mathbb{E}[f(X)] = \overline{f(X)}_n \pm \sqrt{\frac{2\tau_f}{n}\bar{\gamma}(0)}.$$

Intuitively, we can understand this result in comparison with the **i.i.d.**-case, where the $2\tau_f$ factor was missing from the square root. Averaging over $f(X_i)$ over all times $i = 1, 2, \ldots, n$ is essentially as good as averaging $f(X_i)$ at times $i = 0, 2\tau_f, 4\tau_f, \ldots$. In the latter case the samples are almost independent, but we have only $n/(2\tau_f)$ of them. The values that were skipped were sufficiently correlated that they did not contribute much to the accuracy of the average. Therefore the error is as if we were looking at a sequence of $n/(2\tau_f)$ i.i.d. samples.

```
def sample_autocovariance(x,tmax):
    '''Compute the sample autocorrelation of the time series x
    for t = 0,1,...,tmax-1.'''
    x_shifted = x - np.mean(x)
    return np.array([np.dot(x_shifted[:len(x)-t],x_shifted[t:])/(len(x)-t)
                     for t in range(tmax)])

def find_correlation_time(autocov):
    '''Return the index of the first entry that is smaller than autocov[0]/e or␣
 ↪the length of autocov if none are smaller.'''
    smaller = np.where(autocov < np.exp(-1)*autocov[0])[0]
    return smaller[0] if len(smaller) > 0 else len(autocov)

tmax = 800
burnin = 1000
chain = sample_chain(0,delta,distribution_pi,100000)[burnin:]
autocov = sample_autocovariance(chain,tmax)
autocorr_time = find_correlation_time(autocov)
print("autocorrelation time =",autocorr_time)
error = np.sqrt(2*autocorr_time*autocov[0]/len(chain))
print("E[X] = {:.3f} +- {:.3f}".format(np.mean(chain),error))

plt.plot(autocov/autocov[0])
plt.plot(np.exp(-np.arange(tmax)/autocorr_time))
plt.axhline(np.exp(-1),linestyle='--',color='0.5')
plt.xlabel("t")
plt.legend([r"$\bar{\rho}(t)$",r"$\exp(-t / \tau_f)$",r"$1/e$"])
plt.show()
```

```
autocorrelation time = 151
E[X] = 0.841 +- 0.104
```



By running the above code several times, you will notice that the error is in the right ballpark, but is perhaps not as accurate and stable as you would like. In order to arrive at the error we had to make several assumptions and approximations that necessarily introduce systematic biases in the error estimate. Most notably $\hat{\rho}(t)$ is not necessarily well-approximated by an exponential decay e^{-t/τ_f}, even in the limit of large n. We will see more robust and versatile error analysis methods later. Nevertheless, it is almost always a good idea when attacking a problem with MCMC to quickly get an order-of-magnitude estimate of the correlation time τ_f of the observable $f : \Gamma \to \mathbb{R}$ of interest, for two reasons:

- You get an order-of-magnitude estimate of the number n of Markov chain steps needed to reach a desired accuracy.
- If measuring the observable $f(X_i)$ on a given configuration X_i is computationally expensive, you could choose to perform the measurement only once every τ_f steps (or at least that order of magnitude) without increasing the error much. This is called **thinning** (see below).

5.2 MCMC simulation of the 2D Ising model

Let us return to the more interesting ferromagnetic Ising model on a 2D lattice of size $w \times w = N$. We have seen last week how to design a Markov chain on $\Gamma = \{-1, 1\}^N$ that approaches the desired Boltzmann distribution

$$\pi(s) = \frac{1}{Z} e^{-\beta H(s)}, \qquad H(s) = -J \sum_{i \sim j} s_i s_j, \qquad s \equiv (s_1, \dots, s_N) \in \{-1, 1\}^N.$$

Since J and $\beta = 1/(k_\mathrm{B} T)$ only appear in the combination $J\beta$ and we will only consider the ferromagnetic case $J > 0$, we will set $J = 1$ in the following for convenience. At each step of the Markov chain a uniform random site i is selected and a flip $s_i \to s_i' = -s_i$ is proposed. We derived that the appropriate Metropolis-Hastings acceptance probability of this move $s \to s'$ is

$$A(s \to s') = \min(1, e^{-\beta \Delta H}), \qquad \Delta H(s) = 2 s_i \sum_{j \colon i \sim j} s_j.$$

A natural observable is the **magnetization** $M(s) = \sum_{i=1}^{N} s_i$ and its normalized version, the **magnetization per spin** $m(s) = M(s)/N$. However, it is easy to see that $\mathbb{E}[M(s)] = 0$ due to the symmetry of the two spin states ± 1:

$$\pi(-s) = \pi(s), \quad \text{while} \quad M(-s) = -M(s) \quad \Longrightarrow \quad \mathbb{E}[M(s)] = \sum_{s \in \Gamma} \pi(s) M(s) = 0.$$

So it is better to consider the **absolute magnetization** $|M(s)|$. We can easily figure out the behaviour in the low- and high-temperature limit. In the limit $\beta \to \infty$ (or $T \to 0$) the only two states with positive probability $\pi(x)$ are the two completely aligned states, hence $\mathbb{E}[|m(s)|] = 1$. When $\beta = 0$ (or $T \to \infty$) the distribution $\pi(x)$ is uniform on all spin states. The difference between the number of $+1$ and -1 spins is then of the order $\sqrt{N}$, so $\mathbb{E}[|m(s)|] \approx N^{-1/2}$ is very small for decent size lattices. Another natural observable is the **energy** $H(s)$ itself, which, up to normalization and constant shift, counts the number of aligned nearest neighbor pairs in the configuration s.

Our goal will be to estimate the mean (absolute) magnetization $\mathbb{E}[|M(s)|]$ and mean energy $\mathbb{E}[H(s)]$ for a range of different temperatures. Preferably we would like to vary the lattice size as well, but for now we will concentrate on a fixed size $N = 16 \times 16 = 256$.

5.2.1 Equilibration

As we have seen in the toy problem above, it is important as a first step to get a rough idea of the equilibration time. It is customary to measure equilibration times (and other times in the simulation) not in units of individual Markov chain transitions, but in **sweeps**:

$$\begin{aligned} k \text{ sweeps} &= k \cdot N \text{ Markov chain steps} \\ &= k \text{ attempted spin flips per site on average.} \end{aligned}$$

So let us have a look at the traces of $M(s)$ and $H(s)$ for three different initial configurations: the uniformly random configuration; a completely aligned configuration; and the completely anti-aligned configuration.

```
def attempt_spin_flip_for_trace(config,boltzmannfactor):
    '''Perform Metropolis-Hastings transition on config and return
    change in magnetization and energy.'''
    w = len(config)
    i,j = rng.integers(0,w,2)
    neighbour_sum = config[i,j] * (config[(i+1)%w,j] + config[(i-1)%w,j] +
                                   config[i,(j+1)%w] + config[i,(j-1)%w])
    if neighbour_sum <= 0 or rng.random() < boltzmannfactor**neighbour_sum:
        config[i,j] = -config[i,j]
        return 2*config[i,j], 2*neighbour_sum
    else:
        return 0, 0

def compute_energy(config):
    '''Compute the energy H(s) of the state config (with J=1).'''
    h = 0
    w = len(config)
    for i in range(w):
        for j in range(w):
            h -= config[i,j] * (config[i,(j+1)%w] + config[(i+1)%w,j])
    return h

def compute_magnetization(config):
    '''Compute the magnetization M(s) of the state config.'''
    return np.sum(config)

def get_MCMC_trace(config,beta,n):
    '''Sample first n steps of the Markov chain and produce trace
    of magnetization and energy.'''
    boltz = np.exp(-2*beta)
    trace = np.zeros((n,2))
```

(continues on next page)

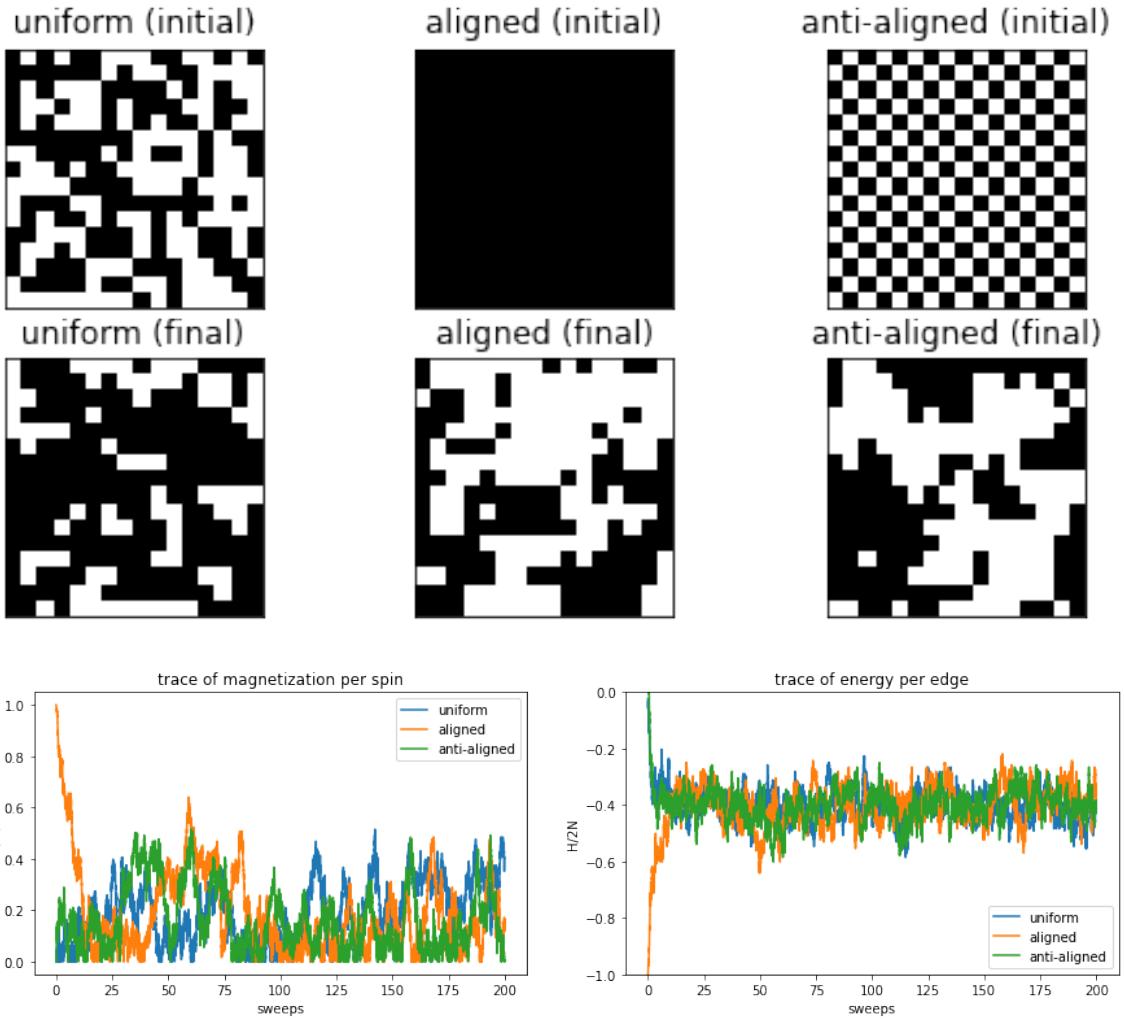
(continued from previous page)

```
    # set the initial magnetization and energy ...
    m = compute_magnetization(config)
    h = compute_energy(config)
    for i in range(n):
        dm, dh = attempt_spin_flip_for_trace(config,boltz)
        # ... and update them after each transition
        m += dm
        h += dh
        trace[i][0] = m
        trace[i][1] = h
    return trace

def uniform_init_config(width):
    '''Produce a uniform random configuration.'''
    return 2*rng.integers(2,size=(width,width))-1

def aligned_init_config(width):
    '''Produce an all +1 configuration.'''
    return np.ones((width,width),dtype=int)

def antialigned_init_config(width):
    '''Produce a checkerboard configuration'''
    if width % 2 == 0:
        return np.tile([[1,-1],[-1,1]],(width//2,width//2))
    else:
        return np.tile([[1,-1],[-1,1]],((width+1)//2,(width+1)//2))[:width,:width]

def plot_ising(config,ax,title):
    '''Plot the Ising configuration.'''
    ax.matshow(config, vmin=-1, vmax=1, cmap=plt.cm.binary)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])

# set parameters
beta = 0.33
width = 16
sweeps = 200
nsites = width * width
length = sweeps * nsites
initializers = [uniform_init_config, aligned_init_config, antialigned_init_config]
labels = ["uniform","aligned","anti-aligned"]

# produce the traces and plot initial and final configurations
traces = []
fig, ax = plt.subplots(2,3,figsize=(8,4))
for i, init in enumerate(initializers):
    config = init(width)
    plot_ising(config,ax[0][i],labels[i] + " (initial)")
    traces.append(get_MCMC_trace(config,beta,length))
    plot_ising(config,ax[1][i],labels[i] + " (final)")
plt.show()

# plot magnetization traces
fig, (ax1,ax2) = plt.subplots(1,2,figsize=(15,4))
xrange = np.arange(length)/nsites
for i in range(3):
    ax1.plot(xrange,np.abs(traces[i][:,0]/nsites))
ax1.legend(labels)
```

(continues on next page)

(continued from previous page)

```
ax1.set_xlabel("sweeps")
ax1.set_ylabel("|m|")
ax1.title.set_text("trace of magnetization per spin")

# plot energy traces
for i in range(3):
    ax2.plot(xrange,traces[i][:,1]/(2*nsites))
ax2.legend(labels)
ax2.set_xlabel("sweeps")
ax2.set_ylabel("H/2N")
ax2.set_ylim(-1,0)
ax2.title.set_text("trace of energy per edge")
plt.show()
```

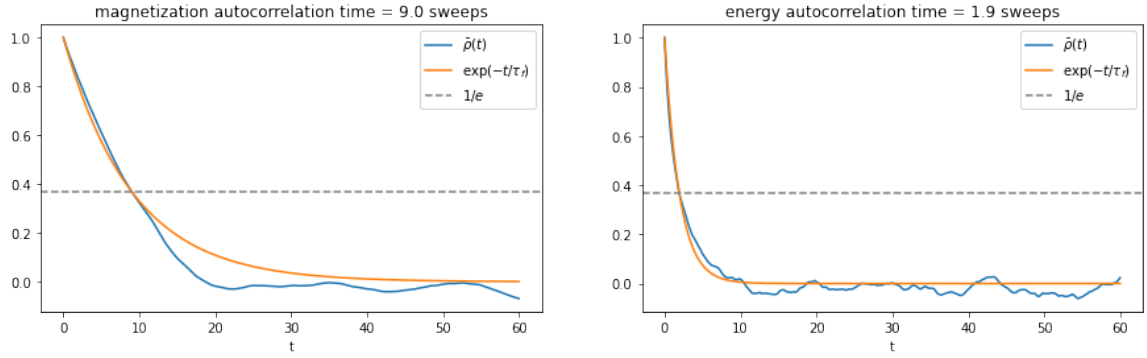


In equilibrium, i.e. when the Markov chain has approached the stationary distribution, the three curves should be close together, or at least fluctuating within the same interval. The equilibration time τ_{eq} is therefore at least as large as the time it takes for the traces to meet each other. Judging from the plot $\tau_{\text{eq}} \gtrsim 25$ sweeps. Note also that the magnetization appears to take longer to reach equilibrium. This is a general phenomenon: in principle we should look at a large collection of observables and wait until all traces have suitable converged to conclude on the equilibration time. In practice, however, if it is the magnetization $|m|$ we wish to estimate, it is typically regarded to be sufficient that its own trace displays equilibrium. To be safe, it is a good idea to spend a number of sweeps in the equilibration phase that is a multiple of the suspected equilibration time τ_{eq}. In this case we choose to equilibrate with 500 sweeps.

5.2.2 Autocorrelation

Next we need an order of magnitude estimate of the autocorrelation time.

```
# gather the required trace
equil_sweeps = 500
autocorr_sweeps = 1600
length = nsites * (equil_sweeps + autocorr_sweeps)
config = uniform_init_config(width)
trace = get_MCMC_trace(config,beta,length)[nsites * equil_sweeps:]

# compute the autocorrelation over a time difference of 60 sweeps
tmax = nsites * 60
fig, ax = plt.subplots(1,2,figsize=(15,4))
for i in range(2):
    autocov = sample_autocovariance(trace[:,i],tmax)
    autocorr_time = find_correlation_time(autocov)
    ax[i].plot(np.arange(tmax)/nsites,autocov/autocov[0])
    ax[i].plot(np.arange(tmax)/nsites,np.exp(-np.arange(tmax)/autocorr_time))
    ax[i].axhline(np.exp(-1),linestyle='--',color='0.5')
    ax[i].set_xlabel("t")
    ax[i].title.set_text("{} autocorrelation time = {:.1f} sweeps"
                         .format(["magnetization","energy"][i],autocorr_time/
 ↪nsites))
    ax[i].legend([r"$\bar{\rho}(t)$",r"$\exp(-t / \tau_{f})$",r"$1/e$"])
plt.show()
```



Here we see a similar phenomenon that the estimated autocorrelation time depends on the observable at hand.

5.2.3 Thinning

We see that for this particular temperature the autocorrelation time is of the order of several sweeps. We could go ahead to estimate the mean magnetization from the trace and calculate the error from the sample variance and autocorrelation time. Instead, let us use the estimate to perform **thinning**: we re-run the simulation with a larger number of sweeps, but this time we only perform a measurement every m sweeps. We choose m between 1 and $\tau_{|M|}$, say $m = 4$. Why? Taking $m \gtrsim \tau_{|M|}$ would be wasteful, because the magnetization values are already largely uncorrelated after such a time interval. An advantage of thinning is that we do not need to keep track of the observables at every Markov step, saving computation time. However, this means that we have to compute the magnetization and energy every time we wish to measure it, which requires visiting all N sites. This will take about the same order of time as performing N transitions, i.e. 1 sweep. A good rule of thumb is to spend less than half of the simulation time on measurements, implying that we should take $m \gtrsim 1$. Let's gather some data with the choice $m = 4$.

```
def attempt_spin_flip(config,boltzmannfactor):
    '''Perform Metropolis-Hastings transition on config.'''
    w = len(config)
    i,j = rng.integers(0,w,2)
```

(continues on next page)

(continued from previous page)

```
    neighbour_sum = config[i,j] * (config[(i+1)%w,j] + config[(i-1)%w,j] +
                                   config[i,(j+1)%w] + config[i,(j-1)%w])
    if neighbour_sum <= 0 or rng.random() < boltzmannfactor**neighbour_sum:
        config[i,j] = -config[i,j]

def run_ising_MCMC(config,beta,n):
    '''Perform n steps of the MH Markov chain on config.'''
    boltz = np.exp(-2*beta)
    for _ in range(n):
        attempt_spin_flip(config,boltz)

equil_sweeps = 500
measure_sweeps = 4
num_measurements = 600

# equilibration phase
config = uniform_init_config(width)
run_ising_MCMC(config,beta,equil_sweeps * nsites)

# measurement phase
magnetizations = np.zeros(num_measurements)
energies = np.zeros(num_measurements)
for i in range(num_measurements):
    run_ising_MCMC(config,beta,measure_sweeps * nsites)
    magnetizations[i] = compute_magnetization(config)
    energies[i] = compute_energy(config)

print("estimated mean magnetization per site =", np.mean(np.abs(magnetizations))/
 ↪nsites)
print("estimated mean energy per edge =", np.mean(energies)/(2*nsites))
```

```
estimated mean magnetization per site = 0.170703125
estimated mean energy per edge = -0.40541666666666665
```

Suppose we do not quite trust our estimate of the autocorrelation time or we wish to estimate a quantity that is not directly an expectation value but can be computed from the samples, like the **magnetic susceptibility**

$$\chi := \beta N \operatorname{Var}(m) = \beta N(\mathbb{E}[m^2] - \mathbb{E}[m]^2),$$

then how can we estimate the error? If we had enough computing resources and patience we would repeat the whole experiment $k \approx 20$ times independently, compute the desired quantity k times, and determine the mean and standard error for the resulting i.i.d. random outcomes. Sometimes this is a good way to go, for instance when the equilibration time is very short. Quite often though one would like to base estimates on a single long run of the MCMC. Luckily there are several statistical tricks available that allows us to use a single batch of data multiple times in order to mimic the repetition of a whole experiment, including **batching** and **jackknife resampling**.

5.2.4 Batching

Batching or **blocking** is a very simple technique to mimic k repetitions of the experiment, simply by chopping the data set into k equal size batches, usually between 10 and 1000. The desired quantity is computed for each batch independently and the outcomes are treated as i.i.d. random variables, for which we know how to estimate the mean and standard error. This procedure is justified when the batch size n/k is much larger than the autocorrelation time τ_f, because then most of the measurements in one batch are uncorrelated with those in another batch.

```
def batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each.
    Returns the mean and standard error.'''
```

(continues on next page)

(continued from previous page)

```
    batches = np.reshape(data,(k,-1))
    values = np.apply_along_axis(observable, 1, batches)
    return np.mean(values), np.std(values)/np.sqrt(k-1)

num_batches = 50
magnet_est = batch_estimate(np.abs(magnetizations)/nsites,lambda x: np.mean(x),␣
↪num_batches)
suscep_est = batch_estimate(magnetizations/nsites,lambda x: beta*nsites*np.var(x),
↪ num_batches)
print("mean magnetization: E[|m|] = {:.4f} +- {:.4f}".format(*magnet_est))
print("magnetic susceptibility: chi = {:.4f} +- {:.4f}".format(*suscep_est))
```

```
mean magnetization: E[|m|] = 0.1707 +- 0.0076
magnetic susceptibility: chi = 2.2799 +- 0.1823
```

5.2.5 Jackknife

Batching works well for estimating expectation values, but can become inaccurate for more elaborate quantities, because individual batches may be too small to sample the quantity well. In such a case an option is to use **jackknife resampling**. The general idea is simple. Suppose we have divided our data into k batches. Instead of computing our quantity of interest independently in the k small batches, we compute in the k datasets that are obtained by considering $k-1$ of the k batches combined, each time leaving one batch out. Of course the k different values obtained in this way are highly correlated, but one can show that this effect can be compensated by taking the error to be $\sqrt{k-1}$ times the standard deviation of the k values (instead of the standard error which is $1/\sqrt{k-1}$ times the standard deviation).

```
def jackknife_batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each
    collection of all but one batches. Returns the mean and corrected
    standard error.'''
    batches = np.reshape(data,(k,-1))
    values = [observable(np.delete(batches,i,0).flatten()) for i in range(k)]
    return np.mean(values), np.sqrt(k-1)*np.std(values)

num_batches = 50
magnet_est = jackknife_batch_estimate(np.abs(magnetizations)/nsites,
                                      lambda x: np.mean(x), num_batches)
suscep_est = jackknife_batch_estimate(magnetizations/nsites,
                                      lambda x: beta*nsites*np.var(x), num_
↪batches)
print("mean magnetization: E[|m|] = {:.4f} +- {:.4f}".format(*magnet_est))
print("magnetic susceptibility: chi = {:.4f} +- {:.4f}".format(*suscep_est))
```

```
mean magnetization: E[|m|] = 0.1707 +- 0.0076
magnetic susceptibility: chi = 3.6460 +- 0.2987
```

Note that the estimate and error in the case of the expectation value are unchanged compared to the batching method (with the same number of batches), but for quantities that are not expectation values, like the magnetic susceptibility, the result will differ.

5.3 Further reading

Toy problem is adapted from: Section 11.5 of [Owe13].

Ising model simulation in practice: Chapter 4 of [NB99]. (See in particular Section 3.4 about the error analysis)

An accessible discussion on the different error estimates can be found in [Rum].

Chapter 6

Criticality & Cluster algorithms

The Markov Chain Monte Carlo (MCMC) algorithms we have seen so far have involved **local updates**. In the case of the Ising model, the proposed transitions were of the type of a single spin flip, while in the disk model a single disk was moved at each transition. The reason for choosing proposals that involve only local changes to the configuration should be clear: the closer the proposed state y is to the current state x, the more likely the ratio $\pi(y)/\pi(x)$ is close to 1, so decent acceptance rates can be maintained. A transition that flips many spins at once in the Ising model generically increases the energy by a large amount, so the Metropolis-Hastings acceptance probability would be microscopic. The downside of performing only local moves is that it requires many to change the state significantly. In other words **autocorrelation times** can be very large. This is something you have observed for the 2d Ising model in last week's exercises. In the low-temperature and high-temperature regimes the autocorrelation time is manageable and of the order of several sweeps. However, around $T = 2.2$ the autocorrelation time is seen to be much larger.

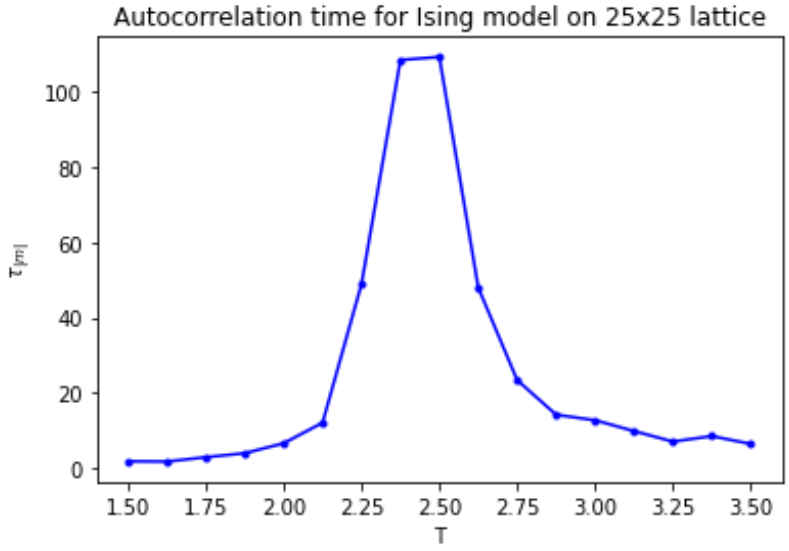


Fig. 6.1: Autocorrelation time of the Ising model.

6.1 Critical phenomena

As you may have noticed from your measurements of the magnetization, this is roughly where the transition occurs between the ordered and disordered phase of the ferromagnet. In fact, Onsager calculated the mean absolute magnetization per spin $\mathbb{E}|m|$ exactly in the limit of large lattices to be

$$\lim_{w\to\infty} \mathbb{E}[|m|] = \left(1 - [\sinh 2\beta J]^{-4}\right)^{\frac{1}{8}}.$$

Of course, this formula can only hold as long as $\sinh 2\beta J \geq 1$, i.e. $T \leq T_c$ where T_c is the critical temperature

$$T_c = \frac{2J}{\log(1+\sqrt{2})} \approx 2.269J.$$

For $T \geq T_c$ the magnetization vanishes, $\lim_{w\to\infty} \mathbb{E}[|m|] = 0$. Hence, the mean absolute-magnetization per site $\mathbb{E}[|m|]$ serves as an **order parameter** for the phase transition.

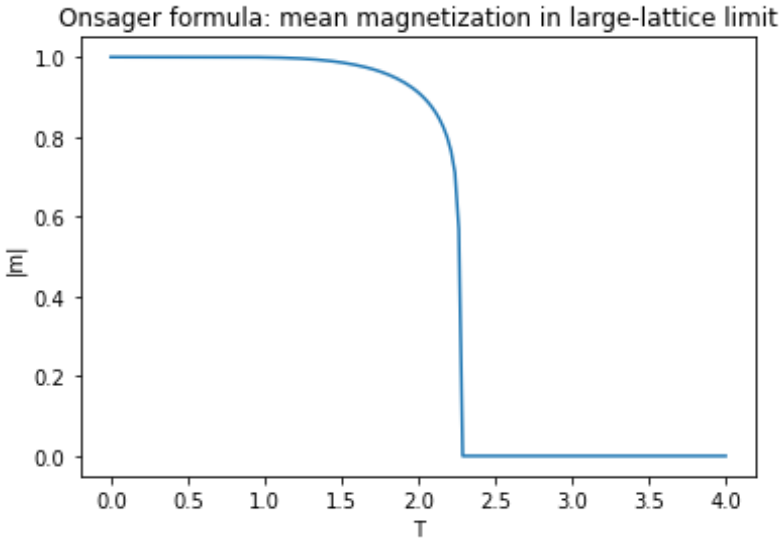


Fig. 6.2: Onsager solution for the magnetization.

A phase transition like this where the order parameter is continuous through the phase transition is called a **continuous phase transition**. Such phase transitions have a severe impact on **correlation functions**, which in the case of the Ising model can be introduced as

$$C(\vec{r}) = \mathbb{E}[s_{\vec{x}} s_{\vec{x}+\vec{r}}] - \mathbb{E}[s_{\vec{x}}]\mathbb{E}[s_{\vec{x}+\vec{r}}], \qquad \vec{r}, \vec{x} \in \mathbb{Z}^2.$$

Here $s_{\vec{x}}$ refers to the spin at site $\vec{x} \in \mathbb{Z}^2$. Note that because of translation symmetry $C(\vec{r})$ does not depend on $\vec{x}$. When $\vec{r}$ is large with respect to the lattice spacing and $T \neq T_c$, then we have that $C(\vec{r}) \sim ce^{-|\vec{r}|/\xi}$, where ξ is the **correlation length**. An important lesson from statistical physics is that continuous phase transitions give rise to **critical phenomena**: on an infinite lattice the correlation length ξ diverges as the temperature T approaches the critical temperature T_c,

$$\xi \sim |T - T_c|^{-\nu}.$$

The quantity ν is called a **critical exponent**. Such quantities are of particular physical interest, because they do not depend on the precise parameters of the model or on the type of lattice (whether square or triangular for instance). The critical exponent ν is therefore called a **universal** property of the Ising model.

When exactly at the critical temperature and on an infinite lattice, the correlation length of the Ising model is infinite, which manifests itself in the presence of spin clusters of every imaginable size. In fact, a feature of a statistical system at criticality is that it acquires a notion of **scale-invariance**. If one zooms out far enough, such that individual lattice sites are no longer visible, i.e. such that spin configuration has become continuous, then one can no longer tell the

difference between one configuration and another that is zoomed out even further. This is nicely illustrated for the 2d Ising model in this YouTube video. If you like, a quantity is universal if it is a property of this exactly scale-invariant regime. In general these scale-invariant models are described by Conformal Field Theories.

One can also investigate the situation where one starts at a non-critical temperature, e.g. $T > T_c$, and zooms out while tuning the temperature towards criticality. More precisely, we can describe "zooming out" as assigning smaller and smaller physical edge length a to the edges of the square lattice. If we scale the lattice spacing a like $|T-T_c|^\nu$, then the physical correlation length $a\xi$ and the corresponding rescaled correlation function $C(\vec{x}/a)$ have well-defined limits. The result is a continuous spin system that is not scale-invariant, but possesses a non-vanishing finite correlation-length.

The upshot of all this is that a lot of physics, in particular field theories, can be found at or close to critical points in discrete statistical systems. This is an important observation for Monte Carlo simulations, because we simply cannot simulate continuous fields with infinite degrees of freedom. One way or another fields have to be discretized before they can even be represented in a computer. If at all possible we would like our simulation results not to depend on the details of the chosen discretization, hence we would like to rely on the universality that can be found at critical points of the discretized system.

6.2 Critical slowing down

Of course, on a finite $w \times w$ lattice the correlation length cannot diverge and one cannot pinpoint an exact phase transition. But it is reasonable to assume that close enough to the critical temperature T_c the spins at opposite ends of the lattice are non-negligibly correlated. This is something that can be seen by looking at individual configurations, like the following on a 100×100 lattice at a temperature below, close to, and above T_c.

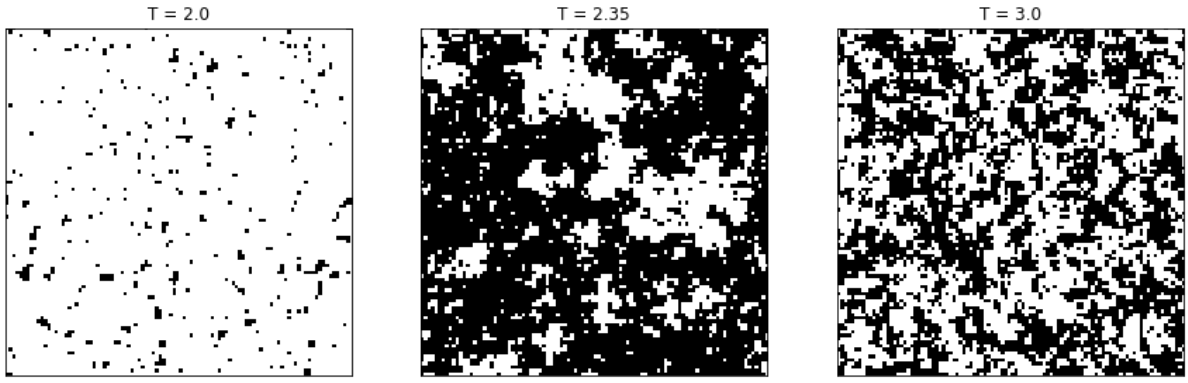


Fig. 6.3: Samples of the Ising model at three different temperatures.

The critical correlation between far-away spins is disastrous for the autocorrelation time of the Metropolis-Hastings spin-flip Markov chain: a phenomenon known as **critical slowing down**. One can imagine that destroying spin clusters that extend over almost the whole lattice with only random single-spin flips is a time-consuming affair. This can be made quantitative by looking at the **dynamic exponent** z of the spin-flip algorithm, which is defined as the exponent appearing in the relation between the autocorrelation time τ in units of sweeps and the temperature T or correlation length ξ,

$$\tau \sim |T - T_c|^{-z\nu} \sim \xi^z.$$

On a finite lattice we cannot have $\xi > w$, so close to criticality we expect $\tau \sim w^z$. Since the value of z for the spin-flip is estimated at $z \approx 2.17$, the auto-correlation time at criticality increases very rapidly with the lattice size.

Critical slowing down is a serious obstacle in many MCMC simulations. Luckily the dynamic exponent z is not a universal property of the statistical model at hand, but depends on the chosen MCMC algorithm. In some cases it is possible to find algorithms that have much smaller exponents. Cluster algorithms are a particularly well-known family of these.

6.3 Cluster algorithm for the Ising model: the Wolff algorithm

We start by describing the transition algorithm and below we will investigate why it satisfies the detailed balance condition. Given a configuration $\mathbf{s} \in \{-1, 1\}^N$ of the $w \times w$ Ising model. The next state $\mathbf{s}'$ in the Markov chain is obtained from $\mathbf{s}$ by flipping all spins in a random cluster C, i.e.

$$s'_i = \begin{cases} -s_i & \text{if } i \in C \\ s_i & \text{if } i \notin C \end{cases}.$$

This cluster C is constructed in an iterative manner starting with a cluster consisting of single "seed" site i_{seed}, which is uniformly chosen in the lattice. Let $\sigma = s_{i_{\text{seed}}}$ be its spin value. The goal is to grow the cluster by iteratively adding neighboring sites that have the same spin σ with appropriate probability. More precisely, each site in the cluster is to be *visited* once. Note that initially only the seed is there, so that is the first site to be visited, but later more sites will be added and the order in which they are then visited does not matter. Upon visiting a site each neighbor that is not yet part of the cluster is added to the cluster with independent probability p_{add}. The process stops when no more unvisited sites remain in the cluster. This resulting cluster is C, and to finish the Wolff transition we should flip each of the spins to $-\sigma$. Note that this is a **rejection-free algorithm** in the sense that $\mathbf{s}'$ is guaranteed to be different from $\mathbf{s}$.

When implementing this algorithm it is actually much more convenient to flip the spins directly at the moment they are added to the cluster. This way one does not have to keep track of the full cluster throughout the procedure, but only of the yet unvisited sites in the cluster. Since only neighbors are added that have spin σ, those are guaranteed to be not yet part of the cluster (visited or unvisited). The algorithm can therefore be implemented as follows.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def aligned_init_config(width):
    '''Produce an all +1 configuration.'''
    return np.ones((width,width),dtype=int)

def plot_ising(config,ax,title):
    '''Plot the configuration.'''
    ax.matshow(config, vmin=-1, vmax=1, cmap=plt.cm.binary)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])

from collections import deque

def neighboring_sites(s,w):
    '''Return the coordinates of the 4 sites adjacent to s on an w*w lattice.'''
    return [((s[0]+1)%w,s[1]),((s[0]-1)%w,s[1]),(s[0],(s[1]+1)%w),(s[0],(s[1]-1)
 ↪%w)]

def cluster_flip(state,seed,p_add):
    '''Perform a single Wolff cluster move with specified seed on the state with␣
 ↪parameter p_add.'''
    w = len(state)
    spin = state[seed]
    state[seed] = -spin
    cluster_size = 1
    unvisited = deque([seed])    # use a deque to efficiently track the unvisited␣
 ↪cluster sites
    while unvisited:    # while unvisited sites remain
        site = unvisited.pop()  # take one and remove from the unvisited list
```

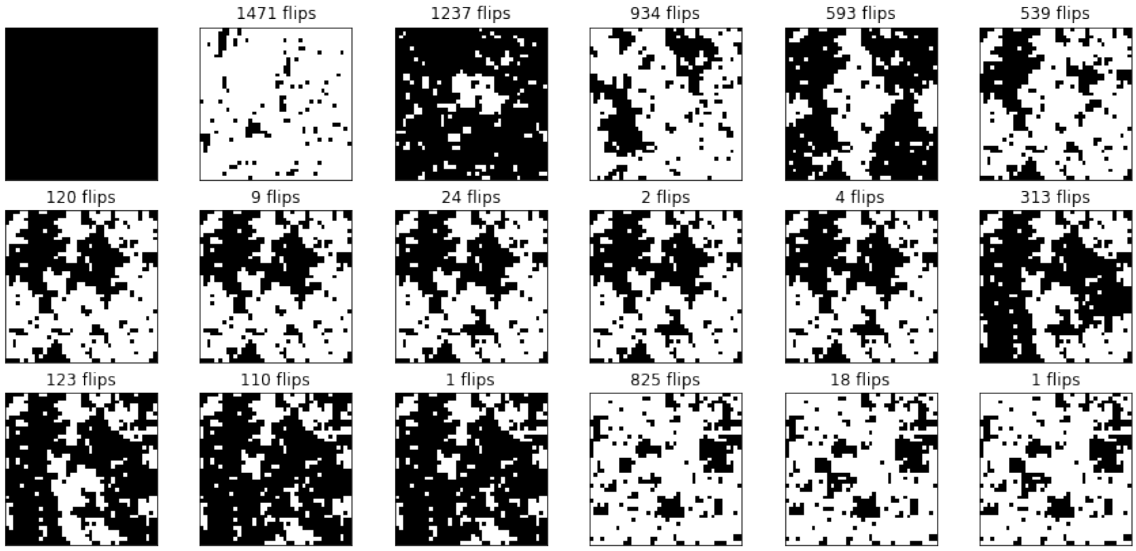
(continues on next page)

(continued from previous page)

```
        for nbr in neighboring_sites(site,w):
            if state[nbr] == spin and rng.uniform() < p_add:
                state[nbr] = -spin
                unvisited.appendleft(nbr)
                cluster_size += 1
    return cluster_size

def wolff_cluster_move(state,p_add):
    '''Perform a single Wolff cluster move on the state with addition probability␣
↪p_add.'''
    seed = tuple(rng.integers(0,len(state),2))
    return cluster_flip(state,seed,p_add)
```

Here is an illustration of the first 17 states in the corresponding Markov chain starting with the fully aligned state on a 40×40 lattice at temperature $T = 2.4$.

```
width = 40
temperature = 2.4
p_add = 1 - np.exp(-2/temperature)
config = aligned_init_config(width)
fig, axs = plt.subplots(3,6,figsize=(15,7))
flips = 0
for axrow in axs:
    for ax in axrow:
        plot_ising(config,ax,"{} flips".format(flips) if flips > 0 else "")
        flips = wolff_cluster_move(config,p_add)
```



It should be clear from the figures that the Markov chain explores the state space a lot quicker than the single-spin flip Markov chain does. This does not yet imply that it is significantly faster in cpu time, because a single Wolff move in which many spins are flipped of course takes proportionally more time than a single-spin flip. Note also that irrespective of the temperature there is a positive chance, at least $(1 - p_{\text{add}})^4$, that a Wolff move only flips a single spin. It is therefore not so easy to reason about the efficiency or even judge what would be a "good" autocorrelation time (whether an autocorrelation time of, say, 100 is good depends a lot on whether the clusters mostly consist of only a few sites or comprise nearly the full lattice!). So we should also get an idea of the average size of clusters depending on temperature.

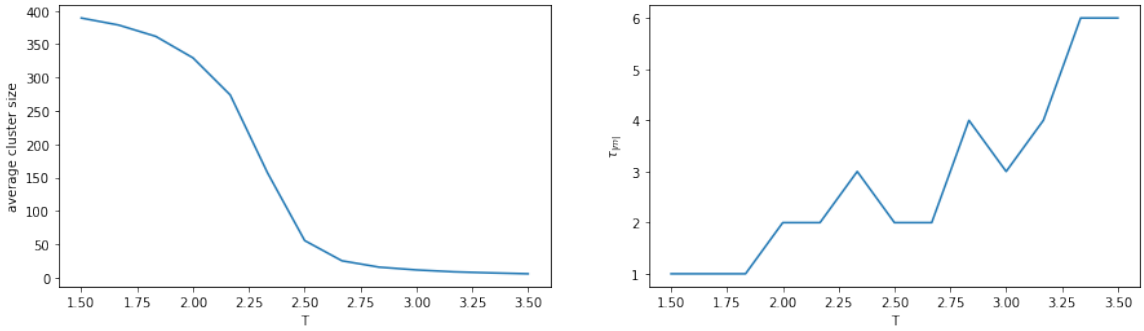
```
def compute_magnetization(config):
    '''Compute the magnetization M(s) of the state config.'''
```

(continues on next page)

(continued from previous page)

```
    return np.sum(config)

def run_ising_wolff_mcmc(state,p_add,n):
    '''Run n Wolff moves on state and return total number of spins flipped.'''
    total = 0
    for _ in range(n):
        total += wolff_cluster_move(state,p_add)
    return total

def sample_autocovariance(x,tmax):
    '''Compute the autocorrelation of the time series x for t = 0,1,...,tmax-1.'''
    x_shifted = x - np.mean(x)
    return np.array([np.dot(x_shifted[:len(x)-t],x_shifted[t:])/len(x)
                     for t in range(tmax)])

def find_correlation_time(autocov):
    '''Return the index of the first entry that is smaller than
    autocov[0]/e or the length of autocov if none are smaller.'''
    smaller = np.where(autocov < np.exp(-1)*autocov[0])[0]
    return smaller[0] if len(smaller) > 0 else len(autocov)

width = 20
nsites = width*width
temperatures = np.linspace(1.5,3.5,13)
equil_moves = 200
autocorr_moves = 2400
tmax = 70

autocorr_times = []

for temp in temperatures:
    p_add = 1 - np.exp(-2/temp)
    state = aligned_init_config(width)
    run_ising_wolff_mcmc(state,p_add,equil_moves)
    total_flips = 0
    trace = np.zeros(autocorr_moves)
    for i in range(autocorr_moves):
        total_flips += run_ising_wolff_mcmc(state,p_add,1)
        trace[i] = compute_magnetization(state)
    autocov = sample_autocovariance(np.abs(trace),tmax)
    time = find_correlation_time(autocov)
    autocorr_times.append((time,total_flips/(autocorr_moves)))
```

```
fig, ax = plt.subplots(1,2,figsize=(15,4))
ax[0].plot(temperatures,[av_size for tau,av_size in autocorr_times])
ax[0].set_xlabel("T")
ax[0].set_ylabel("average cluster size")
ax[1].plot(temperatures,[tau for tau,av_size in autocorr_times])
ax[1].set_xlabel("T")
ax[1].set_ylabel(r"$\tau_{|m|}$")
plt.show()
```



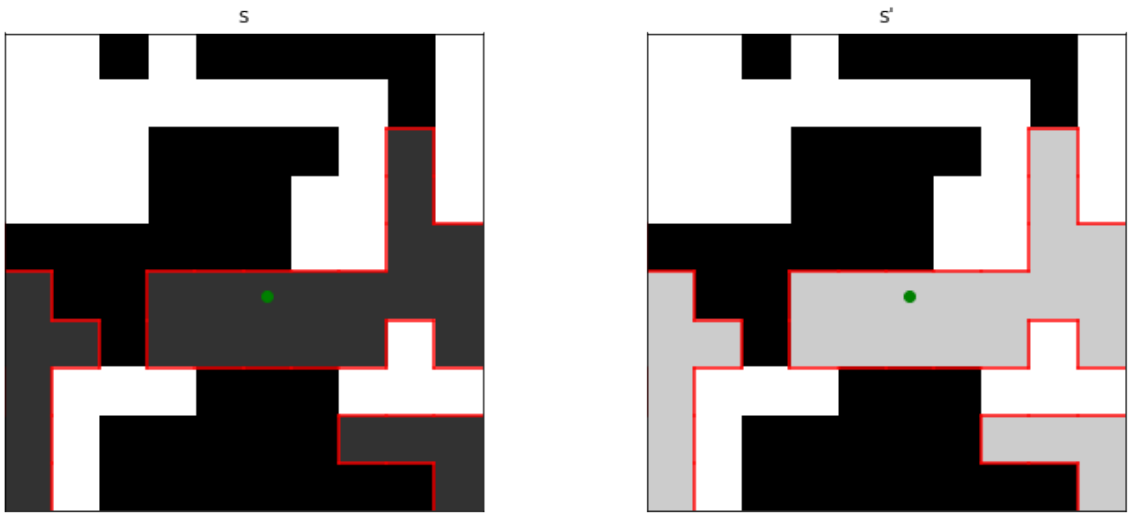
You should take the second plot with a pinch of salt. At low temperature the magnetization oscillates wildly because the typical cluster size is of the order of the full lattice, so the autocovariance function will not fit nicely to an exponential decay. The minimal possible value for $\tau_{|m|}$ in this case is 2, meaning that no autocorrelation was detected. At higher temperature, more data is required to get an accurate estimate. Nevertheless a qualitative picture emerges: to decorrelate the magnetization measurements it suffices to perform a number of Wolff moves such that on average each site is flipped once. Put quantitatively, if $c(T)$ is the average cluster size in a Wolff move, then a **sweep** corresponds to performing roughly $N/c(T)$ moves. Only one or a few sweeps appear to be necessary to produce independent states, even close to the critical temperature!

Let us convince ourselves that the transition matrix $P(s \to s')$ of the Wolff algorithm satisfies the assumptions of last week's convergence theorem with limiting distribution $\pi(s) = \frac{1}{Z} e^{-\beta H(s)}$. Assuming positive temperature $T > 0$, we have $0 < p_{\text{add}} < 1$. So the Wolff moves include the single-spin flips as possibilities, implying that $P(s \to s')$ is irreducible. The transition matrix does not include rejections, but is still aperiodic because it is possible to return to a state s both in two and in three moves (e.g. via several single or double spin flips). It only remains to prove that P satisfies detailed balance,

$$\pi(s)P(s \to s') = \pi(s')P(s' \to s).$$

The plots below show an example of a transition $s \to s'$ as well as the inverse $s' \to s$. The green dots represent the seed and the cluster boundaries are drawn in red.

```
width = 10
temp = 2.7
p_add = 1 - np.exp(-2/temp)

# produce an equilibrated state
states = [aligned_init_config(width)]
run_ising_wolff_mcmc(states[0],p_add,100)
# perform wolff move on a copy with the seed nicely in the middle
states.append(np.copy(states[0]))
seed = (width//2,width//2)
cluster_flip(states[1],seed,p_add)

fig, ax = plt.subplots(1,2,figsize = (12,5))
diff = states[0]*states[1]
vert_boundary = np.nonzero(np.roll(diff,1,axis=1)-diff)
hor_boundary = np.nonzero(np.roll(diff,1,axis=0)-diff)
for i in range(2):
    plot_ising(0.8*states[i]+0.2*states[1-i],ax[i],["s","s'"][i])
    for y,x in np.transpose(vert_boundary):
        ax[i].plot([x-0.5,x-0.5],[y-0.5,y+0.5],'-r')
    for y,x in np.transpose(hor_boundary):
        ax[i].plot([x-0.5,x+0.5],[y-0.5,y-0.5],'-r')
    ax[i].scatter(*seed,color='green')
plt.show()
```



Note that the move $s \to s'$ is uniquely characterized by the cluster C, but there are many ways in which the cluster could have been constructed by the algorithm. Indeed, any of the sites in the cluster could have been the seed and the remaining sites in the cluster could have been added in many different orders. The important observation is that, given the cluster C, the set of such possibilities is exactly the same for the for $s \to s'$ and its inverse $s' \to s$. So if we can verify detailed balance for each particular construction of C, then detailed balance for the union of all constructions is guaranteed as well! We may thus assume that the seeds for both moves are chosen in the same place and that neighbors are explored in the same order. Where do the probabilities in the construction $s \to s'$ and $s' \to s$ differ? By construction, the spins within C are all aligned, so the only difference occurs at the boundary of C. Let $m+n$ be the length of the boundary of C, where m is the number of edges separating aligned sites and n the number of edges separating anti-aligned sites. The probability of the cluster C being selected in s is proportional to $(1-p_{\text{add}})^m$, because for each of the m edges there was an independent $1-p_{\text{add}}$ chance for the neighbor not to be added. Note that the cluster could not have been extended to anti-aligned sites, so the n edges bordering those sites do not contribute to the probability. For the state s' the roles of m and n are exactly interchanged, so there the probability is proportional to $(1-p_{\text{add}})^n$. Hence

$$\frac{P(s \to s')}{P(s' \to s)} = (1-p_{\text{add}})^{m-n}.$$

On the other hand, the energy difference between the two states is also determined entirely by m and n, because the boundary is the only place where changes from aligned to anti-aligned occur (and vice versa). Examining the energy $H(s) = -J\sum_{i\sim j} s_i s_j$, we easily find

$$H(s') = H(s) + 2Jm - 2Jn,$$

implying that the ratio of the Boltzmann distribution is given by

$$\frac{\pi(s')}{\pi(s)} = e^{-\beta(H(s')-H(s)} = e^{-2\beta J(m-n)}.$$

Indeed, taking $p_{\text{add}} = 1 - e^{-2\beta J}$ ensures detailed balance

$$\frac{P(s \to s')}{P(s' \to s)} = \frac{\pi(s')}{\pi(s)}.$$

This finishes the proof.

As a final remark, let me mention that there also exists a global version of the Wolff algorithm, known as the Swendsen-Wang algorithm. It is global in the sense that the full lattice is partitioned into clusters, instead of growing a single cluster, after which each cluster is flipped with an independent probability $1/2$.

6.4 Cluster algorithm for the Disk Model

Also the disk model admits a cluster algorithm, in the sense that there exists a rejection-free algorithm that relies on selecting a cluster of disks, which is then collectively updated. In the case of the Ising model, the cluster algorithm crucially relied on the symmetry between $+1$ and -1 spins to ensure detailed balance. No analogous symmetry exists for individual disks, but one can take advantage of the translational and point-reflection symmetry of the disk model on the torus. Recall that in the disk model, the desired distribution is given by the probability density $\pi(\mathbf{x}) = 1/Z$ on the non-overlapping configurations $\mathbf{x}$. To ensure detailed balance we thus need a symmetric transition density $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}, \mathbf{x})$. This is easily achieved by exploiting the symmetries of the torus, as was done in the exercises a while back by moving a single disk by a symmetrically distributed random displacement. The difficult part is to do this without rejection, i.e. to guarantee that the resulting configuration has no overlaps. One way to do this is via the following **geometric cluster algorithm**.

Suppose we have a non-overlapping configuration $\mathbf{x} = (x_1, \dots, x_N) \in [0, L)^{2N}$ of N disks on the $L \times L$ torus. A new configuration $\mathbf{y}$ is generated as follows (see the figure below).

1. Sample a uniform point $p \in [0, L)^{2N}$ in the torus, called the "pivot" (the green dot in the figure).
2. Consider the configuration $\tilde{\mathbf{x}} \in [0, L)^{2N}$ obtained by point-reflecting all disks in the pivot, i.e. $\tilde{x}_i = 2p - x_i$ modulo L (the orange disks in the figure).
3. Superimpose the disk configurations $\mathbf{x}$ and $\tilde{\mathbf{x}}$ and partition them into clusters of overlapping disks. Note that clusters may occur in pairs that are symmetric with respect to the pivot, but the partition $\{1, 2, \dots, N\} = c_1 \cup \cdots \cup c_k$ of the indices of the disks is unambiguous.
4. For each cluster c_j independently with probability $1/2$ either leave all disks unmoved and set $y_i = x_i$, $i \in c_j$, or point-reflect all of them and set $y_i = \tilde{x}_i, i \in c_j$.

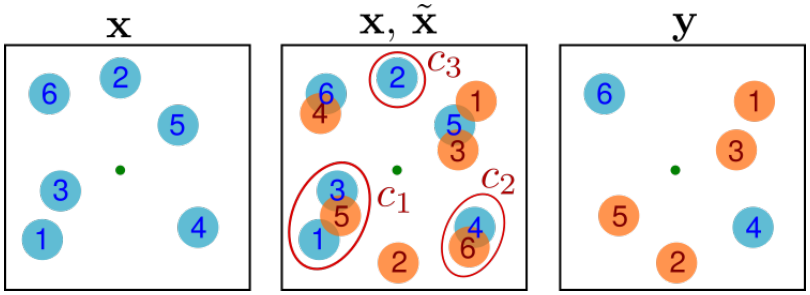


Fig. 6.4: Geometric cluster algorithm. Here the partition is $\{1, 3, 5\} \cup \{4, 6\} \cup \{2\}$.

It is not hard to convince yourself that the resulting configuration $\mathbf{y}$ is non-overlapping and that the transition density is symmetric $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}, \mathbf{x})$.

The geometric cluster algorithm as just described is a global cluster algorithm, like Swendsen-Wang for the Ising model. There is also a local cluster variant, that is easier to implement, the analogue of the Wolff algorithm. In this case a random initial disk is chosen as "seed" in addition to the random pivot, and only the cluster containing the seed is determined. Once the cluster has been found it is point-reflected in the pivot (without tossing a coin). The nice thing is that this can be implemented iteratively. First the seed disk is reflected. If it overlaps any other disks, those are also reflected. And this continues until no more overlaps occur, at which point the Markov chain transition is completed.

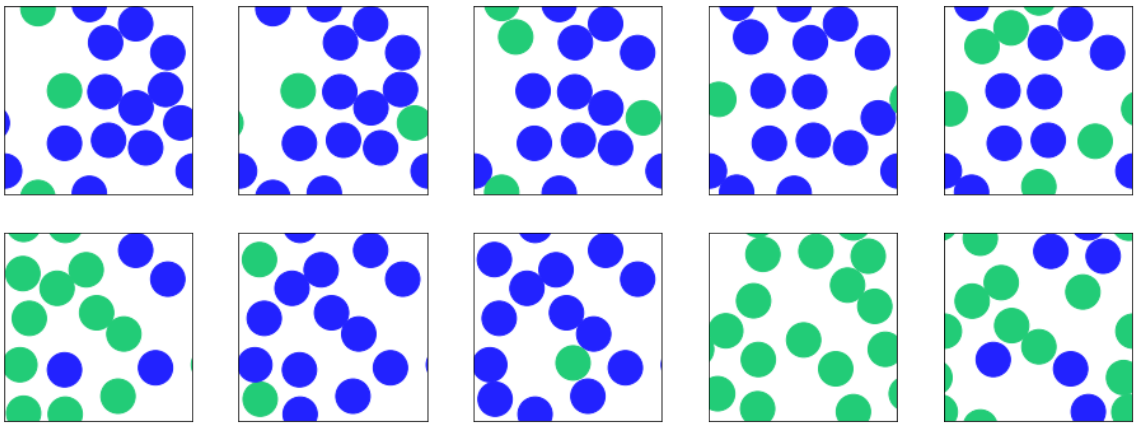
To do this efficiently we need a data structure that can quickly tell us what disks will have an overlap after the reflection. This can be achieved by partitioning the torus into a grid and tracking which sites are occupied by which disk, and maintaining this data in addition to the set of disk coordinates. Of course we should make sure that the grid spacing is small enough, e.g. one unit, such that any site can contain at most one disk. In the following we will assume that L is an integer and use a regular integer grid for this purpose. The occupation is stored in an array with values ranging from -1 to $N - 1$, where -1 means it is empty and $i \geq 0$ means that it contains the center of the disk with label i (see `occupation_array`). To check for overlaps it suffices to examine the nearest and next-nearest neighbors in the grid (see `get_overlap`).

```
def plot_disk_configuration(positions,L,ax,color):
    '''Plot disk configuration. color is an array of bools specifying
    blue or green color.'''
    ax.set_aspect('equal')
    ax.set_ylim(0,L)
    ax.set_xlim(0,L)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])
    for i, x in enumerate(positions):
        # consider all horizontal and vertical copies that may be visible
        for x_shift in [z for z in x[0] + [-L,0,L] if -1<z<L+1]:
            for y_shift in [z for z in x[1] + [-L,0,L] if -1<z<L+1]:
                col = '#2222ff' if color[i] else '#22cc77'
                ax.add_patch(plt.Circle((x_shift,y_shift),1,color=col))

def occupation_array(l,pos):
    '''Return l*l array with indices of positions of disks in regular
    grid, or -1 when empty.'''
    occupation = -np.ones((l,l),dtype='int')
    for i, (x, y) in enumerate(pos):
        occupation[int(x),int(y)] = i
    return occupation

def get_overlap(state,point):
    '''Return a list of indices of disks that have overlap with point.'''
    w = len(state["occ"])
    overlaps = []
    for dx in range(-2,3):
        x = (int(point[0]) + dx)%w
        for dy in range(-2,3):
            y = (int(point[1]) + dy)%w
            index = state["occ"][x,y]
            if index >= 0 and (((point[0]%1)-(state["x"][index,0]%1)-dx)**2 +
                               ((point[1]%1)-(state["x"][index,1]%1)-dy)**2 < 4):
                overlaps.append(index)
    return overlaps

def clear_disk(state,index):
    '''Remove disk from occupation array.'''
    state["occ"][int(state["x"][index,0]),int(state["x"][index,1])] = -1

def add_disk(state,index):
    '''Add disk to occupation array.'''
    state["occ"][int(state["x"][index,0]),int(state["x"][index,1])] = index

def point_reflect(pt,pivot,l):
    '''Reflect the point pt in the pivot (with periodic boundary
    conditions of length l).'''
    pt[:] = (2*pivot - pt)%l

def disk_cluster_move(state,index,pivot):
    '''Iteratively reflect disks in pivot, starting with index,
    until no more overlaps occur.'''
    movers = deque()
    movers.appendleft(index)
    clear_disk(state,index)
    while movers:
        mover = movers.pop()
        point_reflect(state["x"][mover],pivot,len(state["occ"]))
```

(continues on next page)

(continued from previous page)

```
        overlap = get_overlap(state,state["x"][mover])
        for i in overlap:
            movers.appendleft(i)
            clear_disk(state,i)
        add_disk(state,mover)

def random_disk_cluster_move(state):
    '''Perform cluster move starting with random initial disk
    and uniform random pivot.'''
    index = rng.integers(0,len(state["x"]))
    pivot = rng.uniform(0,len(state["occ"]),2)
    disk_cluster_move(state,index,pivot)

side_length = 11
# arbitrary initial configuration
positions = np.array([[ 6.42852835,  9.34497979],
       [ 8.5774671 ,  5.5430735 ],
       [ 3.22720966,  8.0039044 ],
       [ 5.77002192,  4.57665577],
       [ 4.21675366,  2.92234618],
       [ 8.89704476,  2.66159969],
       [ 6.64404186,  7.03428799],
       [ 4.16896717,  6.16114986],
       [ 1.57600143,  6.5405559 ],
       [10.315628  ,  6.53294302],
       [ 0.21481807,  8.65080737],
       [ 7.04888305,  0.32190093],
       [10.93687882,  4.62709832],
       [ 2.3743327 ,  4.35629547]])

# describe the disk model state by the positions and occupation array
state = {"x": positions, "occ": occupation_array(side_length,positions)}

for _ in range(100):
    random_disk_cluster_move(state)

fig, axs = plt.subplots(2,5,figsize=(16,6))
previous_pos = np.copy(state["x"])
for axr in axs:
    for ax in axr:
        random_disk_cluster_move(state)
        color = (state["x"] == previous_pos).all(axis=1)
        plot_disk_configuration(state["x"],side_length,ax,color)
        previous_pos = np.copy(state["x"])
plt.show()
```



6.5 Further reading

On the Ising model phase transition, critical slowing down, and cluster algorithms, see sections 3.7.1 - 4.3 of [NB99].

A gentle introduction to various cluster algorithms, including the Geometric cluster algorithm, can be found in [Lui06].

Chapter 7

Lattice field theory

Last week we have seen that close to a continuous phase transition the correlation length of a discrete statistical system diverges, like in the Ising model. In this regime the physics of the model takes place on scales that are much larger than the individual lattice sites, so one is approaching physics in the continuum. Except when we are trying to model a realistic lattice, say of a crystalline material, the lattice is usually an artifact of trying to represent continuous physics in the necessarily discrete framework in which a computer can operate. Understanding the **continuum limit** of lattice models in terms of their critical properties is therefore of central importance in many applications of Monte Carlo techniques in physics. This week we will concentrate on applications to **Lattice Field Theory**. This is a huge and very active topic of which we will only be able to scratch the surface. The basic goal is simple: given a Quantum Field Theory (QFT) in the continuum, construct a **lattice discretization** that is amenable to Markov Chain Monte Carlo simulation and such that it possesses a continuum limit that agrees with the original QFT. If successful this procedure is, in principle, of great value to Quantum Field Theory for several reasons:

1. Discretization naturally imposes an **ultraviolet cut-off** on the QFT, meaning that the infinities that necessarily appear in continuum calculations are automatically regularized.
2. Upon discretization the degrees of freedom in the fields are reduced from (uncountably) infinite to only a finite number (one or several values per lattice site). This allows one to do calculations beyond the perturbation theory that is usually employed in the continuum, i.e. lattice discretization provides access to **non-perturbative phenomena** in field theory (like confinement of quarks in quantum chromodynamics).
3. The **range of observables** that can be computed in Monte Carlo simulations is different from the range that is accessible via continuum methods, because they have different limitations. The latter are limited by the analytical and mathematical methods available to perform the computations, while in computer simulations there is a lot of freedom in measuring quantities and the main limitations stem from having limited statistics/lattice sizes and artifacts of discretization.

This sounds all good, but we will see that there are some serious constraints on the QFT in question and hurdles in selecting appropriate discretizations.

7.1 Path Integrals in Quantum Mechanics

Before diving into field theory, we focus on the quantum mechanics of a single particle. Let us consider a point particle (with mass $m = 1$) in a one-dimensional potential $V(x)$ with Hamiltonian

$$H = \frac{1}{2}p^2 + V(x).$$

There are two equivalent ways to construct the quantum mechanics of this system:

1. **Canonical quantization**: promoting the position, momentum and Hamiltonian to operators on a Hilbert space satisfying $[\hat{x}, \hat{p}] = i\hbar$.

2. **Path integral quantization**: formally integrating over all possible paths $x(t)$ with a contribution that is proportional to $e^{iS[x(t)]/\hbar}$, where $S[x(t)]$ is the action given by the time integral of the Lagrangian associated to H.

Let us have a look at the relation between the two. For this it is most convenient to work in the Heisenberg picture of canonical quantization in which states are time independent and operators $\hat{x}(t)$ evolve with time as

$$\hat{x}(t) = e^{i\hat{H}t/\hbar}\hat{x}(0)e^{-i\hat{H}t/\hbar}.$$

Denoting the eigenstates of $\hat{x}(t)$ by $|x,t\rangle$, i.e. $\hat{x}(t)|x,t\rangle = x|x,t\rangle$, we then also have $|x,t\rangle = e^{i\hat{H}t/\hbar}|x,0\rangle \equiv e^{i\hat{H}t/\hbar}|x\rangle$. The **transition amplitude** for the particle to move from initial position x_i at time t_i to the final position x_f at time t_f is then given by

$$\langle x_f, t_f | x_i, t_i\rangle = \langle x_f | e^{-i\hat{H}(t_f - t_i)/\hbar}|x_i\rangle.$$

Inserting the identity operator $\int \mathrm{d}x|x,t_j\rangle\langle x,t_j|$ at $N-1$ equally spaced intermediate times $t_1,\dots,t_{N-1}$ and letting $t_0 = t_i$, $x_0 = x_i$, $t_N = t_f$, $x_N = x_f$, we find

$$\begin{aligned}\langle x_f, t_f | x_i, t_i\rangle &= \int \mathrm{d}x_1 \cdots \mathrm{d}x_{N-1} \prod_{j=1}^{N} \langle x_j, t_j | x_{j-1}, t_{j-1}\rangle \\ &= \int \mathrm{d}x_1 \cdots \mathrm{d}x_{N-1} \prod_{j=1}^{N} \langle x_j | e^{-i\hat{H}\delta/\hbar}|x_{j-1}\rangle,\end{aligned}$$

where $\delta = (t_f - t_i)/N$. We are interested in the $N\to\infty$ limit, so we compute to leading order in δ that

$$\begin{aligned}\langle x_j | e^{-i\hat{H}\delta/\hbar}|x_{j-1}\rangle &= \langle x_j | e^{-iV(\hat{x})\delta/\hbar} e^{-i\hat{p}^2\delta/(2\hbar)}|x_{j-1}\rangle + O(\delta^2) \\ &= e^{-iV(x_j)\delta/\hbar}\langle x_j | e^{-i\hat{p}^2\delta/(2\hbar)}|x_{j-1}\rangle + O(\delta^2).\end{aligned}$$

Inserting the identity operator $\hat{I} = \int \frac{\mathrm{d}p}{2\pi}|p\rangle\langle p|$ and using the recognizing the plane wave $\langle p|x\rangle = e^{ipx/\hbar}/\sqrt{\hbar}$, this becomes

$$\begin{aligned}\langle x_j | e^{-i\hat{H}\delta/\hbar}|x_{j-1}\rangle &= e^{-iV(x_j)\delta/\hbar}\int \frac{\mathrm{d}p}{2\pi}\langle x_j | e^{-i\hat{p}^2\delta/(2\hbar)}|p\rangle\langle p|x_{j-1}\rangle + O(\delta^2) \\ &= e^{-iV(x_j)\delta/\hbar}\int \frac{\mathrm{d}p}{2\pi} e^{-ip^2\delta/(2\hbar)}\langle x_j|p\rangle\langle p|x_{j-1}\rangle + O(\delta^2) \\ &= e^{-iV(x_j)\delta/\hbar}\int \frac{\mathrm{d}p}{2\pi\hbar} e^{-ip^2\delta/(2\hbar) - ip(x_j - x_{j-1})/\hbar} + O(\delta^2) \\ &= C\exp\left[\frac{i}{\hbar}\delta\left(\frac{1}{2}\left(\frac{x_j - x_{j-1}}{\delta}\right)^2 - V(x_j)\right)\right] + O(\delta^2),\end{aligned}$$

where C is an unimportant (δ-dependent) constant. Plugging this into the transition amplitude we find

$$\begin{aligned}\langle x_f, t_f | x_i, t_i\rangle &= C^N \int \mathrm{d}x_1 \cdots \mathrm{d}x_{N-1} \exp\left[\frac{i}{\hbar}\delta\sum_{j=1}^{N}\left(\frac{1}{2}\left(\frac{x_j - x_{j-1}}{\delta}\right)^2 - V(x_j)\right)\right] \\ &\xrightarrow{N\to\infty} \int_{\substack{x(t_i)=x_i \\ x(t_f)=x_f}} [\mathcal{D}x(t)]\, e^{\frac{i}{\hbar}S[x(t)]},\end{aligned}$$

where $S[x(t)]$ is the action

$$S[x(t)] = \int_{t_i}^{t_f} \mathrm{d}t\, L(x(t), \dot{x}(t)), \qquad L(x,\dot{x}) = \frac{1}{2}\dot{x}^2 - V(x).$$

Here $L(x,\dot{x})$ is the (classical) Lagrangian, which is related to the (classical) Hamiltonian H by the usual Legendre transform. Note that $[\mathcal{D}x(t)]$ is a formal functional integration measure on the space of all functions $t\to x(t)$ on $[t_i, t_f]$ with the fixed boundary conditions $x(t_i) = x_i$, $x(t_f) = x_f$.

The path-integral formulation has the advantage over canonical quantization, that the variables involved, i.e. the positions $x(t)$ at different times t, are regular numbers instead of non-commutative operators. However, we are left with a functional integral that is highly oscillatory and requires a fragile destructive interference of the phases $e^{\frac{i}{\hbar}S}$ to yield a sensible result.

7.1.1 Correlation functions

The transition amplitude as function of x_i, x_f and t_i, t_f contains all information about the quantum system and is accessible in both frameworks. However, since they play the role as boundary conditions in the path integral, it is not necessarily easy to vary them in practice. Luckily the same information as in the transition amplitude is also available in the collection of **correlation functions** without varying the initial and final state. In canonical quantization a correlation function generally is an expectation value with respect to a fixed state, for instance the ground state $|0\rangle$ of the Hamiltonian, of an operator constructed from $\hat{x}(t)$ at different times, e.g. the **two-point correlation function** $\langle 0|\hat{x}(t_2)\hat{x}(t_1)|0\rangle$ for $t_2 > t_1$. Given a complete set of eigenstates $|\phi_n\rangle$ with increasing energies E_n, $H|\phi\rangle = E_n|\phi_n\rangle$, and assuming that $|\phi_0\rangle = |0\rangle$ has zero energy $E_0 = 0$, this is given by

$$\begin{aligned}\langle 0|\hat{x}(t_2)\hat{x}(t_1)|0\rangle &= \langle 0|e^{iHt_2/\hbar}\hat{x}(0)e^{-iH(t_2-t_1)/\hbar}\hat{x}(0)e^{-iHt_1/\hbar}|0\rangle \\ &= \sum_n \langle 0|\hat{x}(0)e^{-iH(t_2-t_1)/\hbar}|\phi_n\rangle\langle\phi_n|\hat{x}(0)|0\rangle \\ &= \sum_n |\langle 0|\hat{x}(0)|\phi_n\rangle|^2 e^{-iE_n(t_2-t_1)/\hbar}.\end{aligned} \tag{7.1}$$

So we see, for instance, that the different energy eigenstates can be extracted from Fourier analysis of this correlation function. Correlation functions are particularly natural objects in the path integral as they are computed by simply "putting the observable in the integrand", e.g.

$$C(t_2,t_1) = \frac{\langle x_f,t_f|\hat{x}(t_2)\hat{x}(t_1)|x_i,t_i\rangle}{\langle x_f,t_f|x_i,t_i\rangle} = \frac{1}{\mathcal{Z}}\int_{\substack{x(t_i)=x_i\\x(t_f)=x_f}}[\mathcal{D}x]\, x(t_2)x(t_1)e^{\frac{i}{\hbar}S[x(t)]},$$

$$\mathcal{Z} := \int_{\substack{x(t_i)=x_i\\x(t_f)=x_f}}[\mathcal{D}x]e^{\frac{i}{\hbar}S[x(t)]}.$$

Note this is not precisely the same correlation as $\langle 0|\hat{x}(t_2)\hat{x}(t_1)|0\rangle$, but if we wish we could obtain the latter by integrating over the boundary conditions. Point is that being able to compute arbitrary correlation functions gives sufficient information, without the need to change boundary conditions.

7.1.2 Wick rotation

The expression for the correlation $C(t_2,t_1)$ in the path integral formulation looks very much like the expectation value of the observable $x(t_2)x(t_1)$ for a random function $x(t)$ with partition function $\mathcal{Z}$, except that we have a complex phase instead of a real and positive Boltzmann weight. To really be able to make a probabilistic interpretation, we need to somehow make $e^{\frac{i}{\hbar}S[x(t)]}$ real. Equation (7.1) already gives a hint on how to do this: we should analytically continue the time to become purely imaginary, i.e. set $t = -i\tau$, such that $\langle 0|\hat{x}(\tau)\hat{x}(0)|0\rangle = \sum_n |\langle 0|\hat{x}(0)|\phi_n\rangle|^2 e^{-E_n\tau/\hbar}$. This **Wick rotation** can also be done (at least formally) in the path integral,

$$S = \int \mathrm{d}t\left(\frac{1}{2}\dot{x}(t)^2 - V(x(t))\right) \longrightarrow iS_E = i\int \mathrm{d}\tau\left(\frac{1}{2}\dot{x}(\tau)^2 + V(x(\tau))\right),$$

$$\mathcal{Z} = \int[\mathcal{D}x]e^{\frac{i}{\hbar}S[x(t)]} \longrightarrow Z = \int[\mathcal{D}x]e^{-\frac{1}{\hbar}S_E[x(\tau)]}.$$

If you trust this analytic continuation, then we are in a much better place: the integrand $e^{-S_E[x(\tau)]/\hbar}$ is real and positive so we can really interpret the imaginary-time path integral Z as a partition function for random paths $x(\tau)$. What is more, the imaginary-time action $S_E[x(\tau)]$ is bounded from below (if the potential is), so when $\hbar \to 0$ the partition function is peaked at the minimum of the action, which we can interpret as the classical solution of the particle in imaginary time. The action $S_E[x(\tau)]$ will be called the **Euclidean action** for reasons that should become clear later in the field theory context. But why should we trust this operation? Well, under suitable assumptions and boundary conditions, the correlation functions of the real-time and imaginary-time path integrals can be shown to be analytic continuations of each other. This means that if we know one, it can be used to compute the other.

7.1.3 Example: Simple harmonic oscillator in imaginary time

Let us consider the example of a simple harmonic oscillator with potential $V(x) = \frac{1}{2}\omega^2 x^2$. Instead of choosing initial and final boundary conditions in (imaginary) time, it is convenient to choose the time coordinate τ to be periodic with some period T. We therefore obtain the path integral

$$Z = \int_{x(T)=x(0)} [\mathcal{D}x(\tau)] e^{-\frac{1}{\hbar}S_E[x(\tau)]},$$
$$S_E[x(\tau)] = \int_0^T \mathrm{d}\tau \left(\frac{1}{2}\dot{x}(\tau)^2 + \frac{1}{2}\omega^2 x(\tau)^2\right).$$

Its two-point correlation function can be related to the canonical quantization via

$$\begin{aligned}\langle x(s)x(0)\rangle &:= \frac{1}{Z}\int [\mathcal{D}x(\tau)] x(s)x(0) e^{-\frac{1}{\hbar}S_E[x(\tau)]} \\ &= \frac{\mathrm{Tr}[\hat{x}(s)\hat{x}(0)e^{-TH/\hbar}]}{\mathrm{Tr}[e^{-TH/\hbar}]} \\ &= \frac{\sum_n \langle\phi_n|\hat{x}(s)\hat{x}(0)|\phi_n\rangle e^{-TE_n/\hbar}}{\sum_n e^{-TE_n/\hbar}}.\end{aligned}$$

If $T(E_1 - E_0)/\hbar \gg 1$ and $s \ll T$ then this is proportional to $e^{-(E_1-E_0)s}$, so the two-point function $\langle x(s)x(0)\rangle$ should show an exponential decay for small s with rate given by the excitation energy of the simple harmonic oscillator.

In order to simulate this partition function, we need to perform a **lattice discretization**, in a way that is essentially the reverse of what we did above when relating the canonical to path-integral quantization. We only record the position $x_1, x_2, \ldots, x_N = x_0$ on the N sites of a one-dimensional lattice in the time direction and choose a **lattice action** in which the time derivatives are replaced by discrete counterparts,

$$S_E[x] \approx a\sum_{j=1}^{N}\left(\frac{1}{2}\left(\frac{x_j - x_{j-1}}{a}\right)^2 + \frac{\omega^2}{2}x_j^2\right),$$

where $a = T/N$ is the **lattice spacing**. We then introduce **dimensionless** parameters $\hat{x}_i = x_i/a, \hat{\omega} = \omega a, \beta = a/\hbar$, leading to the Boltzmann distribution in terms of a dimensionless action

$$\pi(\hat{x}) = \frac{1}{Z}e^{-\beta S_L}, \qquad S_L[\hat{x}] = \frac{1}{2}\sum_{j=1}^{N}\left(\left(\hat{x}_j - \hat{x}_{j-1}\right)^2 + \hat{\omega}^2\hat{x}_j^2\right).$$

This is our desired distribution, in which the analogy with a statistical partition function has been made explicit through the introduction of a fiducial inverse temperature $\beta = a/\hbar$. Now we are in business, because we know how to sample from this distribution using **Metropolis-Hastings**!

Note that in this case β does not play any role, as it can be absorbed in a rescaling $\hat{x}_j \to \hat{x}'_j = \hat{x}_j\sqrt{\beta}$, so we can safely set $\beta = 1$. As proposal transition we select a uniform site t and take $\hat{y}_t = \hat{x}_t + U$ and $\hat{y}_j = \hat{x}_j$ for $j \neq t$, where U is uniform on $(-\delta, \delta)$. The result is then accepted with probability

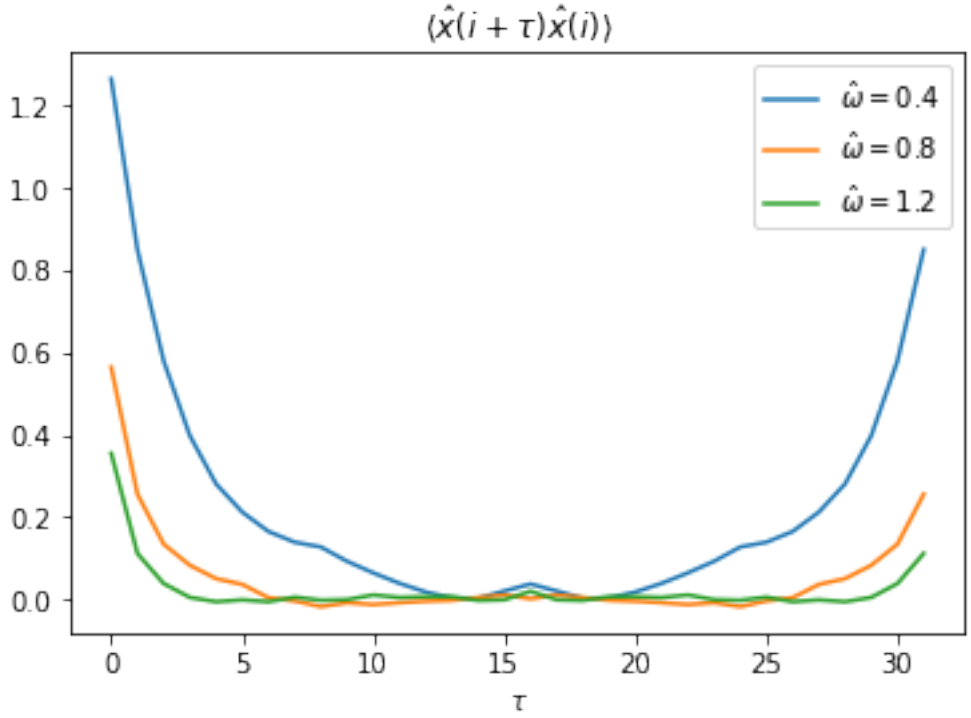
$$\begin{aligned}A(\hat{x}\to\hat{y}) &= \min(1, e^{-\Delta S}), \\ \Delta S &= S_L[\hat{y}] - S_L[\hat{x}] \\ &= (\hat{x}_t - \hat{y}_t)(\hat{x}_{t-1} + \hat{x}_{t+1} - (1 + \tfrac{1}{2}\hat{\omega}^2)(\hat{x}_t + \hat{y}_t)).\end{aligned}$$

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def lattice_action(x,omega_sq):
    '''Compute lattice action S_L[x] on state x.'''
    return 0.5 * ( np.dot(x - np.roll(x,1),x-np.roll(x,1)) + omega_sq*np.dot(x,x)␣
↪)
```

(continues on next page)

(continued from previous page)

```
def lattice_action_diff(x,t,newx,omega_sq):
    '''Compute change in action when x[t] is replaced by newx.'''
    w = len(x)
    return (x[t]-newx)*(x[(t-1)%w]+x[(t+1)%w]-(1+omega_sq/2)*(newx+x[t]))

def oscillator_MH_step(x,omega_sq,delta):
    '''Perform Metropolis-Hastings update on uniform site with range delta.'''
    t = rng.integers(0,len(x))
    newx = x[t] + rng.uniform(-delta,delta)
    deltaS = lattice_action_diff(x,t,newx,omega_sq)
    if deltaS < 0 or rng.uniform() < np.exp(-deltaS):
        x[t] = newx
        return True
    return False

def run_oscillator_MH(x,omega_sq,delta,n):
    '''Perform n Metropolis-Hastings moves on state x.'''
    total_accept = 0
    for _ in range(n):
        if oscillator_MH_step(x,omega_sq,delta):
            total_accept += 1
    return total_accept

def two_point_function(states):
    '''Estimate two-point correlation <x(s+t)x(s)> from states (averaging over s).
 ↪'''
    return np.array([np.mean(states * np.roll(states,t,axis=1)) for t in␣
 ↪range(len(states[0]))])

time_length = 32
omegas = [0.4,0.8,1.2]
delta = 1.8  # chosen to have ~ 50% acceptance
equil_sweeps = 1000
measure_sweeps = 10
measurements = 100

two_point_funs = []
for omega in omegas:
    omega_sq = omega**2
    x_state = np.zeros(time_length)
    run_oscillator_MH(x_state,omega_sq,delta,equil_sweeps * time_length)
    states = np.empty((measurements,time_length))
    for s in states:
        run_oscillator_MH(x_state,omega_sq,delta,measure_sweeps * time_length)
        s[:] = x_state
    two_point_funs.append(two_point_function(states))

plt.plot(np.transpose(np.array(two_point_funs)))
plt.legend([r"$\hat\omega = {}$".format(o) for o in omegas])
plt.title(r"$\langle \hat{x}(i + \tau)\hat{x}(i)\rangle$")
plt.xlabel(r"$\tau$")
plt.show()
```



Indeed we observe an exponential decay $\langle x(s)x(0)\rangle \sim e^{-(E_1-E_0)s}$ for small s, so by fitting we could estimate the excitation energy from the data. Note that the continuum limit should be sought when $N \to \infty$, i.e. the lattice spacing $a = T/N \to 0$ and $\hat{\omega} = \omega a \to 0$. This is in accordance with our claim last week that we should approach criticality, in this case $\hat{\omega} \to 0$, where the dimensionless correlation length diverges.

7.1.4 Heatbath algorithm

Let us take the opportunity to discuss an alternative to (or rather a special case of) the Metropolis-Hastings algorithm that sometimes works better. Like in the Metropolis-Hastings algorithm we described above only a single site is updated, but in the **heatbath algorithm** this is done in a rejection-free fashion by locally thermalizing a randomly chosen site within its neighborhood. In other words, we select a uniform site t and sample a new value $\hat{y}_t$ with density proportional to $\pi(\hat{x}_1, \dots, \hat{x}_{t-1}, \hat{y}_t, \hat{x}_{t+1}, \dots, \hat{x}_N)$. The other values are left unchanged, i.e. $\hat{y}_i = \hat{x}_i, i \neq t$. It is easily checked that the resulting transition density $p(\hat{x}, \hat{y})$ satisfies detailed balance and is irreducible.

In the case of the harmonic oscillator we should thus sample $\hat{y}_t$ with density proportional to

$$\exp\left[(\hat{x}_{t-1} + \hat{x}_{t+1})\hat{y}_t - (1 + \tfrac{1}{2}\hat{\omega}^2)\hat{y}_t^2\right],$$

but this is nothing but a normal random variable with mean μ and variance σ^2 given by

$$\mu = \frac{\hat{x}_{t-1} + \hat{x}_{t+1}}{2 + \hat{\omega}^2}, \qquad \sigma^2 = \frac{1}{2 + \hat{\omega}^2}.$$

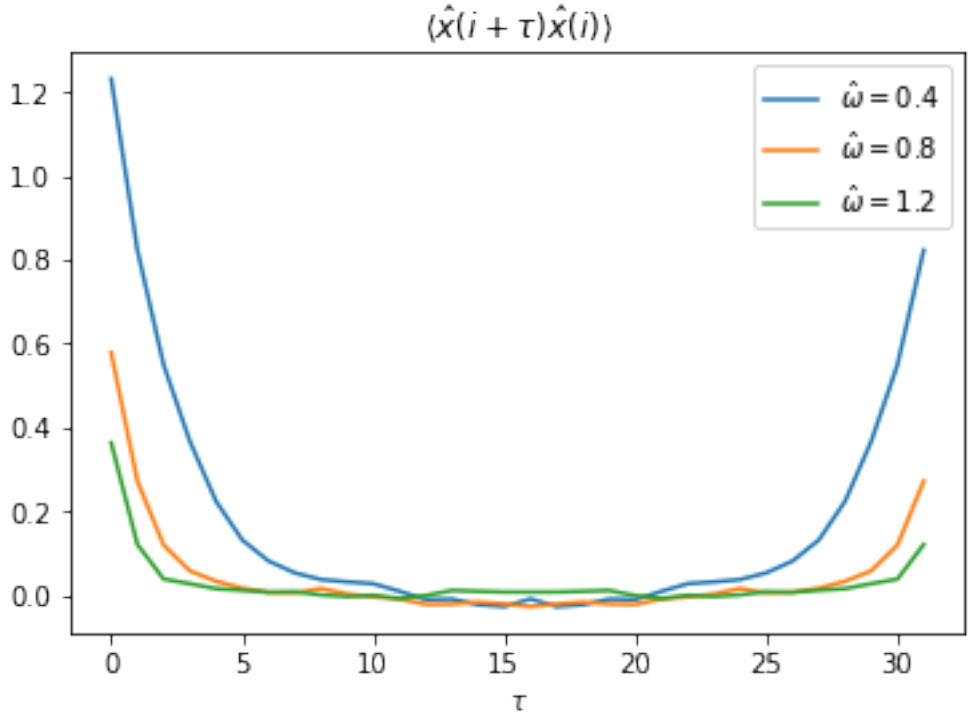
Let us check that this algorithm yields the same result as before.

```
def oscillator_heatbath_step(x,sigma):
    '''Perform heatbath update on state x.'''
    w = len(x)
    t = rng.integers(0,w)
    x[t] = rng.normal(sigma*sigma*(x[(t-1)%w] + x[(t+1)%w]),sigma)

def run_oscillator_heatbath(x,omega_sq,n):
    '''Perform n heatbath updates.'''
    sigma = 1/np.sqrt(2+omega_sq)
```

(continues on next page)

(continued from previous page)

```
    for _ in range(n):
        oscillator_heatbath_step(x,sigma)

two_point_funs = []
for omega in omegas:
    omega_sq = omega**2
    x_state = np.zeros(time_length)
    run_oscillator_heatbath(x_state,omega_sq,equil_sweeps * time_length)
    states = np.empty((measurements,time_length))
    for s in states:
        run_oscillator_heatbath(x_state,omega_sq,measure_sweeps * time_length)
        s[:] = x_state
    two_point_funs.append(two_point_function(states))

plt.plot(np.transpose(np.array(two_point_funs)))
plt.legend([r"$\hat\omega = {}$".format(o) for o in omegas])
plt.title(r"$\langle \hat{x}(i + \tau)\hat{x}(i)\rangle$")
plt.xlabel(r"$\tau$")
plt.show()
```



7.2 Scalar field theory

The step from the path integral of a quantum mechanical particle to the path integral of a scalar field theory is a small one. Recall that the former amounts to an integration over all trajectories $x(\tau)$ in imaginary time τ. We can really interpret $x(\tau)$ as a scalar field in 0 spatial and 1 temporal direction. So the only thing we need to do is to make the field $\phi(x)$ depend on extra spatial dimensions and replace the action by the appropriate action of the field theory we are interested in.

Let us concentrate on the case of a real scalar field $\phi(x^\mu)$ on 4-dimensional Minkowski space with mass m_0 and

quartic self-coupling λ_0. The corresponding action is

$$\begin{aligned} S[\phi] &= \int \mathrm{d}^4x \left[\frac{1}{2}\eta^{\mu\nu}\partial_\mu\phi\partial_\nu\phi - \frac{1}{2}m_0^2\phi^2 - \frac{1}{4!}\lambda_0\phi^4\right] \\ &= \int \mathrm{d}t\mathrm{d}^3x \left[\frac{1}{2}(\partial_t\phi)^2 - \frac{1}{2}\delta^{ij}\partial_i\phi\partial_j\phi - \frac{1}{2}m_0^2\phi^2 - \frac{1}{4!}\lambda_0\phi^4\right]. \end{aligned}$$

As in the quantum-mechanical example, we need to perform a **Wick rotation** to imaginary time $t = -i\tau$, replacing the phase $e^{\frac{i}{\hbar}S}$ by the real weight $e^{-\frac{1}{\hbar}S_E}$, where

$$\begin{aligned} S_E[\phi] &= \int \mathrm{d}\tau\mathrm{d}^3x \left[\frac{1}{2}(\partial_\tau\phi)^2 + \frac{1}{2}\delta^{ij}\partial_i\phi\partial_j\phi + \frac{1}{2}m_0^2\phi^2 + \frac{1}{4!}\lambda_0\phi^4\right] \\ &= \int \mathrm{d}^4x \left[\frac{1}{2}\delta^{\mu\nu}\partial_\mu\phi\partial_\nu\phi + \frac{1}{2}m_0^2\phi^2 + \frac{1}{4!}\lambda_0\phi^4\right]. \end{aligned}$$

It is called the **Euclidean action**, because it has exactly the form of a scalar field in 4-dimensional Euclidean space with metric $\delta_{\mu\nu}$. Note that it is bounded below if $\lambda_0 > 0$, so we can be optimistic about assigning a probabilistic interpretation to $e^{-\frac{1}{\hbar}S_E}$.

But does any physical interpretation remain after this Wick rotation? It turns out that under certain conditions, the correlation functions in real time (known as **Wightman distributions**) and imaginary time (known as **Schwinger functions**) are analytic continuations of each other, just like in the quantum-mechanical case above. On the Euclidean field theory side the conditions are know as the Osterwalder-Schrader axioms, including the important property of **reflection positivity**. Once this is known, the correlation functions of the Euclidean field theory provide access to the spectrum of the real-time Hamiltonian, from which masses of particles can for instance be extracted.

To turn our Euclidean scalar field theory into a lattice field theory, we select a lattice spacing a and put the scalar field on a $w \times w \times w \times w$ grid $\Lambda = \{a, 2a, \dots, wa\}^4$ with periodic boundary conditions and select a discretized action, e.g.

$$S_L[\phi] = a^4 \sum_{x\in\Lambda} \left[\frac{1}{2}m_0^2\phi^2(x) + \frac{1}{4!}\lambda_0\phi^4(x) + \sum_{\hat{\mu}} \frac{1}{2}\left(\frac{\phi(x+\hat{\mu}a) - \phi(x)}{a}\right)^2\right],$$

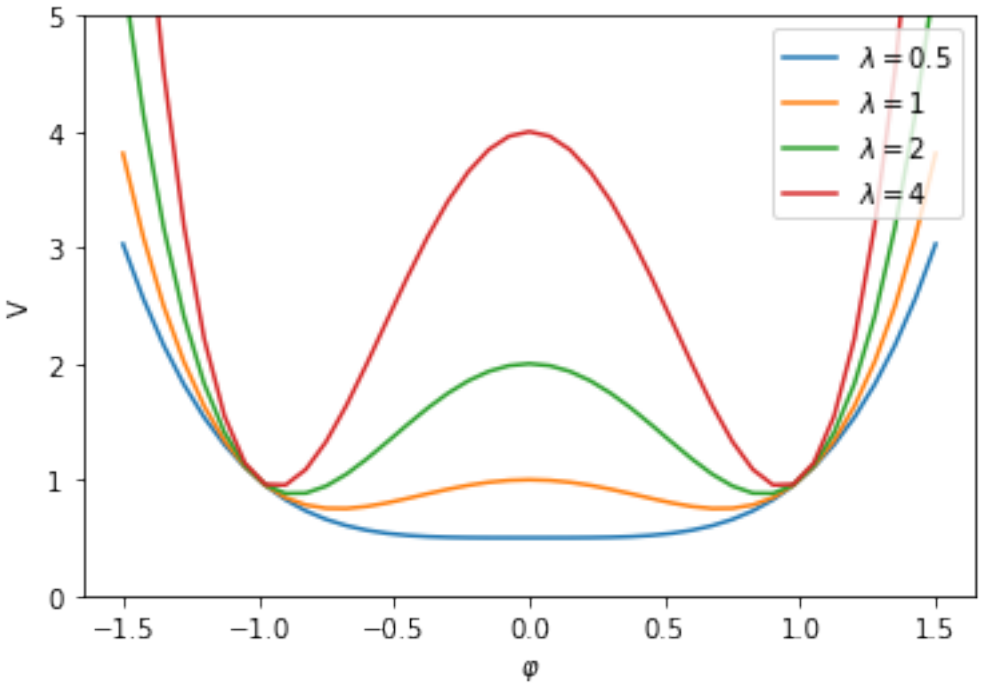
where the second sum is over the 4 directions $\hat{\mu} = (1,0,0,0), \dots, (0,0,0,1)$. We can make the couplings and fields dimensionless via the identifications

$$\phi(ax) = \frac{\sqrt{2\kappa}}{a}\varphi(x), \quad a^2m_0^2 = \frac{1-2\lambda}{\kappa} - 8, \quad \lambda_0 = \frac{6\lambda}{\kappa^2}.$$

Up to an irrelevant constant shift by λw^4 this leads to the standard form

$$S_L[\varphi] = \sum_{x\in\Lambda} \left[V(\varphi(x)) - 2\kappa\sum_{\hat{\mu}} \varphi(x)\varphi(x+\hat{\mu})\right], \qquad V(\varphi) = \lambda(\varphi^2-1)^2 + \varphi^2.$$

```
def potential_v(x,lamb):
    '''Compute the potential function V(x).'''
    return lamb*(x*x-1)*(x*x-1)+x*x

xrange = np.linspace(-1.5,1.5,41)
lambdas = [0.5,1,2,4]
for l in lambdas:
    plt.plot(xrange,potential_v(xrange,l))
plt.ylim(0,5)
plt.legend([r"$\lambda = {}$".format(l) for l in lambdas])
plt.xlabel(r"$\varphi$")
plt.ylabel("V")
plt.show()
```



The advantage of this form is that we can easily understand the $\lambda \to \infty$ limit of the lattice field theory. In this limit the potential $V(\varphi)$ will confine $\varphi(x)$ to take values ± 1, turning the model exactly into the Ising model on the grid with parameter $\beta J = \kappa$. It is thus reasonable to expect that this interacting scalar field theory lives in the same **universality class** as the 4d Ising model, in the sense that it possesses the same continuum limit at criticality (although this continuum limit is known to be rather trivial in the 4d case [ADC21]).

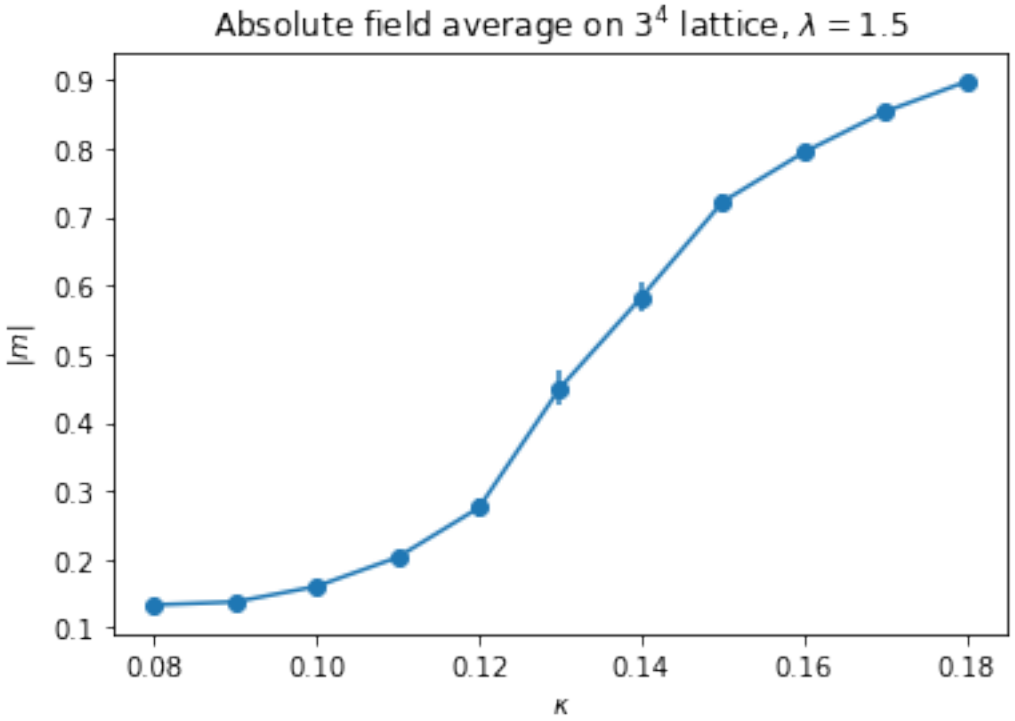
Let us have a look at the field average $m = w^{-4} \sum_{x \in \Lambda} \varphi(x)$, which is the analogue of the magnetization per spin in the Ising model. The following code measures the absolute field average $|m|$ for $\lambda = 1.5$ and $w = 3$ and several values of κ using Metropolis-Hastings. Needless to say, this is a tiny lattice and the statistics are pretty bad. Optimization and patience are required to handle larger lattices, but the spontaneous symmetry-breaking transition is already clearly visible.

```
def potential_v(x,lamb):
    '''Compute the potential function V(x).'''
    return lamb*(x*x-1)*(x*x-1)+x*x

def neighbor_sum(phi,s):
    '''Compute the sum of the state phi on all 8 neighbors of the site s.'''
    w = len(phi)
    return (phi[(s[0]+1)%w,s[1],s[2],s[3]] + phi[(s[0]-1)%w,s[1],s[2],s[3]] +
            phi[s[0],(s[1]+1)%w,s[2],s[3]] + phi[s[0],(s[1]-1)%w,s[2],s[3]] +
            phi[s[0],s[1],(s[2]+1)%w,s[3]] + phi[s[0],s[1],(s[2]-1)%w,s[3]] +
            phi[s[0],s[1],s[2],(s[3]+1)%w] + phi[s[0],s[1],s[2],(s[3]-1)%w] )

def scalar_action_diff(phi,site,newphi,lamb,kappa):
    '''Compute the change in the action when phi[site] is changed to newphi.'''
    return (2 * kappa * neighbor_sum(phi,site) * (phi[site] - newphi) +
            potential_v(newphi,lamb) - potential_v(phi[site],lamb) )

def scalar_MH_step(phi,lamb,kappa,delta):
    '''Perform Metropolis-Hastings update on state phi with range delta.'''
    site = tuple(rng.integers(0,len(phi),4))
    newphi = phi[site] + rng.uniform(-delta,delta)
    deltaS = scalar_action_diff(phi,site,newphi,lamb,kappa)
    if deltaS < 0 or rng.uniform() < np.exp(-deltaS):
        phi[site] = newphi
        return True
```

(continues on next page)

(continued from previous page)

```
    return False

def run_scalar_MH(phi,lamb,kappa,delta,n):
    '''Perform n Metropolis-Hastings updates on state phi and return number of␣
 ↪accepted transtions.'''
    total_accept = 0
    for _ in range(n):
        total_accept += scalar_MH_step(phi,lamb,kappa,delta)
    return total_accept

def batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each.
    Returns the mean and standard error.'''
    batches = np.reshape(data,(k,-1))
    values = np.apply_along_axis(observable, 1, batches)
    return np.mean(values), np.std(values)/np.sqrt(k-1)

lamb = 1.5
kappas = np.linspace(0.08,0.18,11)
width = 3
num_sites = width**4
delta = 1.5  # chosen to have ~ 50% acceptance
equil_sweeps = 1000
measure_sweeps = 2
measurements = 2000

mean_magn = []
for kappa in kappas:
    phi_state = np.zeros((width,width,width,width))
    run_scalar_MH(phi_state,lamb,kappa,delta,equil_sweeps * num_sites)
    magnetizations = np.empty(measurements)
    for i in range(measurements):
        run_scalar_MH(phi_state,lamb,kappa,delta,measure_sweeps * num_sites)
        magnetizations[i] = np.mean(phi_state)
    mean, err = batch_estimate(np.abs(magnetizations),lambda x:np.mean(x),10)
    mean_magn.append([mean,err])

plt.errorbar(kappas,[m[0] for m in mean_magn],yerr=[m[1] for m in mean_magn],fmt=
 ↪'-o')
plt.xlabel(r"$\kappa$")
plt.ylabel(r"$|m|$")
plt.title(r"Absolute field average on $3^4$ lattice, $\lambda = 1.5$")
plt.show()
```



7.3 Further reading

A thorough text book on Lattice Field Theory including an explanation on path integral approach (Chapter 2) and scalar field (Chapter 3) is [Rot12].

For a shorter introduction see Chapter 6 of [Jos20] or the lecture notes [Mor07].

Other good books on the topic are [Cre85], [Smi02] and [MM94].

Chapter 8

Quantum Gravity

Last week we have seen the first application of Monte Carlo techniques to Quantum Field Theory (QFT). In the case of scalar fields in d-dimensional Minkowski space (we only treated the case $d = 4$ last week, but let us be a bit more general now), this involved several steps:

Wick rotation First we had to shift our attention to the corresponding Euclidean Quantum Field Theory (EQFT) of scalar fields on d-dimensional Euclidean space, formally obtained by a Wick rotation in which the time coordinate t of Minkowski space is analytically continued to an imaginary time τ. The path integral associated to the EQFT took the form

$$Z = \int [\mathcal{D}\phi] e^{-\frac{1}{\hbar} S_E[\phi]},$$

where the formal functional integral is over all field configurations ϕ on d-dimensional Euclidean space. We observed that the Euclidean action $S_E[\phi]$ is real and bounded below, which suggests that one may be able to interpret the path integral as a statistical partition function in which each field configuration ϕ appears with probability proportional to $e^{-\frac{1}{\hbar} S_E[\phi]}$.

Lattice discretization To make sense of such a probability distribution and to have hopes of simulating the statistical system, the field theory is turned into a Lattice Field Theory (LFT) by discretizing the continuous scalar field $\phi : \mathbb{R}^d \to \mathbb{R}$ into a field $\varphi : \Lambda \to \mathbb{R}$ on a d-dimensional lattice $\Lambda \subset \mathbb{R}^d$.

Statistical interpretation This way one obtains a lattice partition function

$$Z_\Lambda = \int \Big[\prod_{x \in \Lambda} \mathrm{d}\varphi(x) \Big] e^{-S_L[\varphi]}.$$

Assuming that the finite-dimensional integral converges, which it does in the case of the potential considered last week, it literally defines a probability density $e^{-S_L[\varphi]}/Z$ on field configurations (i.e. on $\mathbb{R}^{|\Lambda|}$) and therefore a well-defined statistical system.

MCMC Then one can use the Markov Chain Monte Carlo techniques that we have developed throughout the course to sample from this distribution and measure expectation values of field observables.

Continuum limit To approach the original EQFT one has to investigate the continuum limit of the LFT, which involves increasing the lattice size while decreasing the physical lattice spacing. We have argued that the latter is only possible if the lattice field theory approaches criticality, e.g. close to a continuous phase transition.

It is worth emphasizing that the first two steps can in general not be made mathematically precise, because they operate at the level of the purely formal infinite-dimensional path integral (from QFT courses we only know how to treat free fields and perturbation theory). Instead those steps should be viewed as guiding principles to motivate the introduction of particular lattice field theories. The latter have a mathematically well-defined basis, as they simply amount to probability distributions on finite-dimensional spaces. Hence, the really non-trivial part resides in the last point: the continuum limit. As the size w of the lattice increases and the lattice spacing $a \sim 1/w$ decreases, does there exist a choice of couplings as function of w such that the important observables, like (properly normalized) correlation functions of the fields, converge as $w \to \infty$? If we could mathematically prove that this is the case, we

could take the limit as our definition of the EQFT! For most field theories of physical interest such a proof is still out of reach (solving it in the case of Yang-Mills theory will make you a millionaire), but that should not stop us from taking the numerical approach. Monte Carlo simulations can provide numerical evidence for the existence of a continuum limit as well as estimates of observables in this limit by extrapolating measurements at increasing lattice sizes.

There is a lot more to say about the conceptual and practical problems of Lattice Field Theory, but unfortunately we do not have time to delve into this during the lectures. However, at this point you should be well equipped to pick up a text book, for instance, on Lattice QCD and to understand the steps necessary to get to physical predictions. This week we focus on a different family of lattice models relevant to the problem of quantum gravity.

8.1 Path integrals in Quantum Gravity

All experiments so far involving interactions with gravity, think of gravitational wave observations and measurements of the cosmological microwave background, are consistent with Einstein's theory of general relativity, in which the geometry of spacetime is governed by a set of dynamical equations. It is a purely classical theory in which the matter content together with initial conditions on the geometry at some initial time, fully determines the spacetime metric $g_{\mu\nu}(x)$ at later times up to an irrelevant choice of coordinates. Such a deterministic evolution is at odds with the fundamental quantum nature of matter fields that interact with the spacetime metric. It is thus generally expected that general relativity is merely the classical limit of a fundamentally quantum theory of gravity. The reason why we have not witnessed effects of this quantum nature yet is that based on dimensional analysis the length scale below which one expects quantum fluctuations in the metric to become important is given by the **Planck length** $\ell_{\mathrm{P}} = \sqrt{\frac{\hbar G}{c^2}} \approx 1.6 \cdot 10^{-35}\mathrm{m}$, which is about a factor 10^{20} smaller than the diameter of a proton. Such length scales are still far beyond the reach of particle accelerators and other experimental setups, making direct detections of quantum-gravitational effects very difficult.

The lack of experimental input is one of the difficulties in selecting appropriate models of quantum gravity. Another problem is that the QFT techniques that have been so successful in making predictions for the other fundamental forces (in the Standard Model of particle physics) run into problems when applied to the gravitational force. The reason is that gravity is **perturbatively non-renormalizable**. Let me briefly explain this. When performing perturbation theory in a QFT one necessarily encounters infinities when integrating over the momenta in Feynman diagrams. These infinities can be absorbed by introducing counter-terms in the action, each carrying an associated coupling that needs to be determined experimentally. The problem in the case of gravity is that at every order of perturbation theory new counter-terms are required. To fully characterize the QFT one should in principle determine an infinite number of couplings from experiment, which is an unsatisfactory situation if you like your model to be predictive. Does this by itself mean that we should not aim to treat gravity as a QFT? No, it is telling us that perturbation theory, in which the metric $g_{\mu\nu}(x) = \eta_{\mu\nu} + h_{\mu\nu}(x)$ is expanded as a perturbation $h_{\mu\nu}(x)$ around flat Minkowski space $\eta_{\mu\nu}$, may not be the most economical approach to formulate the model at arbitrarily short length scales. Instead a **non-perturbative** approach may be necessary.

In the case of scalar field theory we have argued that the transition to a lattice version allows for a non-perturbative treatment, because the fields $\varphi(x)$ on the lattice sites are in no way constrained to be small. It is thus natural to seek a similar discretization of the spacetime metric appearing in the tentative QFT of gravity. There is, however, a conceptual difference between scalar field theory and general relativity. The scalar field lives on a fixed background geometry in the form of Minkowski space, while in general relativity the metric itself describes the geometry on which it lives. This has important consequences for the first two steps in programme outlined above: the Wick rotation and the lattice discretization.

8.1.1 Wick rotation

It is not clear, even at a purely formal level, what a Wick rotation entails in general relativity. An important aspect of general relativity is that one is completely free in choosing coordinates on spacetime. Unless the geometry is highly symmetric, this means in particular that there is no preferred notion of time coordinate t and therefore it is not clear what it means to analytically continue the real time t to imaginary time τ. This is not necessarily a deal breaker, because, as emphasized above, the Wick rotation should only be viewed as a guiding principle anyway. It guides the choice of a Euclidean Quantum Field Theory that shares properties with its real time counterpart. In the case of general relativity a natural proposal for the path integral and its Euclidean version are

$$\mathcal{Z} = \int [\mathcal{D}g_{\mu\nu}] e^{\frac{i}{\hbar} S[g_{\mu\nu}]}, \qquad Z = \int [\mathcal{D}g_{ab}] e^{-\frac{1}{\hbar} S_E[g_{ab}]}$$

where the first functional integral is over all (Lorentzian) metrics of signature $(d-1,1)$, i.e. metrics that are locally d-dimensional Minkowski space, while the integral in the Euclidean path integral is over (Riemannian) metrics of signature $(4,0)$ corresponding to locally d-dimensional Euclidean geometry. A natural choice of action is the classical action that gives rise upon variation to the Einstein equations, i.e. the **Einstein-Hilbert action** and its Euclidean counter part

$$S[g_{\mu\nu}] = \frac{c^4}{16\pi G} \int \mathrm{d}^d x \sqrt{-g}(R - 2\Lambda), \qquad S_E[g_{ab}] = \frac{c^4}{16\pi G} \int \mathrm{d}^d x \sqrt{g}(-R + 2\Lambda),$$

where g and R_g are the determinant and Ricci curvature scalar of the $d \times d$ matrix $g_{\mu\nu}$ respectively g_{ab}. There are two free parameters in the model: Newton's constant G and the cosmological constant Λ. The Euclidean path integral Z stands a chance of having a statistical interpretation since $e^{-\frac{1}{\hbar} S_E[g_{ab}]}$ is always real and positive. The situation is not as nice as in the scalar field case though, because $S_E[g_{ab}]$ is not bounded below: the classical solutions are saddle points of $S_E[g_{ab}]$, not minima as in the scalar field case. Whether the path integral can be rendered finite, despite featuring an integrand $e^{-\frac{1}{\hbar} S_E[g_{ab}]}$ that can be arbitrarily large, is not something one can decide based on purely formal grounds.

8.1.2 Lattice discretization: Dynamical Triangulations

In the case of Euclidean scalar field theory constructing a lattice discretization was relatively straightforward. Since the scalar field was living on a fixed Euclidean geometry, we chose to approximate this geometry by a regular lattice with a fixed distance a between the lattice sites and to only record the scalar field at these sites. In the Euclidean gravity context, we have no fixed Euclidean background that we can replace by a lattice. Of course we could fix a choice of d coordinates for our metrics and consider an ad hoc regular lattice in these coordinates. But this has two unpleasant consequences. First, this breaks the symmetry of general relativity that says that all coordinate systems should be treated equally. Second, the distance between neighboring lattice sites is not a parameter one can set, but is a dynamical quantity itself as it depends on the metric g_{ab} at those sites. This means that, without imposing further restrictions on the metric, introducing a lattice in coordinate space does not introduce a minimal length scale at all, like it did in the scalar field case.

One way to deal with this problem goes under the name of **Dynamical Triangulations**. It takes the idea seriously that the regular hypercubic lattice in ordinary Lattice Field Theory really is a discretization of the flat Euclidean geometry on which the field lives: the field configuration is in a sense built from assembling a large number of elementary hypercubes, each carrying a scalar field value, in a regular grid. Since in general relativity spacetime itself is dynamical, it is natural that in a lattice discretization of the model the lattice structure itself becomes dynamical. One could start with the same elementary building blocks, each representing a hypercube of side length a in d-dimensional Euclidean space, but assembling them in a non-regular fashion, thereby producing lattices with very different geometries. In practice, instead of using the hypercube (square in $d = 2$, cube in $d = 3$, ...) it is more convenient to use a d-simplex (triangle in $d = 2$, tetrahedron in $d = 3$, ...) with fixed side length a as the elementary building block. In order to specify a geometry built from N simplices it is sufficient to specify how the $(d+1)N$ sides of the N simplices are glued together, i.e. how the sides are paired and how each pair is oriented.

A **triangulation** of size N of a d-dimensional manifold $\mathcal{M}$ is a gluing of N identical d-simplices such that the resulting geometry has the same topology as $\mathcal{M}$. For instance, a triangulation of the 2-sphere is a gluing of N equilateral triangles, such that the resulting surface is connected, has no boundaries and no handles. The intuitive idea is now that the geometry corresponding to any metric g_{ab} on the manifold $\mathcal{M}$ can be approximated well (in some

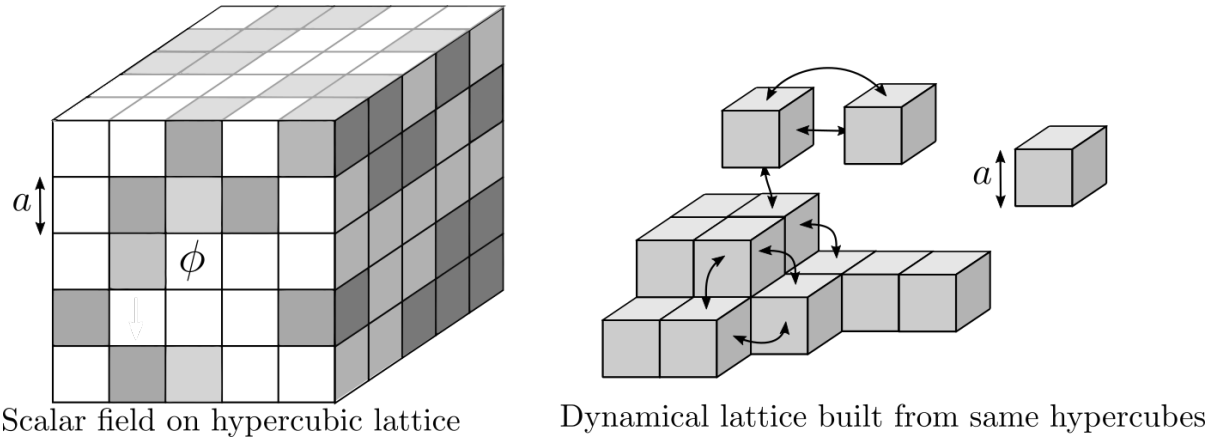


Fig. 8.1: Lattice field theory versus a dynamical lattice.

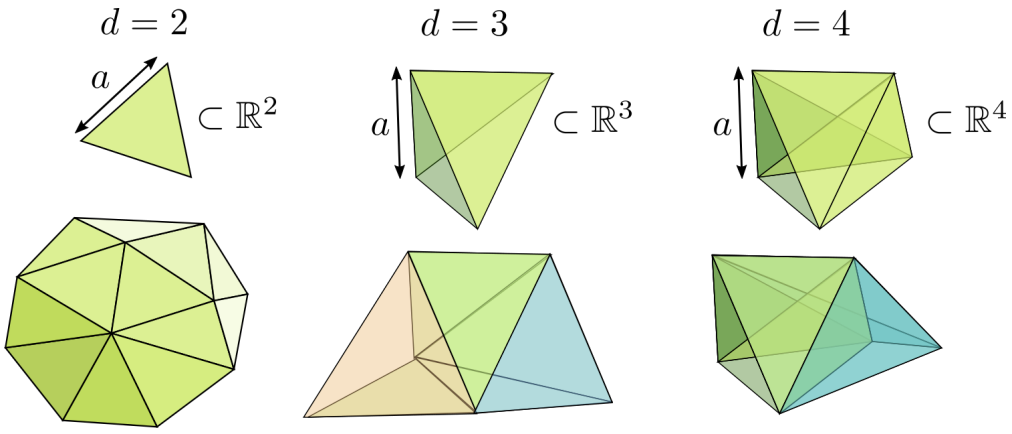


Fig. 8.2: Triangulations are constructed from simplices.

sense) by a triangulation of sufficiently large size N and sufficiently small side length a. It is then natural to discretize the path integral Z by replacing the functional integral over metrics by a summation over all triangulations,

$$Z = \int [\mathcal{D}g_{ab}] e^{-\frac{1}{\hbar} S_E[g_{ab}]} \quad \longrightarrow \quad Z_{\mathcal{M},N} = \sum_T e^{-S_{DT}[T]}.$$

Here the sum is over all triangulations T of $\mathcal{M}$ of size N and $S_{DT}[T]$ is an appropriate discretization of the Einstein-Hilbert action. The cosmological constant Λ in the action $S_E[g_{ab}]$ simply multiplies the volume $\int \mathrm{d}^d x\sqrt{g}$ of the metric space, so a natural discrete counterpart is a term λN in the discrete action since each d-simplex contributes the same fixed volume, which can be absorbed into the defintion of the dimensionless cosmological constant λ. In $d = 2$ the curvature integral $\int \mathrm{d}^2 x\sqrt{g}R$ only depends on the topology of $\mathcal{M}$ and is thus independent of the metric (the Gauss-Bonnet theorem). Since we fix both N and the topology in the discretized path integral, the action is constant, so the natural discretization is $S_{DT}[T] = 0$ in 2D! In $d = 3$ and $d = 4$ it can be shown that the integrated curvature in the piecewise flat geometry of a triangulation (knows as the **Regge action**) is linear in the number N_0 of vertices of the triangulation, so one naturally sets $S_{DT}[T] = -\kappa_0 N_0$.

8.2 Monte Carlo simulation of 2D Dynamical Triangulations

Let us focus on the case $d = 2$, i.e. on the toy model of quantum gravity in two spacetime dimensions, and take the spacetime manifold to be the 2-sphere S^2. To make the statistical system very precise let us discuss how a triangulation T can be described succinctly (which helps when representing it on the computer). It should be constructed from N equilateral triangles by pairwise gluing the $3N$ sides (note that N should thus be even!). Let us label the triangles from 0 to $N-1$ and label all sides from 0 to $3N-1$, such that the triangle with label i has sides with labels $3i, 3i+1, 3i+2$ in counter-clockwise direction. Then specifying the adjacency of sides amounts to specifying a permutation $\mathrm{adj} : \{0, \dots, 3N-1\} \to \{0, \dots, 3N-1\}$, which should satisfy $\mathrm{adj}(x) \neq x$ and $\mathrm{adj}(\mathrm{adj}(x)) = x$ for all x (i.e. it is a "fixed-point free involution"). For instance the following triangulation of S^2 is specified by the permutation

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 \overset{\mathrm{adj}}{\longrightarrow} 9, 4, 6, 7, 1, 11, 2, 3, 10, 0, 8, 5.$$

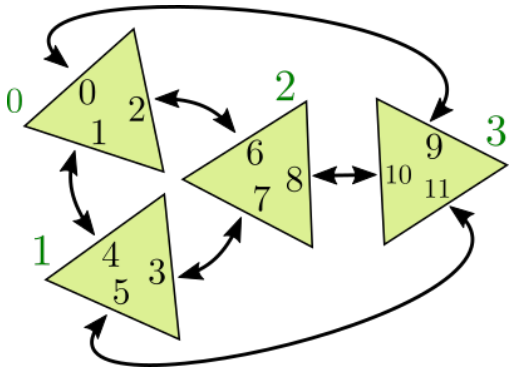


Fig. 8.3: An example of a triangulations of the 2-sphere.

We can store this information conveniently in an integer array.

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

example_adj = np.array([9,4,6,7,1,11,2,3,10,0,8,5],dtype=np.int32)
```

Representing a triangulation through its adjacency permutation, we can define the partition function of 2D dynamical triangulations of the 2-sphere as

$$Z_{S^2,N} = \sum_T e^{-S_{DT}[T]} = \sum_T 1.$$

Therefore the desired probability distribution is the uniform distribution $\pi(T) = 1/Z_{S^2,N}$ on the space of all triangulations of S^2 of size N.

Let's use MCMC techniques to sample from this distribution! We first need a valid initial state of desired size N. Any valid triangulation will do. Here is one that is easy to code, resulting in a "fan-shaped" triangulation of the 2-sphere.

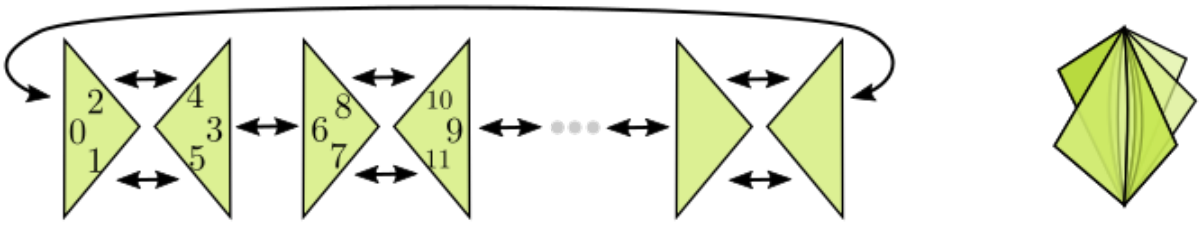


Fig. 8.4: A triangulation of the 2-sphere constructed by hand.

```
def fan_triangulation(n):
    '''Generates a fan-shaped triangulation of even size n.'''
    return np.array([[(i-3)%(3*n),i+5,i+4,(i+6)%(3*n),i+2,i+1] for i in range(0,
 ↪3*n,6)],dtype=np.int32).flatten()
```

8.2.1 Sanity checks

Although this section is not strictly necessary for the implementation of the simulations, it is very useful for debugging purposes to be able to check whether a permutation determines a valid triangulation of the sphere. First of all we need to make sure that it is a fixed-point free involution:

```
def is_fpf_involution(adj):
    '''Test whether adj defines a fixed-point free involution.'''
    for x, a in enumerate(adj):
        if a < 0 or a >= len(adj) or x == a or adj[a] != x:
            return False
    return True

print(is_fpf_involution(example_adj))
print(is_fpf_involution(fan_triangulation(50)))
```

```
True
True
```

This means it determines a proper closed surface, since each side of a triangle is glued to another side. But it could be that the surface has multiple connected components. So let us compute the number of components.

```
from collections import deque

def triangle_neighbours(adj,i):
    '''Return the indices of the three neighboring triangles.'''
    return [j//3 for j in adj[3*i:3*i+3]]

def connected_components(adj):
    '''Calculate the number of connected components of the triangulation.'''
```

(continues on next page)

(continued from previous page)

```
    n = len(adj)//3   # the number of triangles
    component = np.full(n,-1,dtype=np.int32)   # array storing the component␣
↪index of each triangle
    index = 0
    for i in range(n):
        if component[i] == -1:    # new component found, let us explore it
            component[i] = index
            queue = deque([i])   # use an exploration queue for breadth-first␣
↪search
            while queue:
                for nbr in triangle_neighbours(adj,queue.pop()):
                    if component[nbr] == -1:  # the neighboring triangle has not␣
↪been explored yet
                        component[nbr] = index
                        queue.appendleft(nbr)   # add it to the exploration queue
            index += 1
    return index

print(connected_components(example_adj))
print(connected_components(fan_triangulation(50)))
```

```
1
1
```

Finally, if we know that it has a single connected component, we can check that it has the topology of the sphere by verifying Euler's formula $V - E + F = 2$, where V, $E = 3N/2$, $F = N$ are the number of vertices, edges and faces respectively. The number V of vertices can be established by an exploration process.

```
def next_around_triangle(i):
    '''Return the label of the side following side i in counter-clockwise␣
↪direction.'''
    return i - i%3 + (i+1)%3

def prev_around_triangle(i):
    '''Return the label of the side preceding side i in counter-clockwise␣
↪direction.'''
    return i - i%3 + (i-1)%3

def vertex_list(adj):
    '''
    Return the number of vertices and an array `vertex` of the same size as `adj`,
↪
    such that `vertex[i]` is the index of the vertex at the start (in ccw order)␣
↪of the side labeled `i`.
    '''
    vertex = np.full(len(adj),-1,dtype=np.int32)  # a side i that have not been␣
↪visited yet has vertex[i]==-1
    vert_index = 0  #
    for i in range(len(adj)):
        if vertex[i] == -1:
            side = i
            while vertex[side] == -1:  # find all sides that share the same vertex
                vertex[side] = vert_index
                side = next_around_triangle(adj[side])
            vert_index += 1
    return vert_index, vertex

def number_of_vertices(adj):
    '''Calculate the number of vertices in the triangulation.'''
```

(continues on next page)

(continued from previous page)

```
    return vertex_list(adj)[0]

def is_sphere_triangulation(adj):
    '''Test whether adj defines a triangulation of the 2-sphere.'''
    if not is_fpf_involution(adj) or connected_components(adj) != 1:
        return False
    num_vert = number_of_vertices(adj)
    num_face = len(adj)//3
    num_edge = len(adj)//2
    # verify Euler's formula for the sphere
    return num_vert - num_edge + num_face == 2

print("V = {}, vertex id for each side = {}. Can you verify this in the picture␣
 ↪above?"
      .format(*vertex_list(example_adj)))
print("Valid triangulation?",is_sphere_triangulation(example_adj))
print("Valid triangulation?",is_sphere_triangulation(fan_triangulation(50)))
```

```
V = 4, vertex id for each side = [0 1 2 3 2 1 0 2 3 1 0 3]. Can you verify this␣
 ↪in the picture above?
Valid triangulation? True
Valid triangulation? True
```

8.2.2 Monte Carlo Transition

A natural proposal transition $Q(T \to T')$ is obtained by selecting a uniform random side $i \in \{0, \dots, 3N-1\}$ of one of the triangles and performing a **triangle flip move** as illustrated in the following figure. The other labels j, k, ℓ, m, n in the figure can be inferred from i and the permutation adj, e.g. $k = \mathrm{adj}(i)$. It should be clear from the figure that this move amounts to an appropriate update of the six values $\mathrm{adj}(i), \mathrm{adj}(j), \dots, \mathrm{adj}(n)$.

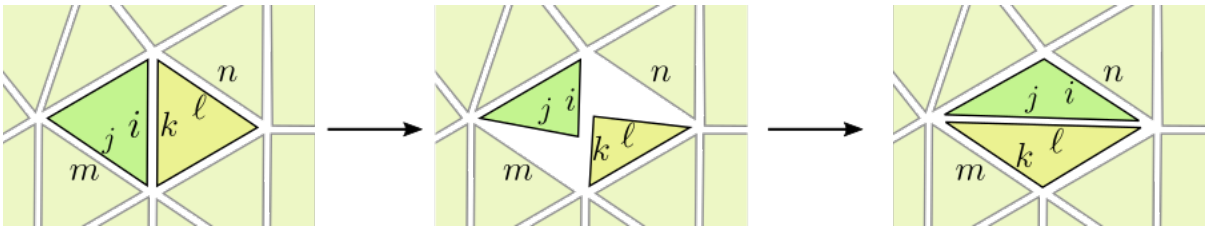


Fig. 8.5: Monte Carlo update: flipping a pair of triangles.

```
def flip_edge(adj,i):
    if adj[i] == next_around_triangle(i) or adj[i] == prev_around_triangle(i):
        # flipping an edge that is adjacent to the same triangle on both sides␣
 ↪makes no sense
        return False
    j = prev_around_triangle(i)
    k = adj[i]
    l = prev_around_triangle(k)
    n = adj[l]
    adj[i] = n  # it is important that we first update
    adj[n] = i  # these adjacencies, before determining m,
    m = adj[j]  # to treat the case j == n appropriately
    adj[k] = m
    adj[m] = k
    adj[j] = l
    adj[l] = j
```

(continues on next page)

(continued from previous page)

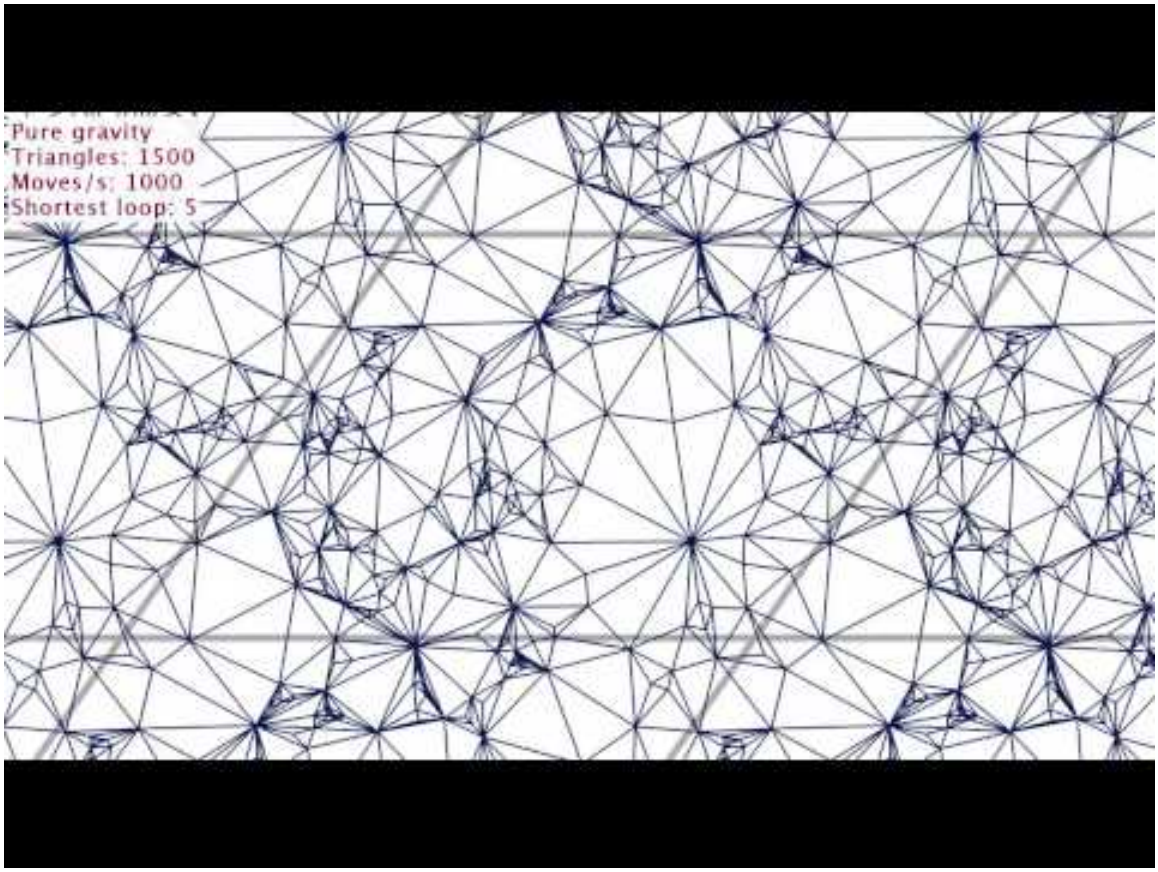
```
    return True

def random_flip(adj):
    random_side = rng.integers(0,len(adj))
    return flip_edge(adj,random_side)

# check that after many triangle flips we still have a valid configuration
adj = fan_triangulation(100)
for _ in range(10000):
    random_flip(adj)
is_sphere_triangulation(adj)
```

```
True
```

Next we should examine detailed balance. First we note that if we had selected k instead of i as the initial side, the move would have been identical. So in this case the proposal probability is $Q(T \to T') = 2/(3N)$. Second we need to compare it with the probability $Q(T' \to T)$ of the reverse being proposed. A moment's consideration however will reveal that $Q(T' \to T) = 0$, due to the way in which the flip is always implemented in a clockwise fashion, while the reverse should flip in counter-clockwise direction. We could ensure rejection-free detailed balance $\tilde{Q}(T \to T') = \tilde{Q}(T' \to T)$ for a slightly adapted proposal $\tilde{Q}(T \to T')$ by first flipping a fair coin and based on the outcome performing the flip move clockwise (as presented above) or counterclockwise (the same but in a mirrored fashion). In practice, there is no point in doing so because both types of flip moves result in structurally identical triangulations: only the physically irrelevant labeling of the triangulation is affected by this choice. We thus conclude that modulo relabeling of the triangulations the proposed flip move satisfies detailed balance for the uniform distribution $\pi(T) = 1/Z_{S^2,N}$!

It should be clear from the figure that the flip move does not change the 2-sphere topology. Although we will not prove it here, it is not too difficult to show that any two triangulations of the 2-sphere with N triangles can be obtained from each other by a finite number of flip moves. Hence, the Markov chain we described is irreducible (and aperiodic). This is a special case of a general theorem by Pachner that holds for any dimension d, with suitable generalizations of the flip moves (known as Pachner moves or bistellar moves).

The following YouTube video illustrates this Markov chain in the case of a triangulation of the torus, starting with a regular triangular tiling. The configurations are displayed as periodic triangulations of the two-dimensional plane (using a canonical graph embedding algorithm known as the Tutte embedding): the torus is obtained by identifying opposite sides of the indicated parallelogram.



8.2.3 Visualizing random triangulations

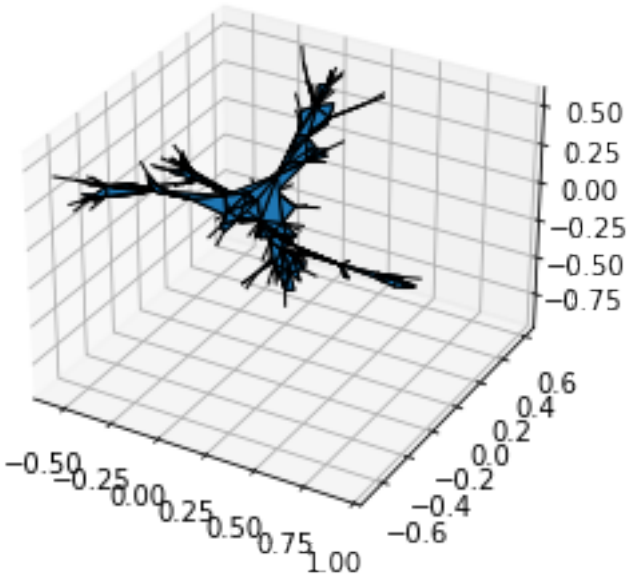
Convergence of the Markov chain to the uniform distribution is therefore guaranteed. What does a uniform random triangulation look like? It is not so easy to produce a faithful visualization of such a triangulation, because it will be very far from regular. We can make an attempt at embedding the surface in 3D, but as can be seen below the result is not entirely satisfactory (mainly due to limitations of the networkX layout algorithm).

```
import networkx as nx
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

def triangulation_edges(triangulation,vertex):
    '''Return a list of vertex-id pairs corresponding to the edges in the␣
 ↪triangulation.'''
    return [(vertex[i],vertex[j]) for i,j in enumerate(triangulation) if i < j]

def triangulation_triangles(triangulation,vertex):
    '''Return a list of vertex-id triples corresponding to the triangles in the␣
 ↪triangulation.'''
    return [vertex[i:i+3] for i in range(0,len(triangulation),3)]

def plot_triangulation_3d(adj):
    '''Display an attempt at embedding the triangulation in 3d.'''
    num_vert, vertex = vertex_list(adj)
    edges = triangulation_edges(adj,vertex)
    triangles = triangulation_triangles(adj,vertex)
    # use the networkX 3d graph layout algorithm to find positions for the␣
 ↪vertices
    pos = np.array(list(nx.spring_layout(nx.Graph(edges),dim=3).values()))
    fig = plt.figure()
```

(continues on next page)

(continued from previous page)

```
    ax = fig.add_subplot(111, projection='3d')
    tris = Poly3DCollection(pos[triangles])
    tris.set_edgecolor('k')
    ax.add_collection3d(tris)
    ax.set_xlim3d(np.amin(pos[:,0]),np.amax(pos[:,0]))
    ax.set_ylim3d(np.amin(pos[:,1]),np.amax(pos[:,1]))
    ax.set_zlim3d(np.amin(pos[:,2]),np.amax(pos[:,2]))
    plt.show()

adj = fan_triangulation(500)
for _ in range(100000):
    random_flip(adj)
plot_triangulation_3d(adj)
```



Rather more impressive visualizations can be produced by increasing the number of triangles and using better layout software. Here is a snapshot of a uniform triangulation with $N = 4\ 000\ 000$ triangles produced by Benedikt Stufler.

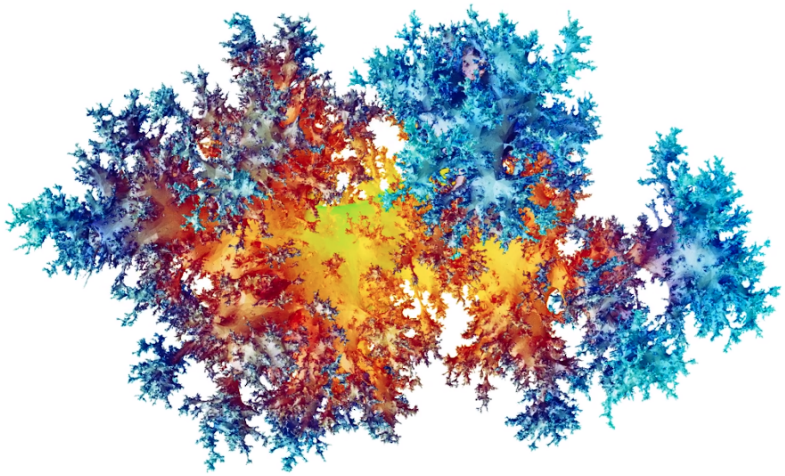


Fig. 8.6: Visualization of a uniform triangulation of the 2-sphere ($N = 4\ 000\ 000$). Credit: Benedikt Stufler

It is remarkable that a model of random geometry with such a simple distribution (uniform distribution on the set

of all triangulations) can produce such intricate structures. One may note some similarities with the Ising model at criticality, namely that the configuration displays **fractal features** in the sense that structures are visible at all length scales. Could it be that 2D dynamical triangulations displays critical phenomena without having tuned the system to a continuous phase transition? To find out we should look for the existence of critical exponents when N becomes large.

8.3 Geometric observables

Just like in general relativity physical quantities should not depend on the chosen coordinate system, in dynamical triangulations observables should not depend on the labeling of the triangulation. It is actually not so straightforward to cook up meaningful observables of this type, since the dynamics purely resides in the connectivity of the triangulation. One well-studied observable that exhibits interesting scaling behavior is the **distance profile** $\rho_T(r)$ of the triangulation T. It is defined as

$$\rho_T(r) = \frac{2}{N+4} \sum_x \sum_y \mathbf{1}_{\{d_T(x,y)=r\}},$$

where the sums run over the $(N+4)/2$ vertices of the triangulation T and $d_T(x,y)$ is the **graph distance** between x and y along the edges of the triangulation, i.e. the smallest number of edges in a path connecting x and y. Since it involves a sum over all the vertices of the triangulation, it should be clear that $\rho_T(r)$ does not depend on the labeling. The expectation value $\mathbb{E}[\rho_T(r)]$ can also be understood as the expected number of vertices that have distance r from a uniformly chosen vertex in a random triangulation T. If T were close to a regular triangulation, i.e. similar to a smooth two-dimensional manifold, then the expected distance profile would grow roughly linearly in r in the regime $1 \ll r \ll \sqrt{N}$.

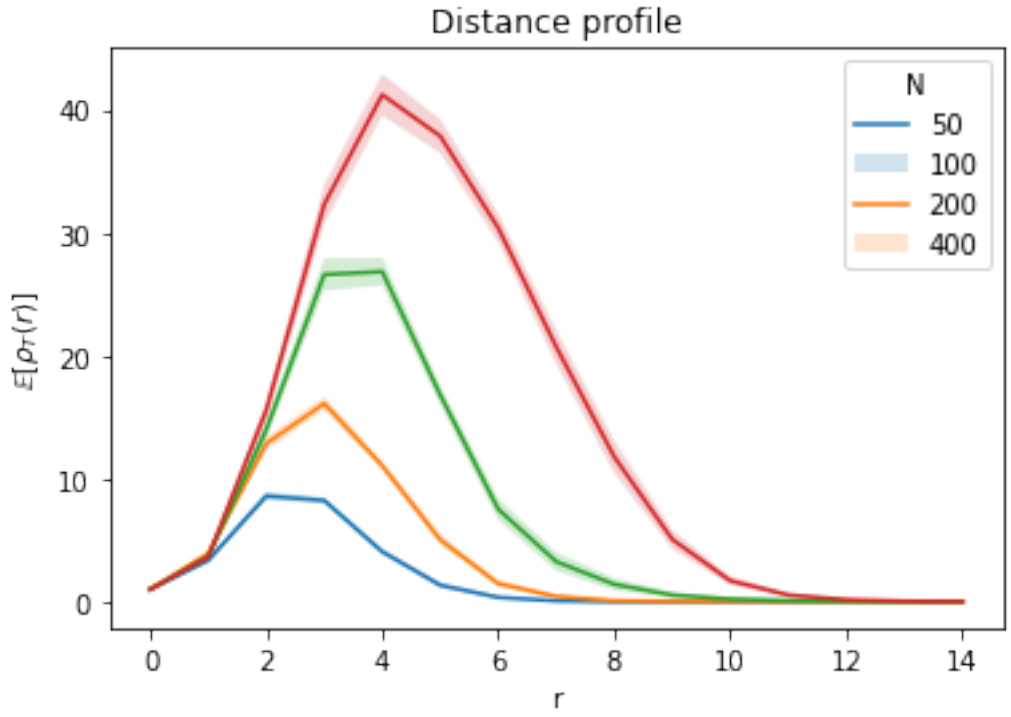
```
def vertex_neighbors_list(adj):
    '''Return a list `neighbors` such that `neighbors[v]` is a list of neighbors␣
↪of the vertex v.'''
    num_vertices, vertex = vertex_list(adj)
    neighbors = [[] for _ in range(num_vertices)]
    for i,j in enumerate(adj):
        neighbors[vertex[i]].append(vertex[j])
    return neighbors

def vertex_distance_profile(adj,max_distance=30):
    '''Return array `profile` of size `max_distance` such that `profile[r]` is␣
↪the number
    of vertices that have distance r to a randomly chosen initial vertex.'''
    profile = np.zeros((max_distance),dtype=np.int32)
    neighbors = vertex_neighbors_list(adj)
    num_vertices = len(neighbors)
    start = rng.integers(num_vertices) # random starting vertex
    distance = np.full(num_vertices,-1,dtype=np.int32)  # array tracking the␣
↪known distances (-1 is unknown)
    queue = deque([start])   # use an exploration queue for the breadth-first␣
↪search
    distance[start] = 0
    profile[0] = 1  # of course there is exactly 1 vertex at distance 0
    while queue:
        current = queue.pop()
        d = distance[current] + 1  # every unexplored neighbour will have this␣
↪distance
        if d >= max_distance:
            break
        for nbr in neighbors[current]:
            if distance[nbr] == -1:  # this neighboring vertex has not been␣
↪explored yet
                distance[nbr] = d
                profile[d] += 1
```

(continues on next page)

(continued from previous page)

```
                queue.appendleft(nbr)    # add it to the exploration queue
    return profile

def perform_sweeps(adj,t):
    '''Perform t sweeps of flip moves, where 1 sweep is N moves.'''
    for _ in range(len(adj)*t//3):
        random_flip(adj)

def batch_estimate(data,observable,num_batches):
    batch_size = len(data)//num_batches
    values = [observable(data[i*batch_size:(i+1)*batch_size]) for i in range(num_
 ↪batches)]
    return np.mean(values), np.std(values)/np.sqrt(num_batches-1)

sizes = [50,100,200,400]
equilibration_sweeps = 500
measurement_sweeps = 2
measurements = 200

mean_profiles = []
for size in sizes:
    adj = fan_triangulation(size)
    perform_sweeps(adj,equilibration_sweeps)
    profiles = []
    for _ in range(measurements):
        perform_sweeps(adj,measurement_sweeps)
        profiles.append(vertex_distance_profile(adj,15))
    mean_profiles.append([batch_estimate(data,np.mean,20) for data in np.
 ↪transpose(profiles)])
```

```
for profile in mean_profiles:
    plt.plot([y[0] for y in profile])
    plt.fill_between(range(len(profile)),[y[0]-y[1] for y in profile],[y[0]+y[1]␣
 ↪for y in profile],alpha=0.2)
plt.legend(sizes,title="N")
plt.xlabel("r")
plt.ylabel(r"$\mathbb{E}[\rho_T(r)]$")
plt.title("Distance profile")
plt.show()
```



It does appear that $\mathbb{E}[\rho_T(r)]$ could converges as N becomes large for fixed r, but then the limit looks nothing like linear in r. What we are seeing are the fractal properties of the random triangulation and there is a natural critical exponent associated to it, which is called the **fractal dimension** or **Hausdorff dimension** d_H, defined as

$$\lim_{N\to\infty} \mathbb{E}[\rho_T(r)] \sim r^{d_H-1} \qquad \text{as } r \to \infty.$$

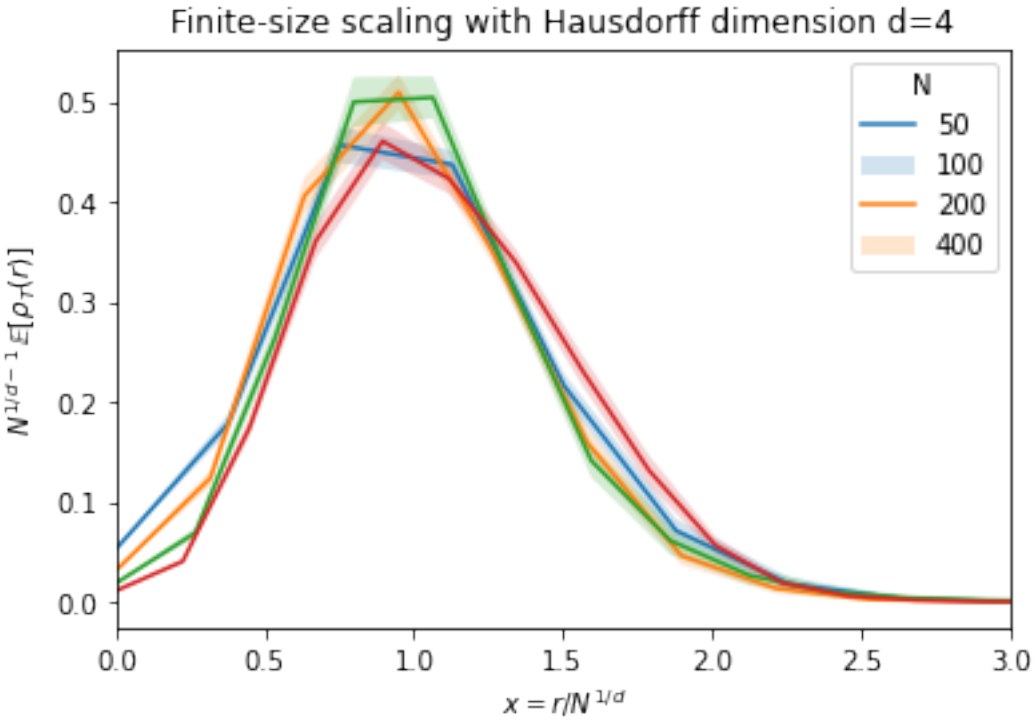
The reason for calling it a dimension is that on a deterministic regular lattice of dimension d, the Hausdorff dimension would be equal to $d_H = d$. Measuring this exponent is difficult because the approximation $\mathbb{E}[\rho_T(r)] \approx r^{d_H-1}$ is only valid when $1 \ll r \ll N^{1/d_H}$.

A more convenient approach goes under the name of **finite-size scaling**, which aims to use the full distance profile in order to deduce the scaling of r relative to the size N. It is based on the ansatz that after appropriate rescaling of the x and y axes of the plots of $\mathbb{E}[\rho_T(r)]$ by powers of N, the curves should **collapse** in the large-N limit. Using that $\sum_r \mathbb{E}[\rho_T(r)] = N$, the appropriately scaled functions are

$$x \to N^{1/d_H-1}\, \mathbb{E}[\rho_T(xN^{1/d_H})].$$

One can then estimate d_H by optimizing the collapse of these functions for several choices of N. This you will investigate in the exercises. The following plot already indicates that d_H is close to 4 (in fact, that $d_H = 4$ can be proved fully rigorously).

```
for i, profile in enumerate(mean_profiles):
    rvals = np.arange(len(profile))
    plt.plot(rvals/sizes[i]**0.25,[y[0]/ sizes[i]**0.75 for y in profile])
    plt.fill_between(rvals/sizes[i]**0.25,[(y[0]-y[1])/sizes[i]**0.75 for y in␣
 ↪profile],[(y[0]+y[1])/sizes[i]**0.75 for y in profile],alpha=0.2)
plt.legend(sizes,title="N")
plt.xlabel(r"$x = r / N^{1/d}$")
plt.ylabel(r"$N^{1/d-1}\,\mathbb{E}[\rho_T(r)]$")
plt.xlim(0,3)
plt.title("Finite-size scaling with Hausdorff dimension d=4")
plt.show()
```



8.4 Further reading

Parts of this week's lecture are based on a mini-course I gave several years back on Monte Carlo methods in Dynamical Triangulations. The slides and recordings are available on the course webpage.

A useful textbook on this topic is the book by Ambjørn, Durhuus, Jonsson, [AmbjornDJ97].

For recent developments in related models, you could have a look at [AGJL12].

Chapter 9

Practicalities

In the past weeks of the course you have had the opportunity to familiarize yourself with the nuts and bolts of Monte Carlo simulations and have tested your skills with the computer assignments in the jupyter notebooks. In doing so, especially for the computation intensive models of the last few weeks, you probably have contemplated whether the process could not be organized more efficiently: perhaps you were running the simulation in the notebook and had to wait for it to finish before continuing with the analysis; or you realized upon coming back to the exercises next day that you had to rerun the simulations because the data was lost; or the 3-hour limit on the jupyterhub sessions was getting in the way of producing a satisfactory amount of data; or the different pieces of code you have written (and rewritten, adapted, patched,…) are scattered in a non-linear fashion in the notebook cells; or…. If you have not encountered these limitations yet, you will certainly face them in the project you are about to commence. It is therefore a good idea that to discuss some organizational aspects of running a medium to large scale simulation project.

9.1 Coding

9.1.1 An example: the 2D Ising model

Let us go through an explicit example: simulating the 2D Ising model with the Wolff algorithm as we did in week 6. The first thing we should do is make a stand-alone program that can perform the required simulation. For this we can use the Python script `ising.py` available in the folder `ising_example`.

```
import numpy as np
from collections import deque
import argparse
import time
import json

rng = np.random.default_rng()

# Required for easily parsing NumPy arrays into a JSON file (https://stackoverflow.
↪com/a/49677241)
class NumpyEncoder(json.JSONEncoder):
    """ Special json encoder for numpy types """
    def default(self, obj):
        if isinstance(obj, np.integer):
            return int(obj)
        elif isinstance(obj, np.floating):
            return float(obj)
        elif isinstance(obj, np.ndarray):
            return obj.tolist()
        return json.JSONEncoder.default(self, obj)
```

(continues on next page)

(continued from previous page)

```
# Ising model code from week 6
def aligned_init_config(width):
    '''Produce an all +1 configuration.'''
    return np.ones((width,width),dtype=int)

def neighboring_sites(s,w):
    '''Return the coordinates of the 4 sites adjacent to s on an w*w lattice.'''
    return [((s[0]+1)%w,s[1]),((s[0]-1)%w,s[1]),(s[0],(s[1]+1)%w),(s[0],(s[1]-1)
 ↪%w)]

def cluster_flip(state,seed,p_add):
    '''Perform a single Wolff cluster move with specified seed on the state with␣
 ↪parameter p_add.'''
    w = len(state)
    spin = state[seed]
    state[seed] = -spin
    cluster_size = 1
    unvisited = deque([seed])    # use a deque to efficiently track the unvisited␣
 ↪cluster sites
    while unvisited:   # while unvisited sites remain
        site = unvisited.pop()  # take one and remove from the unvisited list
        for nbr in neighboring_sites(site,w):
            if state[nbr] == spin and rng.uniform() < p_add:
                state[nbr] = -spin
                unvisited.appendleft(nbr)
                cluster_size += 1
    return cluster_size

def wolff_cluster_move(state,p_add):
    '''Perform a single Wolff cluster move on the state with addition probability␣
 ↪p_add.'''
    seed = tuple(rng.integers(0,len(state),2))
    return cluster_flip(state,seed,p_add)

def run_ising_wolff_mcmc(state,p_add,n):
    '''Run n Wolff moves on state and return total number of spins flipped.'''
    total = 0
    for _ in range(n):
        total += wolff_cluster_move(state,p_add)
    return total

def compute_magnetization(config):
    '''Compute the magnetization M(s) of the state config.'''
    return np.sum(config)

# use the argparse package to parse command line arguments
parser = argparse.ArgumentParser(description='Measures the magnetization in the 2D␣
 ↪Ising model')
parser.add_argument('-w', type=int, help='Lattice size W')
parser.add_argument('-t', type=float, help='Temperature T (in units where J=kB=1)')
parser.add_argument('-n', type=int, help='Number N of measurements (indefinite by␣
 ↪default)')
parser.add_argument('-e', type=int, default=100, help='Number E of equilibration␣
 ↪sweeps')
parser.add_argument('-m', type=int, default=10, help='Number M of sweeps per␣
 ↪measurement')
parser.add_argument('-o', type=int, default=30, help='Time in seconds between file␣
 ↪outputs')
parser.add_argument('-f', help='Output filename')
args = parser.parse_args()
```

(continues on next page)

(continued from previous page)

```
# perform sanity checks on the arguments
if args.w is None or args.w < 1:
    parser.error("Please specify a positive lattice size!")
if args.t is None or args.t <= 0.0:
    parser.error("Please specify a positive temperature!")
if args.e < 10:
    parser.error("Need at least 10 equilibration sweeps")

# fix parameters
temperature = args.t
p_add = 1 - np.exp(-2/temperature)
width = args.w
if args.f is None:
    # construct a filename from the parameters plus a timestamp (to avoid␣
 ↪overwriting)
    output_filename = "data_w{}_t{}_{}.json".format(width,temperature,time.
 ↪strftime("%Y%m%d%H%M%S"))
else:
    output_filename = args.f

# equilibration
state = aligned_init_config(width)
total_spin_flips = 0
total_moves = 0
equilibration_target = args.e * width * width
while total_spin_flips < equilibration_target:
    total_spin_flips += wolff_cluster_move(state,p_add)
    total_moves += 1
average_cluster_size = total_spin_flips / total_moves

# measurement phase
moves_per_measurement = int(args.m * width * width / average_cluster_size)
measurements = 0
start_time = time.time()
last_output_time = time.time()
magnetizations = []
while True:
    run_ising_wolff_mcmc(state,p_add,moves_per_measurement)
    magnetizations.append(compute_magnetization(state))
    measurements += 1

    if measurements == args.n or time.time() - last_output_time > args.o:
        # output as json file
        with open(output_filename,'w') as outfile:
            json.dump({
                'parameters': vars(args),
                'average_cluster_size': average_cluster_size,
                'start_time': time.asctime(time.localtime(start_time)),
                'current_time': time.asctime(),
                'run_time_in_seconds': int(time.time() - start_time),
                'measurements': measurements,
                'moves_per_measurement': moves_per_measurement,
                'magnetizations': magnetizations
                }, outfile, cls=NumpyEncoder)
        if measurements == args.n:
            break
        else:
            last_output_time = time.time()
```

Let me highlight two features that are new compared to what we are used to in the Jupyter notebooks:

Command line arguments It uses the `argparse` package to read the simulation parameters from the command line. This has the great advantage that we can change all parameters without changing the source code. For instance, if we wish to run a simulation of the Ising model on a 20x20 grid at temperature $T = 2.5$ and obtain 200 samples we would run

```
$ python3 ising.py -w 20 -t 2.5 -n 200 -f data_example.json
```

and the output would be stored in the file `data_example.json`. Note that there are several more optional command line parameters one can set (equilibration time, output file, …) but they are assigned default values for convenience.

Storing results The simulation results are stored in a JSON file, which is a versatile and universally readable file format. This is done at the very end of the simulation when the desired number of measurements has been performed, but also periodically every 30 seconds (but this you can set with the parameters). It has the advantage that you can monitor the progress of the simulation and kill the script at any time without loosing much data. Note that we do not just save the observables but also the command line parameters and various other quantities that may be of interest later. In general it is good practice to store all information necessary to reproduce the data.

Now suppose we have run our simulation with the command line above. We can then easily read the data in this notebook…

```
import json

with open('ising_example/data_example.json') as file:
    data = json.load(file)

print(data)
```

```
{'parameters': {'w': 20, 't': 2.5, 'n': 200, 'e': 100, 'm': 10, 'o': 30, 'f':
 ↪'data_example.json'}, 'average_cluster_size': 55.577777777777776, 'start_time
 ↪': 'Mon Nov 15 08:51:28 2021', 'current_time': 'Mon Nov 15 08:51:30 2021',
 ↪'run_time_in_seconds': 2, 'measurements': 200, 'moves_per_measurement': 71,
 ↪'magnetizations': [128, 96, -252, 72, -222, 140, 218, -48, 92, 98, 126, 232, -
 ↪218, -124, 8, 8, -276, -192, 238, 186, -176, 182, -54, -156, -122, -216, 166,␣
 ↪-126, -104, 180, -14, -74, 130, -170, -138, -48, 176, 6, -132, 140, 42, -62, -
 ↪154, 142, -108, -120, 200, -200, -174, 98, -170, 258, -90, -38, -8, -80, 160,␣
 ↪-142, -280, 138, -130, -136, -176, -256, 116, 0, 200, 62, 150, -114, -146, 84,
 ↪ -156, 156, 194, -126, 248, -46, -184, 138, 4, -152, -112, 144, 126, 246, 90,␣
 ↪-196, -34, -64, -10, -170, -28, 98, -102, 210, 160, -218, -84, 160, 90, -84,␣
 ↪38, 158, 102, -4, 120, 50, -72, -122, -166, 174, 264, -94, -132, 178, 174, -
 ↪244, -114, 84, -190, -96, 160, 288, -62, 126, 158, 86, 8, -134, -206, -262, -
 ↪266, 202, 294, 234, 152, -72, 92, 32, 248, 100, 230, -10, -114, -4, -146, 28,␣
 ↪-162, 152, 122, 144, 124, 172, 288, -84, 98, -42, 236, 204, -176, -90, -246, -
 ↪78, -46, 146, 94, 162, -106, -206, 284, 296, -176, -188, -130, 124, 22, 102, -
 ↪6, -280, 226, -130, 154, -220, -270, 280, 48, -240, -166, 174, 242, 12, -78, -
 ↪44, -158, -176, 118, 156, 162, -210]}
```

… and perform data analysis.

```
import numpy as np

def batch_estimate(data,observable,num_batches):
    batch_size = len(data)//num_batches
    values = [observable(data[i*batch_size:(i+1)*batch_size]) for i in range(num_
 ↪batches)]
    return np.mean(values), np.std(values)/np.sqrt(num_batches-1)

abs_magnetizations = np.abs(np.array(data['magnetizations']))
print("Magnetization: E[|m|] = {} +- {}".format(*batch_estimate(abs_
 ↪magnetizations,np.mean,10)))
```

```
Magnetization: E[|m|] = 139.44 +- 4.563215240741261
```

9.2 Data gathering

Once you are confident that your simulation code works as intended and you have a good idea of the parameter ranges you would like to collect data for, you are at a stage to scale up and invest some serious cpu time. Of course you could occupy all cores on your pc or laptop and take a nap, but there is a more practical alternative: the compute cluster of our faculty. Those that have signed up for this course, will have been granted access to the High Energy Physics cluster nodes (node = single machine). Running jobs on the cluster nodes requires some preparations, since they are to be submitted through the SLURM job scheduling system. Don't worry, this is not complicated!

9.2.1 Logging in to one of the compute nodes

First you will have to login with `ssh` into one of the compute nodes using a terminal. There are several ways to do this.

JupyterHub terminal The first option is to use the terminal interface of JupyterHub in the browser. Simply head to the file system page and select `New` (top-right corner) and `Terminal`. This will bring up a browser tab with a direct ssh interface with to compute node on which your jupyter session is running.

From faculty-managed linux machine If you are on a faculty-managed linux machine, you can simply open the built-in Terminal (`Ctrl + Alt + T`).

From your own computer From your own computer you can use the built-in Terminal on linux or MacOS, or PuTTy on Windows, but you first have to login into one of the faculties login-servers, e.g.

```
$ ssh yourusername@lilo7.science.ru.nl
```

which will prompt for your science password.

Then login into one of the cluster nodes (cn26, cn44, cn29, cn96, cn97, cn98) by running e.g.

```
$ ssh cn44
```

If your access is denied, please contact the course coordinator because you may not have been added to the right unix groups yet. The command `sinfo` will provide information on the SLURM partitions to which you have been granted access. It will look something like this:

```
$ sinfo
PARTITION  AVAIL  TIMELIMIT  NODES  STATE NODELIST
hefstud       up   infinite      2    mix cn[96,98,110,111]
hefstud       up   infinite      4  alloc cn[26,29,44,97,112,113]
cnczshort     up   12:00:00      1   idle cn13
cncz          up 7-00:00:00      1   idle cn13
jupyter       up    4:00:00      6   idle cn[60-64,66]
```

The relevant partition is `hefstud` (for High Energy Physics students). As you see it lists the 10 cluster nodes with an indication of the status (in this example cn26, cn29, … are fully occupied and cn96, cn98, … are partially, so it is pretty busy). The currently running and queued jobs can be displayed with `squeue`. Note: `hefstud` is not the only partition using these machines, so the machines may be occupied even if `squeue` does not show any running jobs.

9.2.2 Submitting a job script

Now we wish to perform our simulation on the cluster node. You could execute it in the terminal by running `python3 ising.py ...`, but please do not do this (unless to quickly test if the code will run on this machine)! The proper way is to submit your job as a SLURM task. For this you should prepare a simple bash script. Let us assume the script `ising.py` is located in the folder `monte-carlo-techniques-2021/lectures/ising_example` in your science home directory. A minimal version of the bash script would look like this and we would save it in a file `ising_job.sh`.

```
#!/bin/bash
#SBATCH --partition=hefstud
#SBATCH --output=std_%A_%a.txt
#SBATCH --mem=100M
#SBATCH --time=2:00:00
cd ~/monte-carlo-techniques-2021/lectures/ising_example
/software/anaconda3/bin/python3 ising.py -w 20 -t 2.5 -n 100
```

The lines with `#SBATCH` tell SLURM what resources are requested and how to administrate the job.

- In particular, we are telling SLURM that the job should run on the `hefstud` partition (so on one of its 10 nodes).
- Any output generated by your script together with the job details will be stored in a file whose name depends on the job id (but you could specify a filename yourself if you wish).
- The `--mem=100M` tells SLURM that you plan not to use more than 100MB of memory.
- The `--time=2:00:00` option specifies the maximal run time of your script before it will be killed (in this case 2 hours). You can specify a maximal run time of, say, 3 days using the format `--time=3-0:00:00`.

The lines that do not start with # are commands that will be executed, in this case the working directory is set to your home directory and then the python script is executed. **Important**: use python3 provided by the anaconda3 software instead of the ubuntu-provided python3! The former is the one used by JupyterHub and is up to date, while the latter may use outdated packages (in particular the NumPy package is too old!).

To submit the job simply run the following command in the terminal.

```
$ sbatch ising_job.sh
```

If the job does not finish (almost) instantaneously you will see it in the job queue.

```
$ squeue -u yourusername
             JOBID PARTITION     NAME     USER ST       TIME  NODES␣
 ↪NODELIST(REASON)
           3785215   hefstud isingjob username  R       0:10      1 cn96
```

This means that the scheduler has found a node (cn96) with sufficient resources to execute your job, but it can also happen that the job remains queued until resources are available. The job will disappear as soon as your script has finished. If necessary you can kill the job before it is completed by calling `scancel` with the job id as parameter.

```
$ scancel 3785215
```

9.2.3 Running multiple jobs

Most often you will want to run many jobs with the same script, but each with different parameters. For this you could produce many bash scripts (possibly in an automated fashion), but there are other options as well. If you wish to run the scripts sequentially, you can simply use a for loop in the bash script. For instance, the following will execute the script for temperatures 1.5, 1.6, …, 3.5 one after the other.

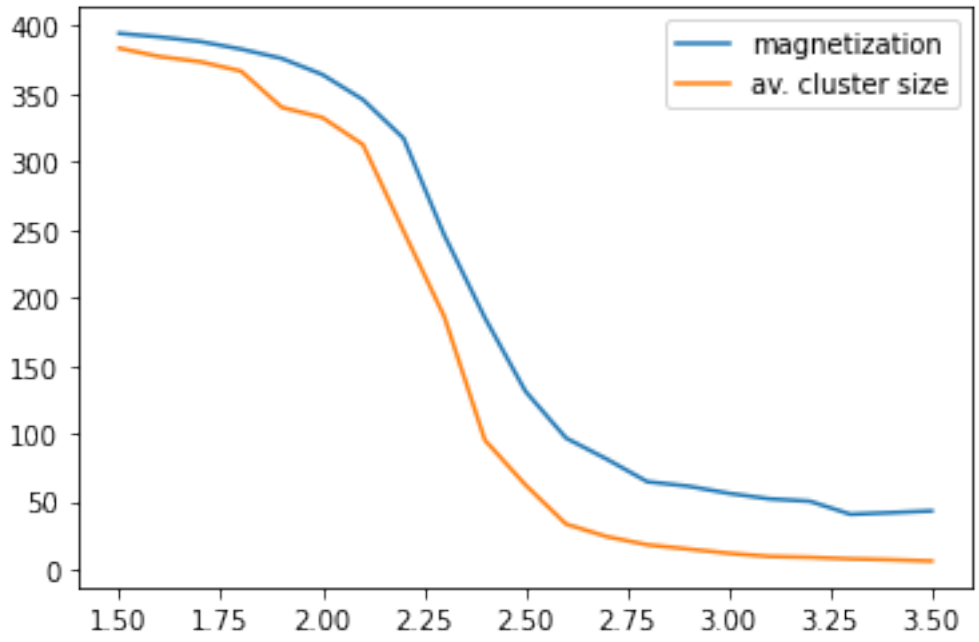
```
#!/bin/bash
#SBATCH --partition=hefstud
#SBATCH --output=std_%A_%a.txt
#SBATCH --mem=100M
#SBATCH --time=2:00:00
cd ~/monte-carlo-techniques-2021/lectures/ising_example
for temp in $(LANG=en_US seq 1.5 0.1 3.5)
do
        /software/anaconda3/bin/python3 ising.py -w 10 -t ${temp} -n 20
done
```

If you want to run several of these jobs simultaneously, you could use a **job array**. This is done by including an instruction like `#SBATCH --array=0-20%3` which tells SLURM to turn the job submissions into 21 individual jobs (with index 0,1,…,20), but never to run more than 3 at the same time. The index of the job is accessible as `$SLURM_ARRAY_TASK_ID` and can be used to determine the parameters for each individual job.

```
#!/bin/bash
#SBATCH --partition=hefstud
#SBATCH --output=std_%A_%a.txt
#SBATCH --mem=100M
#SBATCH --time=2:00:00
#SBATCH --array=0-20%3
cd ~/monte-carlo-techniques-2021/lectures/ising_example
temperatures=($(LANG=en_US seq 1.5 0.1 3.5))
temp=${temperatures[$SLURM_ARRAY_TASK_ID]}
/software/anaconda3/bin/python3 ising.py -w 20 -t ${temp} -n 500
```

We can then read all data files and perform analysis in this notebook:

```
import matplotlib.pylab as plt
%matplotlib inline

import os

def read_all_json(directory,startswith):
    data = []
    for filename in sorted(os.listdir(directory)):
        if filename.startswith(startswith) and filename.endswith(".json"):
            with open(os.path.join(directory,filename)) as file:
                data.append(json.load(file))
    return data

data = read_all_json("ising_example","data_w20_")

plt.plot([d['parameters']['t'] for d in data],[np.mean(np.abs(d['magnetizations
 ↪'])) for d in data])
plt.plot([d['parameters']['t'] for d in data],[d['average_cluster_size'] for d in␣
 ↪data])
plt.legend(["magnetization","av. cluster size"])
plt.show()
```



9.2.4 Remarks

Be considerate to other users. The department's compute nodes are shared between research staff and interns and students from this course. CPU resources and memory are limited. Feel free to occupy 1~3 cores full time. Or use more cores for short periods (hours) when the server load is low, but try at all times to leave resources available to others. Be prepared to kill your jobs, if requested by my colleagues or me.

Have your scripts write partial results to disk periodically. This way the job becomes as cancelable as possible and you have the opportunity to inspect progress. You want to prevent the situation that you have submitted a script that samples a million configurations for a dozen different model parameters and that only writes the results to a file at the very end. Perhaps the job takes more time than you anticipated: if an expected 12-hour job is not finished after two days and you don't know its progress, do you wait another day or kill the job? If for whatever reason you have to kill your script before it has finished, you could end up with no data at all. It is therefore advisable to store partial results periodically throughout the simulation.

Only write files in your home directory. The cluster nodes have local storage (scratch), but for the purposes of your project you will probably not need it. Given that scratch is not backed up and its contents differs from one node to the other, I would advise you to stick to your own home directory.

Part II

Exercises

Chapter 1

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
```

Exercise sheet 1

Code from the lecture:

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

rng = np.random.default_rng()
%matplotlib inline

def random_in_square():
    """Returns a random position in the square [-1,1)x[-1,1)."""
    return rng.uniform(-1,1,2)

def is_in_circle(x):
    return np.dot(x,x) < 1

def simulate_number_of_hits(N):
```

(continues on next page)

(continued from previous page)

```
    """Simulates number of hits in case of N trials in the pebble game."""
    number_hits = 0
    for i in range(N):
        position = random_in_square()
        if is_in_circle(position):
            number_hits += 1
    return number_hits

def random_in_disk():
    """Returns a uniform point in the unit disk via rejection."""
    position = random_in_square()
    while not is_in_circle(position):
        position = random_in_square()
    return position

def is_in_square(x):
    """Returns True if x is in the square (-1,1)^2."""
    return np.abs(x[0]) < 1 and np.abs(x[1]) < 1

def sample_next_position_naively(position,delta):
    """Keep trying a throw until it ends up in the square."""
    while True:
        next_position = position + delta*random_in_disk()
        if is_in_square(next_position):
            return next_position

def naive_markov_pebble(start,delta,N):
    """Simulates the number of hits in the naive Markov-chain version
    of the pebble game."""
    number_hits = 0
    position = start
    for i in range(N):
        position = sample_next_position_naively(position,delta)
        if is_in_circle(position):
            number_hits += 1
    return number_hits

def naive_markov_pebble_generator(start,delta,N):
    """Same as naive_markov_pebble but only yields the positions."""
    position = start
    for i in range(N):
        position = sample_next_position_naively(position,delta)
        yield position

def sample_next_position(position,delta):
    """Attempt a throw and reject when outside the square."""
    next_position = position + delta*random_in_disk()
    if is_in_square(next_position):
        return next_position  # accept!
    else:
        return position  # reject!

def markov_pebble(start,delta,N):
    """Simulates the number of hits in the proper Markov-chain version of the␣
↪pebble game."""
    number_hits = 0
    position = start
    for i in range(N):
        position = sample_next_position(position,delta)
        if is_in_circle(position):
            number_hits += 1
```

(continues on next page)

(continued from previous page)

```
    return number_hits

def markov_pebble_generator(start,delta,N):
    """Same as markov_pebble but only yields the positions."""
    position = start
    for i in range(N):
        position = sample_next_position(position,delta)
        yield position
```

1.1 Empirical convergence rate in the pebble game

(a) Write a function `pi_stddev` that estimates the standard deviation σ of the π-estimate using n trials by running the direct-sampling pebble game m times. Store this data for $n = 2^4, 2^5, \ldots, 2^{14}$ and $m = 200$ in an array. *Hint*: you may use the NumPy function `np.std`. **(12 pts)**

```
def pi_stddev(n,m):
    """Estimate the standard deviation in the pi estimate in the case of n trials,
    based on m runs of the direct-sampling pebble game."""
    # YOUR CODE HERE
    raise NotImplementedError()

stddev_data = np.array([[2**k,pi_stddev(2**k,200)] for k in range(4,12)])
stddev_data
```

```
# If done correctly, your code should pass the following tests
from nose.tools import assert_almost_equal
assert_almost_equal(pi_stddev(2**3,1000),0.582,delta=0.03)
assert_almost_equal(pi_stddev(2**4,1000),0.411,delta=0.03)
```

(b) Write a function `fit_power_law` that takes an array of (n, σ) pairs and determines best-fit parameters a, p for the curve $\sigma = an^p$. This is best done by fitting a straight line to the data on a log-log scale ($\log \sigma = \log a + p \log n$). *Hint*: use `curve_fit` from SciPy. **(12 pts)**

```
def fit_power_law(stddev_data):
    """Compute the best fit parameters a and p."""
    # YOUR CODE HERE
    raise NotImplementedError()
    return a_fit, p_fit
```

```
from nose.tools import assert_almost_equal
assert_almost_equal(fit_power_law(np.array([[16.0,1.4],[32.0,1.1],[64.0,0.
↪9]]))[0],3.36,delta=0.05)
assert_almost_equal(fit_power_law(np.array([[16.0,1.4],[32.0,1.1],[64.0,0.
↪9]]))[1],-0.31,delta=0.03)
```

(c) Plot the data against the fitted curve $\sigma = an^p$ in a log-log plot. Don't forget to properly label your axes. *Hint*: use `loglog` from matplotlib. **(12 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

1.2 Volume of a unit ball in other dimensions

In this exercise you will adapt the direct-sampling Monte Carlo method of the pebble game to higher dimensions to estimate the d-dimensional volume of the d-dimensional ball of radius 1.

(a) Adapt the direct-sampling Monte Carlo method to the pebble game in a d-dimensional hypercube $(-1,1)^d$ with an inscribed d-dimensional unit ball. **(12 pts)**

```
def number_of_hits_in_d_dimensions(N,d):
    """Simulates number of hits in case of N trials in the d-dimensional direct-
 ↪sampling pebble game."""
    # YOUR CODE HERE
    raise NotImplementedError()
    return number_hits
```

```
from nose.tools import assert_almost_equal
assert_almost_equal(number_of_hits_in_d_dimensions(100,1),100,delta=1)
assert_almost_equal(number_of_hits_in_d_dimensions(2000,3),1045,delta=50)
```

(b) Compare your estimates for the volume V_d of the d-dimensional unit ball for $N = 10000$ trials and $d = 1, 2, \ldots, 7$ to the exact formula $V_d = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)}$ in a plot. *Hint*: the Gamma function is available in `scipy.special.gamma`. **(12 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

1.3 Efficiency of the Metropolis algorithm and the 1/2-rule

In the lecture we mentioned the "1/2 rule" for Metropolis algorithms: if moves are rarely rejected then typically you do not explore the domain efficiently, while if most moves are rejected you are wasting efforts proposing moves. A good rule of thumb then is to aim for a rejection rate of $1/2$. In this exercise you are asked to test whether this rule of thumb makes any sense for the Markov-chain pebble game by varying the throwing range δ.

(a) Estimate the mean square deviation $\mathbb{E}[(\frac{4\text{hits}}{\text{trials}} - \pi)^2]$ from π for different values of δ ranging between 0 and 3, but fixed number of trials (say, 2000). For a decent estimate of the mean you will need at least 100 repetitions. **(14 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

(b) Measure the rejection rate for the same simulations as in (a). **(14 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

(c) Plot both the mean square deviation and the rejection rate as function of δ. How well does the 1/2 rule apply in this situation? **(12 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 2

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 2

Code from the lecture:

```
import numpy as np
import matplotlib.pylab as plt
from scipy.integrate import quad

rng = np.random.default_rng()
%matplotlib inline

def inversion_sample(f_inverse):
    '''Obtain an inversion sample based on the inverse-CDF f_inverse.'''
    return f_inverse(rng.random())
```

(continues on next page)

(continued from previous page)

```
def compare_plot(samples,pdf,xmin,xmax,bins):
    '''Draw a plot comparing the histogram of the samples to the expectation␣
↪coming from the pdf.'''
    xval = np.linspace(xmin,xmax,bins+1)
    binsize = (xmax-xmin)/bins
    # Calculate the expected numbers by numerical integration of the pdf over the␣
↪bins
    expected = np.array([quad(pdf,xval[i],xval[i+1])[0] for i in range(bins)])/
↪binsize
    measured = np.histogram(samples,bins,(xmin,xmax))[0]/(len(samples)*binsize)
    plt.plot(xval,np.append(expected,expected[-1]),"-k",drawstyle="steps-post")
    plt.bar((xval[:-1]+xval[1:])/2,measured,width=binsize)
    plt.xlim(xmin,xmax)
    plt.legend(["expected","histogram"])
    plt.show()

def gaussian(x):
    return np.exp(-x*x/2)/np.sqrt(2*np.pi)
```

2.1 Sampling random variables via the inversion method

(35 Points)

Recall from the lecture that for any real random variable X we can construct an explicit random variable via the inversion method that is identically distributed. This random variable is given by $F_X^{-1}(U)$ where F_X is the CDF of X and U is a uniform random variable on $(0,1)$ and

$$F_X^{-1}(p) := \inf\{x \in \mathbb{R} : F_X(x) \geq p\}.$$

This gives a very general way of sampling X in a computer program, as you will find out in this exercise.

(a) Let X be an **exponential random variable** with **rate** λ, i.e. a continuous random variable with probability density function $f_X(x) = \lambda e^{-\lambda x}$ for $x > 0$. Write a function `f_inverse_exponential` that computes $F_X^{-1}(p)$. Illustrate the corresponding sampler with the help of the function `compare_plot` above. **(10 pts)**

YOUR ANSWER HERE

```
def f_inv_exponential(lam,p):
    # YOUR CODE HERE
    raise NotImplementedError()

# plotting
# YOUR CODE HERE
raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert_almost_equal(f_inv_exponential(1.0,0.6),0.916,delta=0.001)
assert_almost_equal(f_inv_exponential(0.3,0.2),0.743,delta=0.001)
```

(b) Let now X have the **Pareto distribution** of **shape** $\alpha > 0$ on (b,∞), which has probability density function $f_X(x) = \alpha b^\alpha x^{-\alpha-1}$ for $x > b$. Write a function `f_inv_pareto` that computes $F_X^{-1}(p)$. Compare a histogram with a plot of $f_X(x)$ to verify your function numerically. **(10 pts)**

YOUR ANSWER HERE

```
### Solution
def f_inv_pareto(alpha,b,p):
    # YOUR CODE HERE
    raise NotImplementedError()

# plotting
# YOUR CODE HERE
raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert_almost_equal(f_inv_pareto(1.0,1.5,0.6),3.75,delta=0.0001)
assert_almost_equal(f_inv_pareto(2.0,2.25,0.3),2.689,delta=0.001)
```

(c) Let X be a discrete random variable taking values in $\{1, 2, \dots, n\}$. Write a Python function `f_inv_discrete` that takes the probability mass function p_X as a list `prob_list` given by $[p_X(1), \dots, p_X(n)]$ and returns a random sample with the distribution of X using the inversion method. Verify the working of your function numerically on an example. **(15 pts)**

```
def f_inv_discrete(prob_list,p):
    # YOUR CODE HERE
    raise NotImplementedError()

# plotting
# YOUR CODE HERE
raise NotImplementedError()
```

```
assert f_inv_discrete([0.5,0.5],0.4)==1
assert f_inv_discrete([0.5,0.5],0.8)==2
assert f_inv_discrete([0,0,1],0.1)==3
```

2.2 Central limit theorem?

(35 Points)

In this exercise we will have a closer look at central limits of the Pareto distribution, for which you implemented a random sampler in the previous exercise. By performing the appropriate integrals it is straightforward to show that

$$\mathbb{E}[X] = \begin{cases} \infty & \text{for } \alpha \leq 1 \\ \frac{\alpha b}{\alpha - 1} & \text{for } \alpha > 1 \end{cases}, \qquad \mathrm{Var}(X) = \begin{cases} \infty & \text{for } \alpha \leq 2 \\ \frac{\alpha b^2}{(\alpha-1)^2(\alpha-2)} & \text{for } \alpha > 2 \end{cases}.$$

This shows in particular that the distribution is **heavy tailed**, in the sense that some moments $\mathbb{E}[X^k]$ diverge.

(a) Write a function `sample_Zn` that produces a random sample for $Z_n = \frac{\sqrt{n}}{\sigma_X}(\bar{X}_n - \mathbb{E}[X])$ given $\alpha > 2$, $b > 0$ and $n \geq 1$. Visually verify the central limit theorem for $\alpha = 4$, $b = 1$ and $n = 1000$ by comparing a histogram of Z_n to the standard normal distribution (you may use `compare_plot`). **(10 pts)**

```
def sample_Zn(alpha,b,n):
    # YOUR CODE HERE
    raise NotImplementedError()

# Plotting
# YOUR CODE HERE
raise NotImplementedError()
```

```
assert_almost_equal(np.mean([sample_Zn(3.5,2.1,100) for _ in range(100)]),0,
 ↪delta=0.3)
assert_almost_equal(np.std([sample_Zn(3.5,2.1,100) for _ in range(100)]),1,
 ↪delta=0.3)
```

(b) Now take $\alpha = 3/2$ and $b = 1$. With some work (which you do not have to do) one can show that the characteristic function of X admits the following expansion around $t = 0$,

$$\varphi_X(t) = 1 + 3it - (|t| + it)\sqrt{2\pi|t|} + O(t^2).$$

Based on this, prove the **generalized CLT** for this particular distribution X which states that $Z_n = c\, n^{1/3}(\bar{X}_n - \mathbb{E}[X])$ in the limit $n \to \infty$ converges in distribution, with a to-be-determined choice of overall constant c, to a limiting random variable $\mathcal{S}$ with characteristic function

$$\varphi_{\mathcal{S}}(t) = \exp\big(-(|t| + it)\sqrt{|t|}\big).$$

(15 pts)

YOUR ANSWER HERE

(c) The random variable $\mathcal{S}$ has a stable Lévy distribution with index $\alpha = 3/2$ and skewness $\beta = 1$. Its probability density function $f_{\mathcal{S}}(x)$ does not admit a simple expression, but can be accessed numerically using SciPy's `scipy.stats.levy_stable.pdf(x,1.5,1.0)`. Verify numerically that the generalized CLT of part (b) holds by comparing an appropriate histogram to this PDF. **(10 pts)**

```
from scipy.stats import levy_stable

# YOUR CODE HERE
raise NotImplementedError()
```

2.3 Joint probability density functions and sampling the normal distribution

(30 Points)

Let Φ be a uniform random variable on $(0, 2\pi)$ and R an independent continuous random variable with probability density function $f_R(r) = r\, e^{-r^2/2}$ for $r > 0$. Set $X = R\cos\Phi$ and $Y = R\sin\Phi$. This is called the **Box-Muller transform**.

(a) Since Φ and R are independent, the joint probability density of Φ and R is $f_{\Phi,R}(\phi, r) = f_\Phi(\phi) f_R(r) = \frac{1}{2\pi}\, r\, e^{-r^2/2}$. Show by change of variables that X and Y are also independent and both distributed as a standard normal distribution $\mathcal{N}$. **(15 pts)**

YOUR ANSWER HERE

(b) Write a function to sample a pair of independent normal random variables using the Box-Muller transform. Hint: to sample R you can use the inversion method of the first exercise. Produce a histogram to check the distribution of your normal variables. **(15 pts)**

```
def random_normal_pair():
    '''Return two independent normal random variables.'''
    # YOUR CODE HERE
    raise NotImplementedError()
    return x, y

# Plotting
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 3

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 3

Code from the lectures:

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

rng = np.random.default_rng()
%matplotlib inline

def sample_acceptance_rejection(sample_z,accept_probability):
    while True:
        x = sample_z()
        if rng.random() < accept_probability(x):
```

(continues on next page)

(continued from previous page)

```
            return x

def estimate_expectation(sampler,n):
    '''Compute beste estimate of mean and 1-sigma error with n samples.'''
    samples = [sampler() for _ in range(n)]
    return np.mean(samples), np.std(samples)/np.sqrt(n-1)

def estimate_expectation_one_pass(sampler,n):
    sample_mean = sample_square_dev = 0.0
    for k in range(1,n+1):
        delta = sampler() - sample_mean
        sample_mean += delta / k
        sample_square_dev += (k-1)*delta*delta/k
    return sample_mean, np.sqrt(sample_square_dev / (n*(n-1)))
```

3.1 Acceptance-rejection sampling

(35 points)

The goal of this exercise is to develop a fast sampling algorithm of the discrete random variable X with probability mass function $$p_X(k) = \frac{6}{\pi^2}k^{-2}, \qquad k = 1, 2, ...$$

(a) Let Z be the discrete random variable with $p_Z(k) = \frac{1}{k} - \frac{1}{k+1}$ for $k = 1, 2, ...$ Write a function to compute the inverse CDF $F_Z^{-1}(u)$, such that you can use the inversion method to sample Z efficiently. **(15 pts)**

```
def f_inverse_Z(u):
    '''Compute the inverse CDF of Z, i.e. F_Z^{-1}(u) for 0 <= u <= 1.'''
    # YOUR CODE HERE
    raise NotImplementedError()

def random_Z():
    return int(f_inverse_Z(rng.random())) # make sure to return an integer
```

```
assert f_inverse_Z(0.2)==1
assert f_inverse_Z(0.51)==2
assert f_inverse_Z(0.76)==4
assert f_inverse_Z(0.991)==111
```

(b) Implement a sampler for X using acceptance-rejection based on the sampler of Z. For this you need to first determine a c such that $p_X(k) \leq c\, p_Z(k)$ for all $k = 1, 2, ...$, and then consider an acceptance probability $p_X(k)/(cp_Z(k))$. Verify the validity of your sampler numerically (e.g. for $k = 1, \ldots, 10$). **(20 pts)**

```
def accept_probability_X(k):
    '''Return the appropriate acceptance probability on the event Z=k.'''
    # YOUR CODE HERE
    raise NotImplementedError()

def random_X():
    return sample_acceptance_rejection(random_Z,accept_probability_X)

# Verify numerically
# YOUR CODE HERE
raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert min([random_X() for _ in range(10000)]) >= 1
```

(continues on next page)

(continued from previous page)

```
assert_almost_equal([random_X() for _ in range(10000)].count(1),6079,delta=400)
assert_almost_equal([random_X() for _ in range(10000)].count(3),675,delta=75)
```

3.2 Monte Carlo integration & Importance sampling

(30 Points)

Consider the integral

$$I = \int_0^1 \sin(\pi x(1-x))\mathrm{d}x = \mathbb{E}[X], \quad X = g(U), \quad g(U) = \sin(\pi U(1-U)),$$

where U is a uniform random variable in $(0, 1)$.

(a) Use Monte Carlo integration based on sampling U to estimate I with 1σ error at most 0.001. How many samples do you need? (It is not necessary to automate this: trial and error is sufficient.) **(10 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

(b) Choose a random variable Z on $(0, 1)$ whose density resembles the integrand of I and which you know how to sample efficiently (by inversion method, acceptance-rejection, or a built-in Python function). Estimate I again using importance sampling, i.e. $I = \mathbb{E}[X']$ where $X' = g(Z)f_U(Z)/f_Z(Z)$, with an error of at most 0.001. How many samples did you need this time? **(20 pts)**

```
def sample_nice_Z():
    '''Sample from the nice distribution Z'''
    # YOUR CODE HERE
    raise NotImplementedError()

def sample_X_prime():
    '''Sample from X'.'''
    # YOUR CODE HERE
    raise NotImplementedError()

# YOUR CODE HERE
raise NotImplementedError()
```

3.3 Direct sampling of Dyck paths

(35 points)

Direct sampling of random variables in high dimensions requires some luck and/or ingenuity. Here is an example of a probability distribution on $\mathbb{Z}^{2n+1}$ that features prominently in the combinatorial literature and can be sampled directly in an efficient manner. A sequence $\mathbf{x} \equiv (x_0, x_1, \ldots, x_{2n}) \in \mathbb{Z}^{2n+1}$ is said to be a **Dyck path** if $x_0 = x_{2n} = 0, x_i \geq 0$ and $|x_i - x_{i-1}| = 1$ for all $i = 1, \ldots, 2n$. Dyck paths are counted by the Catalan numbers $C(n) = \frac{1}{n+1}\binom{2n}{n}$. Let $\mathbf{X} = (X_0, \ldots, X_n)$ be a **uniform Dyck path**, i.e. a random variable with probability mass function $p_{\mathbf{X}}(\mathbf{x}) = 1/C(n)$ for every Dyck path $\mathbf{x}$. Here is one way to sample $\mathbf{X}$.

```
def random_dyck_path(n):
    '''Returns a uniform Dyck path of length 2n as an array [x_0, x_1, ..., x_{2n}
↪] of length 2n.'''
    # produce a (2n+1)-step random walk from 0 to -1
    increments = [1]*n +[-1]*(n+1)
```

(continues on next page)

(continued from previous page)

```
    rng.shuffle(increments)
    unconstrained_walk = np.cumsum(increments)
    # determine the first time it reaches its minimum
    argmin = np.argmin(unconstrained_walk)
    # cyclically permute the increments to ensure walk stays non-negative until␣
↪last step
    rotated_increments = np.roll(increments,-argmin)
    # turn off the superfluous -1 step
    rotated_increments[0] = 0
    # produce dyck path from increments
    walk = np.cumsum(rotated_increments)
    return walk


plt.plot(random_dyck_path(50))
plt.show()
```

(a) Let H be the (maximal) height of X, i.e. $H = \max_i X_i$. Estimate the expected height $\mathbb{E}[H]$ for $n = 2^5, 2^6, \ldots, 2^{11}$ (including error bars). Determine the growth $\mathbb{E}[H] \approx a\, n^\beta$ via an appropriate fit. *Hint*: use the `scipy.optimize.curve_fit` function with the option `sigma = ...` to incorporate the standard errors on $\mathbb{E}[H]$ in the fit. Note that when you supply the errors appropriately, fitting on linear or logarithmic scale should result in the same answer. **(25 pts)**

```
# Collect height estimates
n_values = [2**k for k in range(5,11+1)]
# YOUR CODE HERE
raise NotImplementedError()
```

```
from scipy.optimize import curve_fit

# Fitting
# YOUR CODE HERE
raise NotImplementedError()
print("Fit parameters: a = {}, beta = {}".format(a_fit,beta_fit))
```

```
# Plotting
# YOUR CODE HERE
raise NotImplementedError()
```

(b) Produce a histogram of the height $H/\sqrt{n}$ for $n = 2^5, 2^6, \ldots, 2^{11}$ and 3000 samples each and demonstrate with a plot that it appears to converge in distribution as $n \to \infty$. *Hint*: you could call `plt.hist(..., density=True,histtype='step')` for each n to plot the densities on top of each other. **(10 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 4

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 4

Code from the lectures:

```
import numpy as np
import matplotlib.pylab as plt
import networkx as nx

rng = np.random.default_rng()
%matplotlib inline

def draw_transition_graph(P):
    # construct a directed graph directly from the matrix
    graph = nx.DiGraph(P)
```

(continues on next page)

(continued from previous page)

```
    # draw it in such a way that edges in both directions are visible and have␣
 ↪appropriate width
    nx.draw_networkx(graph,connectionstyle='arc3, rad = 0.15',width=[6*P[u,v] for␣
 ↪u,v in graph.edges()])

def sample_next(P,current):
    return rng.choice(len(P),p=P[current])

def sample_chain(P,start,n):
    chain = [start]
    for _ in range(n):
        chain.append(sample_next(P,chain[-1]))
    return chain

def stationary_distributions(P):
    eigenvalues, eigenvectors = np.linalg.eig(np.transpose(P))
    # make list of normalized eigenvectors for which the eigenvalue is very close␣
 ↪to 1
    return [eigenvectors[:,i]/np.sum(eigenvectors[:,i]) for i in␣
 ↪range(len(eigenvalues))
              if np.abs(eigenvalues[i]-1) < 1e-10]

def markov_sample_mean(P,start,function,n):
    total = 0
    state = start
    for _ in range(n):
        state = sample_next(P,state)
        total += function[state]
    return total/n
```

4.1 Markov Chain on a graph

(50 points)

The goal of this exercise is to use Metropolis-Hastings to sample a uniform vertex in a (finite, undirected) connected graph G. More precisely, the state space $\Gamma = \{0, \dots, n-1\}$ is the set of vertices of a graph and the desired probability mass function is $\pi(x) = 1/n$ for $x \in \Gamma$. The set of edges is denoted $E = \{\{x_1, y_1\}, \dots, \{x_k, y_k\}\}$, $x_i, y_i \in \Gamma$, and we assume that there are no edges connecting a vertex with itself ($x_i \neq y_i$) and there is at most one edge between any pair of vertices. The **neighbors** of a vertex x are the vertices $y \neq x$ such that $\{x, y\} \in E$. The **degree** d_x of a vertex x is its number of neighbours. An example is the following graph:

```
edges = [(0,1),(1,2),(0,3),(1,4),(2,5),(3,4),(4,5),(3,6),(4,7),(5,8),(6,7),(7,8),
 ↪(5,7),(0,4)]
example_graph = nx.Graph(edges)
nx.draw(example_graph,with_labels=True)
```

A natural proposal transition matrix is $Q(x \to y) = \frac{1}{d_x} \mathbf{1}_{\{\{x,y\} \in E\}}$. In other words, when at x the proposed next state is chosen uniformly among its neighbors.

(a) Write a function `sample_proposal` that, given a (networkX) graph and node x, samples y according to transition matrix $Q(x \to y)$. *Hint*: a useful Graph member function is `neighbors`. **(10 pts)**

```
def sample_proposal(graph,x):
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert sample_proposal(nx.Graph([(0,1)]),0)==1
assert_almost_equal([sample_proposal(example_graph,3) for _ in range(1000)].
 ↪count(4),333,delta=50)
assert_almost_equal([sample_proposal(example_graph,8) for _ in range(1000)].
 ↪count(5),500,delta=60)
```

(b) Let us consider the Markov chain corresponding to the transition matrix $Q(x \to y)$. Produce a histogram of the states visited in the first ~20000 steps. Compare this to the exact stationary distribution found by the function `stationary_distributions` from the lecture applied to the transition matrix Q. *Hint*: another useful Graph member function is `degree`. **(15 pts)**

```
def chain_Q_histogram(graph,start,k):
    '''Produce a histogram (a Numpy array of length equal to the number of
    nodes of graph) of the states visited (excluding initial state) by the
    Q Markov chain in the first k steps when started at start.'''
    # YOUR CODE HERE
    raise NotImplementedError()

def transition_matrix_Q(graph):
    '''Construct transition matrix Q from graph as two-dimensional Numpy array.'''
    # YOUR CODE HERE
    raise NotImplementedError()

# Compare histogram and stationary distribution in a plot
# YOUR CODE HERE
raise NotImplementedError()
```

```
assert_almost_equal(transition_matrix_Q(example_graph)[3,4],1/3,delta=1e-9)
assert_almost_equal(transition_matrix_Q(example_graph)[3,7],0.0,delta=1e-9)
assert_almost_equal(transition_matrix_Q(example_graph)[2,2],0.0,delta=1e-9)
assert_almost_equal(np.sum(transition_matrix_Q(example_graph)[7]),1.0,delta=1e-9)
assert chain_Q_histogram(nx.Graph([(0,1)]),0,100)[1] == 50
assert len(chain_Q_histogram(example_graph,0,100)) == example_graph.number_of_
 ↪nodes()
```

(c) Determine the appropriate Metropolis-Hastings acceptance probability $A(x \to y)$ for $x \neq y$ and write a function that, given a graph and x, samples the next state with y according to the Metropolis-Hastings transition matrix $P(x \to y)$. **(10 pts)**

```
def acceptance_probability(graph,x,y):
    '''Compute A(x -> y) for the supplied graph (assuming x!=y).'''
    # YOUR CODE HERE
    raise NotImplementedError()

def sample_next_state(graph,x):
    '''Return next random state y according to MH transition matrix P(x -> y).'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
assert_almost_equal(acceptance_probability(example_graph,3,4),0.6,delta=1e-9)
assert_almost_equal(acceptance_probability(example_graph,8,7),0.5,delta=1e-9)
assert_almost_equal(acceptance_probability(example_graph,7,8),1.0,delta=1e-9)
assert_almost_equal(acceptance_probability(nx.Graph([(0,1)]),0,1),1,delta=1e-9)
```

(d) Do the same as in part (b) but now for the Markov chain corresponding to P. Verify that the histogram of the Markov chain approaches a flat distribution and corroborate this by calculating the explicit matrix P and applying `stationary_distributions` to it. *Hint*: for determining the explicit matrix $P(x \to y)$, remember that the formula $P(x \to y) = Q(x \to y)A(x \to y)$ only holds for $x \neq y$. What is $P(x \to x)$? **(15 pts)**

```
def chain_P_histogram(graph,start,n):
    '''Produce a histogram of the states visited (excluding initial state)
    by the P Markov chain in the first n steps when started at start.'''
    # YOUR CODE HERE
    raise NotImplementedError()

def transition_matrix_P(graph):
    '''Construct transition matrix Q from graph as numpy array.'''
    # YOUR CODE HERE
    raise NotImplementedError()

# plotting
# YOUR CODE HERE
raise NotImplementedError()
```

```
assert_almost_equal(transition_matrix_P(example_graph)[3,4],1/5,delta=1e-9)
assert_almost_equal(transition_matrix_P(example_graph)[3,7],0.0,delta=1e-9)
assert_almost_equal(transition_matrix_P(example_graph)[2,2],0.41666666,delta=1e-5)
assert_almost_equal(np.sum(transition_matrix_P(example_graph)[7]),1.0,delta=1e-9)
```

```
assert len(chain_P_histogram(example_graph,0,100)) == example_graph.number_of_
↪nodes()
assert_almost_equal(chain_P_histogram(example_graph,0,20000)[8],2222,delta=180)
```

4.2 MCMC simulation of disk model

(50 points)

Recall that in the disk model with we would like to sample the positions $x = (x_1, y_1, \ldots, x_N, y_N) \in [0, L)^{2N}$ of N disks of radius 1 in the torus $[0, L)^2$ with uniform density $\pi(x) = \mathbf{1}_{\{\text{all pairwise distance} \geq 2\}}(x)/Z$, where Z is the unknown partition function of the model. We will assume $L > 2$ and $N \geq 1$. For the purposes of this simulation we will store the state x in a `np.array` of dimension $(N, 2)$ with values in $[0, L)$. Such a configuration can be conveniently plotted using the following function:

```
def plot_disk_configuration(positions,L):
    fig,ax = plt.subplots()
    ax.set_aspect('equal')
    ax.set_ylim(0,L)
    ax.set_xlim(0,L)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])
    for x,y in positions:
        # consider all horizontal and vertical copies that may be visible
        for x_shift in [z for z in x + [-L,0,L] if -1<z<L+1]:
            for y_shift in [z for z in y + [-L,0,L] if -1<z<L+1]:
                ax.add_patch(plt.Circle((x_shift,y_shift),1))
    plt.show()

# Example with N=3 and L=5
positions = np.array([[0.1,0.5],[2.1,1.5],[3.2,3.4]])
plot_disk_configuration(positions,5)
```

(a) Write a function `two_disks_overlap` that tests whether disks at position $\mathbf{x}_1 \in [0, L)^2$ and position $\mathbf{x}_2 \in [0, L)^2$ overlap and a function `disk_config_valid` that checks whether a full configuration is valid (non-overlapping and non-touching). *Hint:* The minimal separation in the x-direction can be expressed as a function of

`x1[0]-x2[0]` and the minimal separation in the y-direction as a function of `x1[1]-x2[1]`. Then use pythagoras. **(15 pts)**

```
def two_disks_overlap(x1,x2,L):
    '''Return True if the disks centered at x1 and x2 (represented as 2-element
 ↪arrays) overlap in [0,L)^2.'''
    # YOUR CODE HERE
    raise NotImplementedError()

def disk_config_valid(x,L):
    '''Return True if the configuration x (as two-dimensional array) is non-
 ↪overlapping in [0,L)^2.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
assert two_disks_overlap(np.array([1,1]),np.array([1,1]),5)
assert two_disks_overlap(np.array([0.6,0.6]),np.array([4.1,0.5]),5)
assert two_disks_overlap(np.array([0.3,0.3]),np.array([4.6,4.6]),5)
assert not two_disks_overlap(np.array([1,1]),np.array([3.1,1]),7)
assert not two_disks_overlap(np.array([1,1]),np.array([1,3.1]),7)
assert not two_disks_overlap(np.array([1,1]),np.array([1.01+np.sqrt(2),1.01+np.
 ↪sqrt(2)]),6)
assert two_disks_overlap(np.array([1,1]),np.array([0.99+np.sqrt(2),0.99+np.
 ↪sqrt(2)]),6)
```

```
assert disk_config_valid(np.array([[0.1,0.5],[2.1,1.5],[3.2,3.4]]),5)
assert not disk_config_valid(np.array([[0.1,0.5],[2.1,1.5],[3.2,3.4],[4.1,2.3]]),
 ↪5)
assert disk_config_valid(np.array([[1,1],[3.1,1],[1,3.1]]),6)
assert not disk_config_valid(np.array([[1,1],[3.1,1],[1,3.1],[2.5,2.5]]),6)
```

(b) Assuming $N \leq \lceil \frac{1}{2}L - 1 \rceil^2$ where $\lceil r \rceil$ is the smallest integer larger or equal to r, write a function `generate_initial_positions` that produces an arbitrary non-overlapping (and non-touching) initial condition given N and L. The layout need not be random, any deterministic layout is ok (e.g. grid). **(10 pts)**

```
def generate_initial_positions(N,L):
    '''Return array of positions of N disks in non-overlapping positions.'''
    # YOUR CODE HERE
    raise NotImplementedError()

plot_disk_configuration(generate_initial_positions(33,14.5),14.5)
```

```
for l in [4,9.2,14.5]:
    for n in range(1,int(np.ceil(l/2-1)**2)+1):
        assert disk_config_valid(generate_initial_positions(n,l),l), "Failed for
 ↪n = {}, l = {}".format(n,l)
```

(c) Write a function `remains_valid_after_move` that determines whether in a non-overlapping configuration x moving the ith disk to `next_position` results in a valid non-overlapping configuration. **(10 pts)**

```
def remains_valid_after_move(x,i,next_position,L):
    '''Returns True if replacing x[i] by next_position would yield a valid
 ↪configuration,
    otherwise False.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
assert remains_valid_after_move([[0.1,0.5],[2.1,1.5],[3.2,3.4]],0,[4.5,0.5],5)
assert not remains_valid_after_move([[0.1,0.5],[2.1,1.5],[3.2,3.4]],1,[4.5,0.5],5)
assert not remains_valid_after_move([[0.1,0.5],[2.1,1.5],[3.2,3.4]],2,[3.2,2.5],5)
assert remains_valid_after_move([[0.1,0.5],[2.1,1.5],[3.2,3.4]],2,[3.2,3.8],5)
```

(d) Implement the Metropolis-Hastings transition by selecting a uniformly chosen disk and displacing it by $(\delta\,\mathcal{N}_1, \delta\,\mathcal{N}_2)$ where $\delta > 0$ is a parameter and $\mathcal{N}_i$ are independent normal random variables (make sure to keep positions within $[0, L)^2$ by taking the new position modulo L). Test run your simulation for $L = 11.3$ and $N = 20$ and $\delta = 0.3$ and about 10000 Markov chain steps and plot the final state. **(15 pts)**

```
def MH_disk_move(x,L,delta):
    '''Perform random MH move on configuration x, thus changing the array x (if␣
 ↪accepted).
    Return True if move was accepted, False otherwise.'''
    # YOUR CODE HERE
    raise NotImplementedError()

# Test run and plot resulting configuration
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 5

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 5

Code from the lectures:

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

rng = np.random.default_rng()
%matplotlib inline

def sample_autocovariance(x,tmax):
    '''Compute the sample autocorrelation of the time series x
    for t = 0,1,...,tmax-1.'''
    x_shifted = x - np.mean(x)
```

(continues on next page)

(continued from previous page)

```
    return np.array([np.dot(x_shifted[:len(x)-t],x_shifted[t:])/(len(x)-t)
                     for t in range(tmax)])

def find_correlation_time(autocov):
    '''Return the index of the first entry that is smaller than autocov[0]/e or␣
 ↪the length of autocov if none are smaller.'''
    smaller = np.where(autocov < np.exp(-1)*autocov[0])[0]
    return smaller[0] if len(smaller) > 0 else len(autocov)

# Ising model
def attempt_spin_flip_for_trace(config,boltzmannfactor):
    '''Perform Metropolis-Hastings transition on config and return
    change in magnetization and energy.'''
    w = len(config)
    i,j = rng.integers(0,w,2)
    neighbour_sum = config[i,j] * (config[(i+1)%w,j] + config[(i-1)%w,j] +
                                   config[i,(j+1)%w] + config[i,(j-1)%w])
    if neighbour_sum <= 0 or rng.random() < boltzmannfactor**neighbour_sum:
        config[i,j] = -config[i,j]
        return 2*config[i,j], 2*neighbour_sum
    else:
        return 0, 0

def compute_energy(config):
    '''Compute the energy H(s) of the state config (with J=1).'''
    h = 0
    w = len(config)
    for i in range(w):
        for j in range(w):
            h -= config[i,j] * (config[i,(j+1)%w] + config[(i+1)%w,j])
    return h

def compute_magnetization(config):
    '''Compute the magnetization M(s) of the state config.'''
    return np.sum(config)

def get_MCMC_trace(config,beta,n):
    '''Sample first n steps of the Markov chain and produce trace
    of magnetization and energy.'''
    boltz = np.exp(-2*beta)
    trace = np.zeros((n,2))
    # set the initial magnetization and energy ...
    m = compute_magnetization(config)
    h = compute_energy(config)
    for i in range(n):
        dm, dh = attempt_spin_flip_for_trace(config,boltz)
        # ... and update them after each transition
        m += dm
        h += dh
        trace[i][0] = m
        trace[i][1] = h
    return trace

def uniform_init_config(width):
    '''Produce a uniform random configuration.'''
    return 2*rng.integers(2,size=(width,width))-1

def aligned_init_config(width):
    '''Produce an all +1 configuration.'''
    return np.ones((width,width),dtype=int)
```

(continues on next page)

(continued from previous page)

```
def antialigned_init_config(width):
    '''Produce a checkerboard configuration'''
    if width % 2 == 0:
        return np.tile([[1,-1],[-1,1]],(width//2,width//2))
    else:
        return np.tile([[1,-1],[-1,1]],((width+1)//2,(width+1)//2))[:width,:width]

def plot_ising(config,ax,title):
    '''Plot the Ising configuration.'''
    ax.matshow(config, vmin=-1, vmax=1, cmap=plt.cm.binary)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])

def attempt_spin_flip(config,boltzmannfactor):
    '''Perform Metropolis-Hastings transition on config.'''
    w = len(config)
    i,j = rng.integers(0,w,2)
    neighbour_sum = config[i,j] * (config[(i+1)%w,j] + config[(i-1)%w,j] +
                                   config[i,(j+1)%w] + config[i,(j-1)%w])
    if neighbour_sum <= 0 or rng.random() < boltzmannfactor**neighbour_sum:
        config[i,j] = -config[i,j]

def run_ising_MCMC(config,beta,n):
    '''Perform n steps of the MH Markov chain on config.'''
    boltz = np.exp(-2*beta)
    for _ in range(n):
        attempt_spin_flip(config,boltz)

def batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each.
    Returns the mean and standard error.'''
    batches = np.reshape(data,(k,-1))
    values = np.apply_along_axis(observable, 1, batches)
    return np.mean(values), np.std(values)/np.sqrt(k-1)

def jackknife_batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each
    collection of all but one batches. Returns the mean and corrected
    standard error.'''
    batches = np.reshape(data,(k,-1))
    values = [observable(np.delete(batches,i,0).flatten()) for i in range(k)]
    return np.mean(values), np.sqrt(k-1)*np.std(values)
```

5.1 Storing and loading data in HDF5 files

(20 points)

While working on the previous exercises, you may have noticed that it can be impractical to have generated data only available during an active jupyter session. If you loose connection with the python kernel or wish to continue your calculation at a later time, there was no other option than to regenerate the data. From this week on, the computing time involved in generating data will generally increase, making these issues worse. Not just for you, but also for the (auto)grading of your exercise sheets. So we need a method to store and load data, which is the topic of this warm-up exercise. Of course, this will also be very relevant for managing your research project at the end of the course.

There are many different file formats that one can use to store data: CSV, JSON, YAML. We will make use of a file format that is well adapted to handling and organizing (large) numerical data sets: HDF5. Python has a convenient

package h5py to read and write hdf5-files. The basic usage is demonstrated in the following code snippet, that stores randomly generated data in a file `test.hdf5`, which will appear in the same directory as this notebook, and reads it again.

```
import h5py

# generate some floating point test data as Numpy array
testarray = rng.random(1000)
print("testarray[325] =",testarray[325])

# open test.hdf5 for writing (note that if the file already exists, it will be␣
↪overwritten)
with h5py.File('test.hdf5','w') as f:
    # add the test data as a new data set to the file
    f.create_dataset("test",data=testarray)

# open test.hdf5 for reading
with h5py.File('test.hdf5','r') as f:
    # we can directly access some numbers
    print('f["test"][325] =',f["test"][325])
    # or retrieve the full Numpy array
    test = f["test"][()]
    print('test[325] =',test[325])

# open test.hdf5 for updating (creating an empty file if it does not yet exist)
with h5py.File('test.hdf5','a') as f:
    # add another data set
    f.create_dataset("test-plus-one",data=testarray+1)
    print('f["test-plus-one"][325] =',f["test-plus-one"][325])

# open test.hdf5 for reading
with h5py.File('test.hdf5','r') as f:
    print('f["test"][325] =',f["test"][325])
    print('f["test-plus-one"][325] =',f["test-plus-one"][325])

# we can delete a data set from the file as follows
with h5py.File('test.hdf5','a') as f:
    print( "data sets in test.hdf5:", list(f.keys()) )
    del f["test"]
    print( "data sets in test.hdf5:", list(f.keys()) )
```

(a) Write code that opens the file `test.hdf5` and checks whether the data set `random-walk` is present. If not, generate a random walk $X_0 = 0, X_1, \dots, X_{1000}$ with 1000 steps starting at 0 with increments that are distibuted as a standard normal random variables, and store $X_0, X_1, \dots, X_{1000}$ in the data set `random-walk`. **(10 pts)**

```
# When debugging it is useful to erase the data first:
# with h5py.File('test.hdf5','a') as f:
#     if "random-walk" in f:
#         del f["random-walk"]

# YOUR CODE HERE
raise NotImplementedError()
```

```
with h5py.File('test.hdf5','r') as f:
    assert "random-walk" in f
    assert len(f["random-walk"]) == 1001
```

(b) Write code that reads the file `test.hdf5` and produces a plot of the random walk, so X_i versus i for $i = 0, \dots, 1000$. **(10 pts)**

```
# YOUR CODE HERE
raise NotImplementedError()
```

5.2 Exploring the 2D Ising model

(80 points)

The goal of this exercise is to get a robust measurement of the magnetization with error bars of the 2D Ising model on a 12×12 grid and the following range of temperatures $T \in [1, 4]$. We will work with units where $\beta = 1/T$ (so $k_B = 1$).

```
temperatures = np.linspace(1.5,3.5,21)
width = 12
nsites = width*width
print(temperatures)
```

(a) The first goal is to estimate equilibration times. Write a function `time_until_magnetizations_converge` that starts two Markov chains X_i^a and X_i^u based on the Metropolis-Hastings spin flip algorithm, one with fully aligned initial state X_0^a and one with uniform random initial state X_0^u. Return the first time t when the absolute magnetization of the first drops below that of the second, i.e. $t = \inf\{i : |M(X_i^a)| < |M(X_i^u)|\}$. *Hint*: you may use `attempt_spin_flip_for_trace`. **(10 pts)**

```
def time_until_magnetizations_converge(width,beta):
    '''Sample two Markov chains simultaneously, one started in an aligned state
    and one in a uniform one. Return the first time (in steps) at which the
    absolute magnetization of the first drops below the second.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert time_until_magnetizations_converge(3,0.2) > 0
assert_almost_equal(np.mean([time_until_magnetizations_converge(3,0.2) for _ in␣
↪range(500)]),17.7,delta=1.2)
assert_almost_equal(np.mean([time_until_magnetizations_converge(3,0.45) for _ in␣
↪range(500)]),59,delta=7)
```

(b) For each of the temperatures T and $w = 12$ collect the 10 the results of `time_until_magnetizations_converge` and store it in a data set named `"eq-times"` in the file `ising_data.hdf5` (as an array of dimension 21 x 10). Make a plot of the average time with error bars as function of the temperature, based on this data in `ising_data.hdf5`. **(15 pts)**

```
with h5py.File('ising_data.hdf5','a') as f:
    if not "eq-times" in f:
        # produce and store data
        # YOUR CODE HERE
        raise NotImplementedError()
```

```
with h5py.File('ising_data.hdf5','r') as f:
    # read data in produce plot
    # YOUR CODE HERE
    raise NotImplementedError()
```

(c) Write a function `estimate_ising_autocorrelation_time` that estimates the autocorrelation time $\tau_{|M|} = \inf\{t : \bar{\rho}(t) < 1/e\}$ of the absolute magnetization using q steps of equilibration, started from the uniform random initial state, followed by collecting a trace of n steps and a maximal time difference in the sample

autocovariance of $t = t_{\max}$. You may use the functions `get_MCMC_trace`, `sample_autocovariance` and `find_correlation_time`. **(10 pts)**

```
def estimate_ising_autocorrelation_time(width,beta,q,n,tmax):
    '''Estimate the autocorrelation time using q steps of equilibration followed
    by collecting a trace of n steps, calculating the autocovariance up to t=tmax
 ↪< n.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
assert_almost_equal(np.mean([estimate_ising_autocorrelation_time(3,0.2,100,400,
 ↪20) for _ in range(50)]),7.2,delta=0.7)
assert_almost_equal(np.mean([estimate_ising_autocorrelation_time(3,0.45,100,400,
 ↪40) for _ in range(50)]),15.2,delta=2.5)
```

(d) For each of the temperatures T and $w = 12$ collect the result of `estimate_ising_autocorrelation_time` using sensible equilibration (use your findings in part b) and trace length and store it in a data set named `"autocorr-times"` in the file `ising_data.hdf5` (as an array of dimension 21). Make a plot of the autocorrelation time (without error bars) as function of the temperature. **(15 pts)**

```
with h5py.File('ising_data.hdf5','a') as f:
    if not "autocorr-times" in f:
        # produce and store data
        # YOUR CODE HERE
        raise NotImplementedError()
```

```
with h5py.File('ising_data.hdf5','r') as f:
    # read and plot data
    # YOUR CODE HERE
    raise NotImplementedError()
```

(e) Finally we can get to our estimate of the mean absolute magnetization $\mathbb{E}[|M(X)|]$. Write a function `sample_magnetizations` that produces n magnetization measurements, using q equilibration sweeps and k sweeps between measurements (thinning). Store $n = 1000$ measurements for each temperature T in a data set `"magnetizations"` in the file `ising_data.hdf5` with appropriate choices for q and k. Plot the resulting estimate of $\mathbb{E}[|M(X)|]$ as function of T with error bars. You may use `run_ising_MCMC` and `compute_magnetization`, `batch_estimate`, `jackknife_batch_estimate`. **(20 pts)**

```
def sample_magnetizations(width,beta,q,k,n):
    '''Run the Ising model simulation with q equilibration sweeps, k sweeps␣
 ↪between measurements,
    and performing a total of n measurements of the absolute magnetization, which␣
 ↪are returned as
    NumPy array.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
assert_almost_equal(np.mean(sample_magnetizations(3,0.3,50,1,400)),5.72,delta=0.6)
assert_almost_equal(np.mean(sample_magnetizations(3,0.6,50,1,400)),8.72,delta=0.2)
```

```
with h5py.File('ising_data.hdf5','a') as f:
    if not "magnetizations" in f:
        # collect and store data
        # YOUR CODE HERE
        raise NotImplementedError()
```

```
with h5py.File('ising_data.hdf5','r') as f:
    # read and plot
    # YOUR CODE HERE
    raise NotImplementedError()
```

(f) Produce a plot with error bars of the magnetic susceptibility $\chi = \frac{\beta}{N} \mathrm{Var}(|M(X)|)$ as function of temperature, using the same data set. Apply the jackknife batch estimate for the errors. **(10 pts)**

```
with h5py.File('ising_data.hdf5','r') as f:
    # YOUR CODE HERE
    raise NotImplementedError()
```

Chapter 6

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 6

Code from the lecture

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

rng = np.random.default_rng()
%matplotlib inline

def aligned_init_config(width):
    '''Produce an all +1 configuration.'''
    return np.ones((width,width),dtype=int)
```

(continues on next page)

(continued from previous page)

```
def plot_ising(config,ax,title):
    '''Plot the configuration.'''
    ax.matshow(config, vmin=-1, vmax=1, cmap=plt.cm.binary)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])

from collections import deque

def neighboring_sites(s,w):
    '''Return the coordinates of the 4 sites adjacent to s on an w*w lattice.'''
    return [((s[0]+1)%w,s[1]),((s[0]-1)%w,s[1]),(s[0],(s[1]+1)%w),(s[0],(s[1]-1)
↪%w)]

def cluster_flip(state,seed,p_add):
    '''Perform a single Wolff cluster move with specified seed on the state with␣
↪parameter p_add.'''
    w = len(state)
    spin = state[seed]
    state[seed] = -spin
    cluster_size = 1
    unvisited = deque([seed])   # use a deque to efficiently track the unvisited␣
↪cluster sites
    while unvisited:   # while unvisited sites remain
        site = unvisited.pop()  # take one and remove from the unvisited list
        for nbr in neighboring_sites(site,w):
            if state[nbr] == spin and rng.uniform() < p_add:
                state[nbr] = -spin
                unvisited.appendleft(nbr)
                cluster_size += 1
    return cluster_size

def wolff_cluster_move(state,p_add):
    '''Perform a single Wolff cluster move on the state with addition probability␣
↪p_add.'''
    seed = tuple(rng.integers(0,len(state),2))
    return cluster_flip(state,seed,p_add)

def compute_magnetization(config):
    '''Compute the magnetization M(s) of the state config.'''
    return np.sum(config)

def run_ising_wolff_mcmc(state,p_add,n):
    '''Run n Wolff moves on state and return total number of spins flipped.'''
    total = 0
    for _ in range(n):
        total += wolff_cluster_move(state,p_add)
    return total

def sample_autocovariance(x,tmax):
    '''Compute the autocorrelation of the time series x for t = 0,1,...,tmax-1.'''
    x_shifted = x - np.mean(x)
    return np.array([np.dot(x_shifted[:len(x)-t],x_shifted[t:])/len(x)
                     for t in range(tmax)])

def find_correlation_time(autocov):
    '''Return the index of the first entry that is smaller than
    autocov[0]/e or the length of autocov if none are smaller.'''
    smaller = np.where(autocov < np.exp(-1)*autocov[0])[0]
```

(continues on next page)

(continued from previous page)

```
    return smaller[0] if len(smaller) > 0 else len(autocov)
```

6.1 MCMC simulation of the XY model

(100 Points)

Goal of this exercise: Practice implementing MCMC simulation of the XY spin model using an appropriate cluster algorithm and analyzing the numerical effectiveness.

The **XY model** is a relative of the Ising model in which the discrete ± 1 spin at each lattice site is replaced by a continuous 2-dimensional spin on the unit circle

$$S_1 = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 = 1\}.$$

To be precise, we consider a $w \times w$ lattice with periodic boundary conditions and a XY configuration $s = (s_1, \dots, s_N) \in \Gamma = S_1^N$, $N = w^2$, with Hamiltonian that is very similar to the Ising model,

$$H_{XY}(s) = -J \sum_{i \sim j} s_i \cdot s_j.$$

Here, as in the Ising model, the sum runs over nearest neighbor pairs i and j and $s_i \cdot s_j$ is the usual Euclidean inner product of the vectors $s_i, s_j \in S_1$. We will only consider the ferromagnetic XY model and set $J = 1$ in the remainder of the exercise. Note that nowhere in the definition the x- and y-components of the spins are related to the two directions of the lattice (one could also have studied the XY model on a one-dimensional or three-dimensional lattice and the spins would still have two components). As usual we are interested in sampling configurations $s \in \Gamma$ with distribution $\pi(s)$ given by the Boltzmann distribution

$$\pi(s) = \frac{1}{Z_{XY}} e^{-\beta H_{XY}(s)}, \qquad \beta = 1/T.$$

The XY model admits a (local) **cluster algorithm** that is very similar to the Wolff algorithm of the Ising model. It amounts to the following recipe:

1. Sample a uniform seed site i_{seed} in $1, \dots, N$ and an independent uniform unit vector $\hat{n} \in S_1$.
2. Grow a cluster C starting from the seed i_{seed} consisting only of sites j whose spin s_j is "aligned" with the seed, in the sense that $s_j \cdot \hat{n}$ has the same sign as $s_{i_{\text{seed}}} \cdot \hat{n}$, or $(s_j \cdot \hat{n})(s_{i_{\text{seed}}} \cdot \hat{n}) > 0$. Like in the Ising model this is done iteratively by examining the neighbors of sites that are already in the cluster, and adding those that are aligned with appropriate probability. The difference with the Ising model is that this probability depends on the spins s_i and s_j that are linked (meaning that s_j is an aligned neighbor of s_i) via the formula $\$p_{\text{add}}(s_i, s_j) = 1 - \exp\big(-2\beta\,(s_i \cdot \hat{n})(s_j \cdot \hat{n})\big).\$$
3. Once the cluster C is constructed, all of its spins are "flipped" in the sense that they are reflected in the plane perpendicular to $\hat{n}$, i.e. $s_j \to s_j - 2(s_j \cdot \hat{n})\hat{n}$.

(a) Verify by a calculation, to be included using markdown and LaTeX below, that the probabilities $p_{\text{add}}(s_i, s_j)$ are the appropriate ones to ensure detailed balance for the Boltzmann distribution $\pi(s)$. *Hint*: Follow the same reasoning as for the Ising model. Compare the probabilities involved in producing the cluster C in state s and state s'. Why do the probabilities only differ at the boundary edges in the cluster C? **(25 pts)**

YOUR ANSWER HERE

(b) In order to implement the cluster update described above, we take the state to be described by a Numpy array of dimension $(w, w, 2)$, for which we have already provided a function `xy_aligned_init_config` to generate an all-aligned initial state. Write the function `xy_cluster_flip`, that grows and flips a cluster starting from the given seed site and $\hat{n}$ and returns the cluster size, and `xy_cluster_move`, that performs the previous function to a random seed and direction $\hat{n}$ and also returns the cluster size. **(20 pts)**

```
def xy_aligned_init_config(width):
    '''Return an array of dimension (width,width,2) representing aligned spins in␣
 ↪x-direction.'''
    return np.dstack((np.ones((width,width)),np.zeros((width,width))))

def xy_cluster_flip(state,seed,nhat,beta):
    '''Perform a cluster move with specified seed and vector nhat on the state at␣
 ↪temperature beta.'''
    w = len(state)
    # YOUR CODE HERE
    raise NotImplementedError()
    return cluster_size

def xy_cluster_move(state,beta):
    '''Perform a single Wolff cluster move on the state with addition probability␣
 ↪p_add.'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert 1 <= xy_cluster_flip(xy_aligned_init_config(4),(0,0),np.array([np.cos(0.5),
 ↪np.sin(0.5)]),0.5) <= 16
assert_almost_equal(np.mean([xy_cluster_flip(xy_aligned_init_config(3),(0,0),
                                             np.array([np.cos(0.5),np.sin(0.5)]),
 ↪0.3)
                             for _ in range(200)]),5.3,delta=0.7)
assert_almost_equal(np.mean([xy_cluster_flip(xy_aligned_init_config(3),(1,2),
                                             np.array([np.cos(0.2),np.sin(0.2)]),
 ↪0.2)
                             for _ in range(200)]),4.3,delta=0.6)
```

```
assert 1 <= xy_cluster_move(xy_aligned_init_config(4),0.5) <= 16
assert_almost_equal(np.mean([xy_cluster_move(xy_aligned_init_config(3),0.3)
                             for _ in range(200)]),3.6,delta=0.75)
assert_almost_equal(np.mean([xy_cluster_move(xy_aligned_init_config(3),0.9)
                             for _ in range(200)]),6.3,delta=0.75)
```

(c) Estimate and plot the average cluster size in equilibrium for a 25x25 lattice ($w = 25$) for the range of temperatures $T = 0.5, 0.6, \ldots, 1.5$. It is not necessary first to estimate the equilibration time: you may start in a fully aligned state and use 400 moves for equilibration and 1000 for estimating the average cluster size. It is not necessary to estimate errors for this average. Store your averages in the data set `"cluster-size"` (an array of size 11) in the HDF5-file `xy_data.hdf5`, just like you did in Exercise sheet 5. Then read the data from file and produce a plot. **(20 pts)**

```
temperatures = np.linspace(0.5,1.5,11)
width = 25
equilibration_moves = 400
measurement_moves = 1000

# YOUR CODE HERE
raise NotImplementedError()
```

```
with h5py.File('xy_data.hdf5','r') as f:
    assert f["cluster-size"][()].shape == (11,)
    assert_almost_equal(f["cluster-size"][4],225,delta=40)
    assert_almost_equal(f["cluster-size"][10],8,delta=8)
```

```
# Plotting
# YOUR CODE HERE
```

(continues on next page)

(continued from previous page)

```
raise NotImplementedError()
```

(d) Make an MCMC estimate (and plot!) of the **mean square magnetization per spin** $\mathbb{E}[m^2(s)]$ for the same set of temperatures, where

$$m^2(s) = \left(\frac{1}{N}\sum_{i=1}^{N} s_i\right) \cdot \left(\frac{1}{N}\sum_{i=1}^{N} s_i\right).$$

To choose the equilibration time and time between measurement, use the average cluster size from (c) to estimate how many moves correspond to 1 *sweep*, i.e. roughly $N = w^2$ updates to spins. Then use 100 equilibration *sweeps* and 200 measurements of $m^2(s)$, with 2 *sweeps* between each measurement. Store the measured values of $m^2(s)$ in the data set `"square-magn"` of dimension $(11, 200)$ in `xy_data.hdf5`. Then read the data and plot estimates for $\mathbb{E}[m^2(s)]$ including errors (based on batching or jackknife). If the errors are too small to see, you may multiply them by some number and indicate this in the title of the plot. **(20 pts)**

```
measurements = 200
equil_sweeps = 100
measure_sweeps = 2

# YOUR CODE HERE
raise NotImplementedError()
```

```
with h5py.File('xy_data.hdf5','r') as f:
    assert f["square-magn"][()].shape == (11, 200)
    assert_almost_equal(np.mean(f["square-magn"][4]),0.456,delta=0.02)
    assert_almost_equal(np.mean(f["square-magn"][9]),0.023,delta=0.01)
```

```
# Plotting
# YOUR CODE HERE
raise NotImplementedError()
```

(e) Produce a single equilibrated state for each temperature and store them in the data set `"states"` of dimension $(11, 25, 25, 2)$ in `xy_data.hdf5`. Then read them and produce a table of plots using the provided function `plot_xy`, which shows colors based on the angle of the spin, each with a title to indicate the temperature. Can you observe the **Kosterlitz–Thouless transition** of the XY model? **(15 pts)**

```
width = 25
state = xy_aligned_init_config(width)
equil_sweeps = 200

# YOUR CODE HERE
raise NotImplementedError()
```

```
with h5py.File('xy_data.hdf5','r') as f:
    assert f["states"][()].shape == (11, 25, 25, 2)
```

```
def plot_xy(state,ax,title=""):
    '''Plot the XY configuration given by state. Takes an Axes object ax from
    matplotlib to draw to, and adds the specified title.'''
    angles = np.arctan2(*np.transpose(state,axes=(2,0,1)))
    ax.matshow(angles, vmin=-np.pi, vmax=np.pi, cmap=plt.cm.hsv)
    ax.title.set_text(title)
    ax.set_yticklabels([])
    ax.set_xticklabels([])
    ax.set_yticks([])
    ax.set_xticks([])
```

(continues on next page)

(continued from previous page)

```
# Make a table of plots
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 7

Exercise sheet

Some general remarks about the exercises:

- For your convenience functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a solution box has been added, but you may insert additional boxes. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`). But make sure to replace any part that says `YOUR CODE HERE` or `YOUR ANSWER HERE` and remove the `raise NotImplementedError()`.
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- For some exercises test cases have been provided in a separate cell in the form of `assert` statements. When run, a successful test will give no output, whereas a failed test will display an error message.
- Each sheet has 100 points worth of exercises. Note that only the grades of sheets number 2, 4, 6, 8 count towards the course examination. Submitting sheets 1, 3, 5, 7 & 9 is voluntary and their grades are just for feedback.

Please fill in your name here:

```
NAME = ""
NAMES_OF_COLLABORATORS = ""
```

Exercise sheet 7

Code from the lectures:

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def potential_v(x,lamb):
    '''Compute the potential function V(x).'''
    return lamb*(x*x-1)*(x*x-1)+x*x

def neighbor_sum(phi,s):
```

(continues on next page)

(continued from previous page)

```
    '''Compute the sum of the state phi on all 8 neighbors of the site s.'''
    w = len(phi)
    return (phi[(s[0]+1)%w,s[1],s[2],s[3]] + phi[(s[0]-1)%w,s[1],s[2],s[3]] +
            phi[s[0],(s[1]+1)%w,s[2],s[3]] + phi[s[0],(s[1]-1)%w,s[2],s[3]] +
            phi[s[0],s[1],(s[2]+1)%w,s[3]] + phi[s[0],s[1],(s[2]-1)%w,s[3]] +
            phi[s[0],s[1],s[2],(s[3]+1)%w] + phi[s[0],s[1],s[2],(s[3]-1)%w] )

def scalar_action_diff(phi,site,newphi,lamb,kappa):
    '''Compute the change in the action when phi[site] is changed to newphi.'''
    return (2 * kappa * neighbor_sum(phi,site) * (phi[site] - newphi) +
            potential_v(newphi,lamb) - potential_v(phi[site],lamb) )

def scalar_MH_step(phi,lamb,kappa,delta):
    '''Perform Metropolis-Hastings update on state phi with range delta.'''
    site = tuple(rng.integers(0,len(phi),4))
    newphi = phi[site] + rng.uniform(-delta,delta)
    deltaS = scalar_action_diff(phi,site,newphi,lamb,kappa)
    if deltaS < 0 or rng.uniform() < np.exp(-deltaS):
        phi[site] = newphi
        return True
    return False

def run_scalar_MH(phi,lamb,kappa,delta,n):
    '''Perform n Metropolis-Hastings updates on state phi and return number of␣
 ↪accepted transtions.'''
    total_accept = 0
    for _ in range(n):
        total_accept += scalar_MH_step(phi,lamb,kappa,delta)
    return total_accept

def batch_estimate(data,observable,k):
    '''Devide data into k batches and apply the function observable to each.
    Returns the mean and standard error.'''
    batches = np.reshape(data,(k,-1))
    values = np.apply_along_axis(observable, 1, batches)
    return np.mean(values), np.std(values)/np.sqrt(k-1)
```

7.1 Lattice scalar field & heatbath algorithm

(100 points)

Goal: Implement and test the heatbath algorithm for the lattice scalar field simulation.

As in the lecture we consider the scalar field $\varphi : \mathbb{Z}_w^4 \to \mathbb{R}$ on the w^4 lattice with periodic boundary conditions (here we use the notation $\mathbb{Z}_w = \{0, 1, \dots, w-1\}$ for the integers modulo w) and lattice action

$$S_L[\varphi] = \sum_{x\in\Lambda} \left[V(\varphi(x)) - 2\kappa \sum_{\hat{\mu}} \varphi(x)\varphi(x+\hat{\mu}) \right], \qquad V(\varphi) = \lambda(\varphi^2-1)^2 + \varphi^2,$$

where the second sum is over the 4 unit vectors $\hat{\mu} = (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)$.

In the lecture notes the Metropolis-Hastings algorithm for local updates to ϕ has been implemented to sample from the desired distribution $\pi(\varphi) = \frac{1}{Z} e^{-S_L[\varphi]}$. As in the case of the Ising model this local updating suffers from **critical slowing down** when close to the symmetry-breaking transition. So one should always investigate alternative Monte Carlo updates. One of these is the **heatbath algorithm** that we have already discussed in the setting of the harmonic oscillator. It entails selecting a uniform site $x_0 \in \mathbb{Z}_w^4$ and sampling a new random value Y for $\varphi(x_0)$ with density proportional to $f_Y(y) \propto \pi(\varphi(x)|_{\varphi(x_0)=y})$ on $\mathbb{R}$. To sample such a y we will make use of **acceptance-rejection sampling** (recall the lecture of week 3).

(a) Show with a calculation, to be included below in markdown and LaTeX, that for any choice of $c > 0$ we have

$$f_Y(y) \propto \exp\left(-c(y - s/c)^2 - \lambda(y^2 - v)^2\right),$$

where s and v are given by

$$s := \kappa \sum_{\hat{\mu}} (\varphi(x_0 + \hat{\mu}) + \varphi(x_0 - \hat{\mu})), \qquad v := 1 + \frac{c-1}{2\lambda}.$$

(15 pts)

YOUR ANSWER HERE

(b) Implement acceptance/rejection sampling of the random variable Y with pdf $f_Y(y) \propto \exp\left(-c(y - s/c)^2 - \lambda(y^2 - v)^2\right)$. You may treat c, s, λ as given parameters for this. Test your sampling by producing a histogram for $\lambda = 3$, $s = 0.3$ and three different values for c (e.g. 0.5, 1.0, 2.0). *Hint:* use proposal distribution $f(y) = \sqrt{\frac{c}{\pi}} \exp\left(-c(y - s/c)^2\right)$ and appropriate acceptance probability $A(y)$. **(25 pts)**

```
def sample_Y(s,lamb,c):
    '''Sample Y width density f_Y(y).'''
    # YOUR CODE HERE
    raise NotImplementedError()
```

```
from nose.tools import assert_almost_equal
assert_almost_equal(np.mean([sample_Y(0.8,1.5,2.0) for _ in range(800)]),0.71,
 ↪delta=0.06)
assert_almost_equal(np.mean([sample_Y(-0.7,0.5,0.6) for _ in range(800)]),-0.60,
 ↪delta=0.06)
```

```
# Plotting
# YOUR CODE HERE
raise NotImplementedError()
```

(c) We still have a free parameter $c > 0$ that we can choose as we see fit. We would like to choose c such that the acceptance probability $\mathbb{P}(\text{accept}) = \int_{-\infty}^{\infty} dy f(y)A(y)$ is maximized. Show by a calculation, to be included in markdown and LaTeX below, that c must then be a positive root of the polynomial

$$-c^3 + (1 - 2\lambda)c^2 + \lambda c + 2s^2\lambda = 0.$$

Hint: evaluate $\frac{d}{dc} f(y)A(y)$ and set it equal to 0. **(10 pts)**

YOUR ANSWER HERE

In fact, it can be easily shown that the polynomial above has only a single positive root when $\lambda > 0$, so this uniquely determines an optimal c. Computing it accurately can be costly, but it is possible to obtain a good approximation based on a Taylor series estimate, which yields:

```
def approx_optimal_c(s,lamb):
    '''Compute a value of c that is close to a positive root of the polynomial␣
 ↪above.'''
    u = np.sqrt(1+4*lamb*lamb)
    return ((3 + 3*u*(1-2*lamb)+4*lamb*(3*lamb-1)
             + np.sqrt(16*s*s*(1+3*u-2*lamb)*lamb +(1+u-
 ↪2*u*lamb+4*lamb*lamb)**2)) /
            (2+6*u-4*lamb))

def sample_Y_optimal(s,lamb):
    c = approx_optimal_c(s,lamb)
    return sample_Y(s,lamb,c)

approx_optimal_c(0.3,3)
```

(d) Implement the MCMC update using the heatbath with (approximately) optimal c. **(10 pts)**

```
def heatbath_update(phi,lamb,kappa):
    '''Perform a random heatbath update on the state phi (no return value␣
 ↪necessary).'''
    # YOUR CODE HERE
    raise NotImplementedError()

def run_scalar_heatbath(phi,lamb,kappa,n):
    '''Perform n heatbath updates on state phi.'''
    for _ in range(n):
        heatbath_update(phi,lamb,kappa)
```

(e) Gather 800 samples of $|m| = |w^{-4}\sum_{x\in\mathbb{Z}_w^4}\varphi(x)|$ for $w = 3$, $\lambda = 1.5$ and $\kappa = 0.08, 0.09, \ldots, 0.18$ for the heatbath algorithm as well as for the Metropolis-Hastings algorithm (width $\delta = 1.5$) from the lecture. Store your data in the HDF5-file `scalarfield.hdf5`. You may use 500 sweeps for equilibration 1 sweep in between measurements. **(15 pts)**

```
# Gathering and storing data
# YOUR CODE HERE
raise NotImplementedError()
```

(f) Read your data from `scalarfield.hdf5` and :

- plot your estimates for $\mathbb{E}[|m|]$ with error bars obtained from the heatbath and Metropolis algorithm for comparison;
- plot only the errors in your estimate so you can compare their magnitude for both algorithms;
- plot the estimated autocorrelation time in units of sweeps for both algorithms.

Where do you see an improvement? **(25 pts)**

```
# Reading data and plotting
# YOUR CODE HERE
raise NotImplementedError()
```

Chapter 8

Exercise sheet 8

Some general remarks about the exercises:

- For your convenience the functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a single solution box has been added, but you may insert additional boxes above or below using `Insert > Insert cell above / below`. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`).
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- This exercise sheet **will be graded**. The maximal number of points is **10** and a subdivision of these points for each exercise is indicated. Please submit your completed notebook before **Wednesday 17 Nov 10:30** by saving the notebook to your local computer via `File > Download as > Notebook (.ipynb)` and uploading it to the appropriate Brightspace Assignment.
- If you have submitted your solutions in time and your grade is below a 7.0, you have the option to resubmit your solutions within a week, so before Wednesday 24 Nov 10:30. Then they will be regraded, but with a cap of 7.0 on the grade.

8.1 Code from the lectures

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def fan_triangulation(n):
    '''Generates a fan-shaped triangulation of even size n.'''
    return np.array([[(i-3)%(3*n),i+5,i+4,(i+6)%(3*n),i+2,i+1] for i in range(0,
 ↪3*n,6)],dtype=np.int32).flatten()

def is_fpf_involution(adj):
    '''Test whether adj defines a fixed-point free involution.'''
    for x, a in enumerate(adj):
        if a < 0 or a >= len(adj) or x == a or adj[a] != x:
            return False
```

(continues on next page)

(continued from previous page)

```
    return True

from collections import deque

def triangle_neighbours(adj,i):
    '''Return the indices of the three neighboring triangles.'''
    return [j//3 for j in adj[3*i:3*i+3]]

def connected_components(adj):
    '''Calculate the number of connected components of the triangulation.'''
    n = len(adj)//3   # the number of triangles
    component = np.full(n,-1,dtype=np.int32)   # array storing the component␣
↪index of each triangle
    index = 0
    for i in range(n):
        if component[i] == -1:    # new component found, let us explore it
            component[i] = index
            queue = deque([i])   # use an exploration queue for breadth-first␣
↪search
            while queue:
                for nbr in triangle_neighbours(adj,queue.pop()):
                    if component[nbr] == -1:  # the neighboring triangle has not␣
↪been explored yet
                        component[nbr] = index
                        queue.appendleft(nbr)   # add it to the exploration queue
            index += 1
    return index

def next_around_triangle(i):
    '''Return the label of the side following side i in counter-clockwise␣
↪direction.'''
    return i - i%3 + (i+1)%3

def prev_around_triangle(i):
    '''Return the label of the side preceding side i in counter-clockwise␣
↪direction.'''
    return i - i%3 + (i-1)%3

def vertex_list(adj):
    '''
    Return the number of vertices and an array `vertex` of the same size as `adj`,
↪
    such that `vertex[i]` is the index of the vertex at the start (in ccw order)␣
↪of the side labeled `i`.
    '''
    vertex = np.full(len(adj),-1,dtype=np.int32)  # a side i that have not been␣
↪visited yet has vertex[i]==-1
    vert_index = 0  #
    for i in range(len(adj)):
        if vertex[i] == -1:
            side = i
            while vertex[side] == -1:  # find all sides that share the same vertex
                vertex[side] = vert_index
                side = next_around_triangle(adj[side])
            vert_index += 1
    return vert_index, vertex

def number_of_vertices(adj):
    '''Calculate the number of vertices in the triangulation.'''
    return vertex_list(adj)[0]
```

(continues on next page)

(continued from previous page)

```
def is_sphere_triangulation(adj):
    '''Test whether adj defines a triangulation of the 2-sphere.'''
    if not is_fpf_involution(adj) or connected_components(adj) != 1:
        return False
    num_vert = number_of_vertices(adj)
    num_face = len(adj)//3
    num_edge = len(adj)//2
    # verify Euler's formula for the sphere
    return num_vert - num_edge + num_face == 2

def flip_edge(adj,i):
    if adj[i] == next_around_triangle(i) or adj[i] == prev_around_triangle(i):
        # flipping an edge that is adjacent to the same triangle on both sides␣
 ↪makes no sense
        return False
    j = prev_around_triangle(i)
    k = adj[i]
    l = prev_around_triangle(k)
    n = adj[l]
    adj[i] = n  # it is important that we first update
    adj[n] = i  # these adjacencies, before determining m,
    m = adj[j]  # to treat the case j == n appropriately
    adj[k] = m
    adj[m] = k
    adj[j] = l
    adj[l] = j
    return True

def random_flip(adj):
    random_side = rng.integers(0,len(adj))
    return flip_edge(adj,random_side)

import networkx as nx
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection

def triangulation_edges(triangulation,vertex):
    '''Return a list of vertex-id pairs corresponding to the edges in the␣
 ↪triangulation.'''
    return [(vertex[i],vertex[j]) for i,j in enumerate(triangulation) if i < j]

def triangulation_triangles(triangulation,vertex):
    '''Return a list of vertex-id triples corresponding to the triangles in the␣
 ↪triangulation.'''
    return [vertex[i:i+3] for i in range(0,len(triangulation),3)]

def plot_triangulation_3d(adj):
    '''Display an attempt at embedding the triangulation in 3d.'''
    num_vert, vertex = vertex_list(adj)
    edges = triangulation_edges(adj,vertex)
    triangles = triangulation_triangles(adj,vertex)
    # use the networkX 3d graph layout algorithm to find positions for the␣
 ↪vertices
    pos = np.array(list(nx.spring_layout(nx.Graph(edges),dim=3).values()))
    fig = plt.figure()
    ax = Axes3D(fig, auto_add_to_figure=False)
    fig.add_axes(ax)
    tris = Poly3DCollection(pos[triangles])
    tris.set_edgecolor('k')
    ax.add_collection3d(tris)
    ax.set_xlim3d(np.amin(pos[:,0]),np.amax(pos[:,0]))
```

(continues on next page)

(continued from previous page)

```
    ax.set_ylim3d(np.amin(pos[:,1]),np.amax(pos[:,1]))
    ax.set_zlim3d(np.amin(pos[:,2]),np.amax(pos[:,2]))
    plt.show()

def vertex_neighbors_list(adj):
    '''Return a list `neighbors` such that `neighbors[v]` is a list of neighbors␣
 ↪of the vertex v.'''
    num_vertices, vertex = vertex_list(adj)
    neighbors = [[] for _ in range(num_vertices)]
    for i,j in enumerate(adj):
        neighbors[vertex[i]].append(vertex[j])
    return neighbors

def vertex_distance_profile(adj,max_distance=30):
    '''Return array `profile` of size `max_distance` such that `profile[r]` is␣
 ↪the number
    of vertices that have distance r to a randomly chosen initial vertex.'''
    profile = np.zeros((max_distance),dtype=np.int32)
    neighbors = vertex_neighbors_list(adj)
    num_vertices = len(neighbors)
    start = rng.integers(num_vertices) # random starting vertex
    distance = np.full(num_vertices,-1,dtype=np.int32)  # array tracking the␣
 ↪known distances (-1 is unknown)
    queue = deque([start])   # use an exploration queue for the breadth-first␣
 ↪search
    distance[start] = 0
    profile[0] = 1  # of course there is exactly 1 vertex at distance 0
    while queue:
        current = queue.pop()
        d = distance[current] + 1  # every unexplored neighbour will have this␣
 ↪distance
        if d >= max_distance:
            break
        for nbr in neighbors[current]:
            if distance[nbr] == -1:  # this neighboring vertex has not been␣
 ↪explored yet
                distance[nbr] = d
                profile[d] += 1
                queue.appendleft(nbr)   # add it to the exploration queue
    return profile

def perform_sweeps(adj,t):
    '''Perform t sweeps of flip moves, where 1 sweep is N moves.'''
    for _ in range(len(adj)*t//3):
        random_flip(adj)

def batch_estimate(data,observable,num_batches):
    batch_size = len(data)//num_batches
    values = [observable(data[i*batch_size:(i+1)*batch_size]) for i in range(num_
 ↪batches)]
    return np.mean(values), np.std(values)/np.sqrt(num_batches-1)
```

8.2 8.1 Estimating Hausdorff dimensions in various 2D quantum gravity models (10 Points)

In the lecture we considered the model of two-dimensional Dynamical Triangulations of the 2-sphere. The corresponding partition function is

$$Z^U_{S^2,N} = \sum_T 1, \tag{8.1}$$

where the sum is over all triangulations of size N with the topology of S^2, each of which is represented as an adjacency list $\mathrm{adj} : \{0, \dots, 3N-1\} \to \{0, \dots, 3N-1\}$. To emphasize that we are dealing with the **uniform** probability distribution on such triangulations, we have added the label ^{U}. It is a lattice model of two-dimensional Euclidean quantum gravity with no coupled matter.

One can also consider two-dimensional quantum gravity coupled to matter fields (e.g. a scalar field) supported on the geometry. Formally the corresponding path integral in the continuum reads

$$Z = \int [\mathcal{D}g_{ab}] \int [\mathcal{D}\phi] e^{-\frac{1}{\hbar}(S_E[g_{ab}]+S_m[\phi,g_{ab}])} = \int [\mathcal{D}g_{ab}] e^{-\frac{1}{\hbar}S_E[g_{ab}]} Z^*_m[g_{ab}],$$

where $S_m[\phi, g_{ab}]$ and $Z_m[g_{ab}]$ are the matter action and path integral of the field ϕ on the geometry described by g_{ab}. The natural analogue in Dynamical Triangulations is

$$Z^*_{S^2,N} = \sum_T Z^*_m[T],$$

where the sum is over the same triangulations as in (8.1) but now the summand $Z^*_m[T]$ is the lattice partition function of a matter system supported on the triangulation T, which generically depends in a non-trivial way on T. For instance, the matter system could be an Ising model in which the spin are supported on the triangles of T and $Z^{\text{Ising}}_m[T]$ would be the corresponding Ising partition function. In other words, when Dynamical Triangulations are coupled to matter the uniform distribution $\pi^U(T) = 1/Z^U_{S^2,N}$ is changed into a non-uniform distribution $\pi^*(T) = Z^*_m[T]/Z^*_{S^2,N}$. This can have significant effect on the critical exponents of the random triangulation as $N \to \infty$, like the Hausdorff dimension.

The goal of this exercise is to estimate the **Hausdorff dimension** of random triangulations in four different models and to conclude based on this that they belong to four different universality classes (i.e. that if they possess well-defined continuum limits that they are described by four different EQFTs):

- $Z^U_{S^2,N}$: the standard Dynamical Triangulations with **U**niform distribution (U)
- $Z^W_{S^2,N}$: triangulations coupled to a matter system called a Schnyder **W**ood (W)
- $Z^S_{S^2,N}$: triangulations coupled to a matter system called a **S**panning tree (S)
- $Z^B_{S^2,N}$: triangulations coupled to a matter system called a **B**ipolar orientation (B)

What these matter systems precisely represent will not be important. We have provided for you a **black box generator** that samples from the corresponding four distributions $\pi^U(T)$, $\pi^W(T)$, $\pi^S(T)$, $\pi^B(T)$. It does so in an efficient manner (linear time in N) using direct Monte Carlo sampling algorithms and therefore returns independent samples with exactly the right distribution (within numerical precision).

8.2.1 Using the black box generator

The black box generator is provided by the executable program `generator` in the folder `triangulationcode`. It can be called directly from this notebook with the following function, that takes the desired size N and model (U, W, S, B) and returns a single random triangulation in the usual form of an adjacency list.

```
def generate_random_triangulation(n,model):
    '''
    Returns a random triangulation generated by the program `generator` in the␣
↪form
```

(continues on next page)

(continued from previous page)

```
    of an array of length 3n storing the adjacency information of the triangle␣
↪sides.
    Parameters:
      n - number of triangles in the triangulation, must be positive and even
      model - a one-letter string specifying the model from which the␣
↪triangulation is sampled:
          'U': Uniform triangulations
          'W': Schnyder-Wood-decorated triangulations
          'S': Spanning-tree decorated triangulations
          'B': Bipolar-oriented triangulations
    '''
    output = !./triangulationcode/generator -s{n} -t{model}
    return np.array([int(num) for num in output],dtype=np.int32)

adj = generate_random_triangulation(10,'B')
print(adj)
print("Is this a sphere triangulations? {}".format(is_sphere_triangulation(adj)))
```

```
True
```

Error? If your get an error or the result is not `True` (and you did not forget to run the lecture code above first), then most likely the program `generator` is not correctly compiled on your system. The easy solution is to switch to JupyterHub where the program was tested to work. Otherwise you need to compile the c++ code yourself as follows. Assuming you have the GNU c++ compiler installed you can easily do this as follows: visit the directory `monte-carlo-techniques-2021/exercises/triangulationcode/` in the terminal and run `g++ generate.cpp -std=c++11 -O3 -o generator`. It should work similarly with other c++ compilers.

Recall that the **distance profile** $\rho_T(r)$ of a triangulation is defined as

$$\rho_T(r) = \frac{1}{V}\sum_{x=0}^{V-1}\sum_{y=0}^{V-1}\mathbf{1}_{\{d_T(x,y)=r\}},$$

where $V=(N+4)/2$ is the number of vertices and $d_T(x,y)$ is the graph distance between the vertices with label x and y.

(a) Let T be a random triangulation of size N and X, Y two independent numbers chosen uniformly from $0,\dots,V-1$, corresponding to two random vertices in T. Explain that $\frac{1}{V}\mathbb{E}[\rho_T(r)] = \mathbb{P}(d_T(X,Y)=r)$ and that the expected distance between X and Y is related to the distance profile via

$$\mathbb{E}[d_T(X,Y)] = \frac{1}{V}\sum_{r=0}^{\infty} r\,\mathbb{E}[\rho_T(r)]. \tag{8.2}$$

(b) We will work under the assumption that $\mathbb{E}[\rho_T(r)] \approx V^{1-1/d_H} f(rV^{-1/d_H})$ for a positive real number d_H called the **Hausdorff dimension** and a continuous function f that are both independent of N but do depend on the model. Show that

$$\mathbb{E}[d_T(X,Y)] \approx c\,V^{1/d_H}, \qquad c=\int_0^\infty \mathrm{d}x\, x\, f(x). \tag{8.3}$$

Hint: Approximate the summation by an integral.

(c) For each of the four models estimate $\mathbb{E}[d_T(X,Y)]$ for $N=2^7,2^8,\dots,2^{11}$ using (8.2) and based on 100 samples each. Do not forget to estimate errors! Make an estimate of d_H for each of the models by fitting c and d_H in the ansatz (8.3). Plot the data together with the fit in a log-log plot. (Do not worry if d_H for model U comes out significantly below 4.)

(d) Perform the following exercise for each of the four models. Estimate the full mean distance profile $\mathbb{E}[\rho_t(r)]$ for $N=2^7,2^8,\dots,2^{11}$ based on 100 samples each. Show the data for all sizes in a single figure by plotting

$$V^{1/d_H}\,\mathbb{E}[\frac{1}{V}\rho_T(r)] \quad \text{as function of } x = r/V^{1/d_H},$$

where for d_H you take the estimate obtained in the previous exercise. Verify that the curves **collapse** reasonably well (as demonstrated in the lecture for model U and $d_H = 4$).

(e) Bonus exercise: Make more robust estimates of d_H by optimizing the quality of the collapse.

Chapter 9

Exercise sheet 9

Some general remarks about the exercises:

- For your convenience the functions from the lecture are included below. Feel free to reuse them without copying to the exercise solution box.
- For each part of the exercise a single solution box has been added, but you may insert additional boxes above or below using `Insert > Insert cell above / below`. Do not hesitate to add Markdown boxes for textual or LaTeX answers (via `Cell > Cell Type > Markdown`).
- Please make your code readable by humans (and not just by the Python interpreter): choose informative function and variable names and use consistent formatting. Feel free to check the PEP 8 Style Guide for Python for the widely adopted coding conventions or this guide for explanation.
- Make sure that the full notebook runs without errors before submitting your work. This you can do by selecting `Kernel > Restart & Run All` in the jupyter menu.
- This exercise sheet will **not** be graded, but if you may still submit it via Brightspace to receive feedback from the teaching assistant. Do this before **Wednesday 24 November 10:30** by saving the notebook to your local computer via `File > Donwload as > Notebook (.ipynb)` and uploading to the appropriate Brightspace Assignment.

9.1 Code from the lectures

```
import numpy as np
import matplotlib.pylab as plt
import os
%matplotlib inline

def read_all_json(directory,startswith):
    '''Read all json-files in `directory` that start with `startswith` and return␣
↪the data as a list.'''
    data = []
    for filename in sorted(os.listdir(directory)):
        if filename.startswith(startswith) and filename.endswith(".json"):
            with open(os.path.join(directory,filename)) as file:
                data.append(json.load(file))
    return data

def batch_estimate(data,observable,num_batches):
    '''Determine estimate of observable on the data and its error using batching.'
↪''
```

(continues on next page)

(continued from previous page)

```
    batch_size = len(data)//num_batches
    values = [observable(data[i*batch_size:(i+1)*batch_size]) for i in range(num_
↪batches)]
    return np.mean(values), np.std(values)/np.sqrt(num_batches-1)
```

9.2 9.1 Running code on the compute cluster: lattice scalar field

Goal: Learn how to scale up simulations by transitioning from running them in the notebook to stand-alone scripts on the compute cluster.

In lecture 7 we implemented the Metropolis-Hastings simulation of a lattice scalar field using the following code:

```
import numpy as np
rng = np.random.default_rng()
import matplotlib.pylab as plt
%matplotlib inline

def potential_v(x,lamb):
    '''Compute the potential function V(x).'''
    return lamb*(x*x-1)*(x*x-1)+x*x

def neighbor_sum(phi,s):
    '''Compute the sum of the state phi on all 8 neighbors of the site s.'''
    w = len(phi)
    return (phi[(s[0]+1)%w,s[1],s[2],s[3]] + phi[(s[0]-1)%w,s[1],s[2],s[3]] +
            phi[s[1],(s[1]+1)%w,s[2],s[3]] + phi[s[1],(s[1]-1)%w,s[2],s[3]] +
            phi[s[0],s[1],(s[2]+1)%w,s[3]] + phi[s[0],s[1],(s[2]-1)%w,s[3]] +
            phi[s[0],s[1],s[2],(s[3]+1)%w] + phi[s[0],s[1],s[2],(s[3]-1)%w] )

def scalar_action_diff(phi,site,newphi,lamb,kappa):
    '''Compute the change in the action when phi[site] is changed to newphi.'''
    return (2 * kappa * neighbor_sum(phi,site) * (phi[site] - newphi) +
            potential_v(newphi,lamb) - potential_v(phi[site],lamb) )

def scalar_MH_step(phi,lamb,kappa,delta):
    '''Perform Metropolis-Hastings update on state phi with range delta.'''
    site = tuple(rng.integers(0,len(phi),4))
    newphi = phi[site] + rng.uniform(-delta,delta)
    deltaS = scalar_action_diff(phi,site,newphi,lamb,kappa)
    if deltaS < 0 or rng.uniform() < np.exp(-deltaS):
        phi[site] = newphi
        return True
    return False

def run_scalar_MH(phi,lamb,kappa,delta,n):
    '''Perform n Metropolis-Hastings updates on state phi and return number of␣
↪accepted transtions.'''
    total_accept = 0
    for _ in range(n):
        total_accept += scalar_MH_step(phi,lamb,kappa,delta)
    return total_accept

lamb = 1.5
kappas = np.linspace(0.08,0.18,11)
width = 3
num_sites = width**4
delta = 1.5  # chosen to have ~ 50% acceptance
equil_sweeps = 800
```

(continues on next page)

(continued from previous page)

```
measure_sweeps = 2
measurements = 1000

mean_magn = []
for kappa in kappas:
    phi_state = np.zeros((width,width,width,width))
    run_scalar_MH(phi_state,lamb,kappa,delta,equil_sweeps * num_sites)
    magnetizations = np.empty(measurements)
    for i in range(measurements):
        run_scalar_MH(phi_state,lamb,kappa,delta,measure_sweeps * num_sites)
        magnetizations[i] = np.mean(phi_state)
    mean, err = batch_estimate(np.abs(magnetizations),lambda x:np.mean(x),10)
    mean_magn.append([mean,err])

plt.errorbar(kappas,[m[0] for m in mean_magn],yerr=[m[1] for m in mean_magn],fmt=
 ↪'-o')
plt.xlabel(r"$\kappa$")
plt.ylabel(r"$|m|$")
plt.title(r"Absolute field average on $3^4$ lattice, $\lambda = 1.5$")
plt.show()
```

The goal will be to reproduce and extend its output (below).

(a) Turn the simulation into a standalone script `latticescalar.py` (similar to `ising.py`) that takes the relevant parameters e.g.

```
$ python3 latticescalar.py -l 1.5 -k 0.12 -w 3 -n 1000
```

for $\lambda = 1.5$, $\kappa = 0.12$, $w = 3$, and 1000 measurements, together with optional parameters δ and numbers of sweeps, as command line arguments and stores the relevant simulation outcomes to a json-file.

(b) Write a bash script that submits jobs to the cluster for $w = 3$ and 2000 measurements and $\lambda = 1.0, 1.5, 2.0$ and $\kappa = 0.08, 0.09, ..., 0.18$ (so 33 simulations in total). Submit the job to the `hefstud` slurm partition (do not run all 33 in parallel).

(c) Load the stored data into this notebook and reproduce the plot above (with $\lambda = 1$ and $\lambda = 2$ added).

Part III

Appendix

Chapter 1

Bibliography

[ADC21] Michael Aizenman and Hugo Duminil-Copin. Marginal triviality of the scaling limits of critical 4D Ising and $\phi _4^4$ models. *Ann. of Math. (2)*, 194(1):163–235, 2021. URL: https://doi.org/10.4007/annals.2021.194.1.3, doi:10.4007/annals.2021.194.1.3.

[AmbjornDJ97] Jan Ambjørn, Bergfinnur Durhuus, and Thordur Jonsson. *Quantum Geometry: A Statistical Field Theory Approach*. Cambridge University Press, 1997. URL: https://www.cambridge.org/core/books/quantum-geometry/943681B03BE0B8D34BC666C22EA18713, doi:10.1017/CBO9780511524417.

[AGJL12] J. Ambjørn, A. Görlich, J. Jurkiewicz, and R. Loll. Nonperturbative quantum gravity. *Physics Reports*, 519(4):127–210, 2012. Nonperturbative Quantum Gravity. URL: https://www.sciencedirect.com/science/article/pii/S0370157312001482, doi:https://doi.org/10.1016/j.physrep.2012.03.007.

[Cre85] Michael Creutz. *Quarks, gluons and lattices*. Volume 8. Cambridge University Press, 1985.

[Jos20] Anosh Joseph. *Markov Chain Monte Carlo Methods in Quantum Field Theories: A Modern Primer*. Springer Briefs in Physics, 2020. arXiv:1912.10997v3, doi:10.1007/978-3-030-46044-0.

[Kra06] Werner Krauth. *Statistical Mechanics*. Oxford University Press, USA, 2006. ISBN 9780198515364. URL: https://global.oup.com/ukhe/product/statistical-mechanics-algorithms-and-computations-9780198515364?cc=nl&lang=en&.

[Lui06] E. Luijten. *Introduction to Cluster Monte Carlo Algorithms*, pages 13–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. URL: https://doi.org/10.1007/3-540-35273-2_1, doi:10.1007/3-540-35273-2_1.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953. doi:10.1063/1.1699114.

[MM94] Istvan Montvay and Gernot Münster. *Quantum Fields on a Lattice*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, 1994. doi:10.1017/CBO9780511470783.

[Mor07] Colin Morningstar. The monte carlo method in quantum field theory. *arXiv preprint hep-lat/0702020*, 2007. URL: https://arxiv.org/abs/hep-lat/0702020.

[NB99] M.E.J. Newman and G.T. Barkema. *Monte Carlo Methods in Statistical Physics*. Clarendon Press, 1999. ISBN 9780191589867. URL: https://books.google.nl/books?id=HgBREAAAQBAJ.

[Nor97] J. R. Norris. *Markov Chains*. Cambridge University Press, feb 1997. URL: https://www.cambridge.org/core/books/markov-chains/A3F966B10633A32C8F06F37158031739, doi:10.1017/cbo9780511810633.

[Owe13] Art B. Owen. *Monte Carlo theory, methods and examples*. online, 2013. URL: https://artowen.su.domains/mc/.

[Rot12] Heinz J Rothe. *Lattice gauge theories: an introduction*. World Scientific Publishing Company, 2012.

[Rum] Kari Rummukainen. Monte carlo simulation methods: lecture notes. URL: https://www.mv.helsinki.fi/home/rummukai/lectures/montecarlo_oulu/.

[Smi02] Jan Smit. *Introduction to Quantum Fields on a Lattice*. Cambridge Lecture Notes in Physics. Cambridge University Press, 2002. doi:10.1017/CBO9780511583971.

[Ula91] Stanislaw M. Ulam. *Adventures of a mathematician*. Scribners, 1991.