Simulation and Optimization: A Model-Driven Approach

Rubén Ruiz-Torrubiano

Feb 4, 2026

Table of contents

Welcome

This is the website for the **Simulation and Optimization** book that will teach you the basics of simulation approaches and optimization techniques in the context of modern AI systems. The source code of the book and the examples are provided as open-source and free to use Creative Commons Attribution-NonCommercial-NoDerivs 4.0 license.

Preface

This book was created as companion material for a semester graduate course on simulation and optimization. It is the author's opinion that in an age of rapid advances in the field of artificial intelligence, it is of utmost importance to focus not only on machine learning, but to study in detail the techniques that make current advances in AI possible. From those, the areas of simulation and optimization have the highest potential to reveal how intrincate current AI is intertwined with other areas of mathematics, statistics and computer science.

Simulation techniques are widely used in many scientific disciplines, ranging from climate models, epidemiology, and engineering to finance and logistics. These methods allow researchers and practitioners to analyze complex systems, evaluate scenarios, and make informed decisions when analytical solutions are infeasible or unavailable. Throughout this book, we will explore foundational concepts and practical approaches to simulation and optimization, providing both theoretical background and hands-on examples. In the context of AI, simulation approaches can be used to produce synthetic data for training in situations where these data are scarce, expensive, or simply impossible to collect. Another uses of simulation approaches include stress-testing algorithms, validating models under various hypothetical scenarios, and supporting decision-making in uncertain environments. By leveraging simulation, practitioners can gain insights into system behavior, identify potential risks, and optimize performance before deploying solutions in real-world settings.

Optimization approaches lie at the core of how machine learning is used in modern AI systems. Foundational algorithms like stochastic gradient descent make it possible to find optimal parameters for machine learning models using training datasets composed of millions of data points. Additionaly, optimization algorithms are used for hyperparameter tuning and can be found at the heart of classical approaches like support vector machines and logistic regression. In this context, both classical and metaheuristic approaches play a pivotal role in finding optimal or near-optimal solutions which are used in the broader context of specific applications in practice.

Throughout the book, we will assume that the reader has familiarity with linear algebra and calculus and possesses a good command of statistics and the basics of machine learning. Additionally, good background knowledge of the Python programming language is adviced for the practical part of this book.

1 Introduction

Simulation and optimization approaches are present in our everyday lives, albeit most of the time operating in a background plane. For example, when navigating with a GPS, the system simulates different routes and optimizes for the shortest or fastest path. Similarly, supply chains use optimization algorithms to minimize costs and maximize efficiency, while simulations help predict demand and manage inventory. These techniques are fundamental tools in decision-making processes across various industries, from transportation and logistics to finance and healthcare.

But what do simulation and optimization approaches have in common, apart from being complementary tools? The answer lies in the concept of a *model*. In the context of machine learning, we normally refer to a model as a mathematical or computational representation that captures the relationships between input data and output predictions. In simulation and optimization, a model similarly serves as an abstraction of a real-world system or process, allowing us to analyze, predict, and improve its behavior through experimentation and algorithmic techniques.

In the following, we will delve deeper into the concept of a model and how models are used in simulation and optimization contexts using some practical examples.

1.1 Example 1: Simulating supermarket dynamics

Imagine you are in your favourite grocery store waiting at the checkout queue. For simplicity, let's assume there is only one open counter. When you arrive at the queue, there might be other customers already waiting, while the first customer at the queue is currently being served. Shortly after you, a new customer arrives, taking the next free spot right behind you. And then another customer arrives, and another one, and another one…

Let's try to break down how this system behaves and what are the most important interactions between the parts of the system. In general, we will distinguish between *components*, *states*, *events*, *inputs* and *metrics*.

- **Components:** These are the entities that interact with each other. In our example, we have customers, cashiers and the queue itself.

- **States:** The configurations of the system that represent valid combinations of specific properties of the components at a given moment of time. For instance, at each time the queue has a specific length: zero if it's empty, one customer, two customers, etc. Additionally, the cashier can be busy or idle. We can also count the number of customers currently present in the supermarket which have not yet arrive at the checkout queue.
- **Events:** The interactions themselves, like a new customer arriving at the queue, checkout start or checkout completion.
- **Inputs:** Whatever information is fed into the system, e.g. arrival times, service times, etc. These inputs can contain statistical assumptions, like the distribution of arrival times.
- **Metrics:** How we evaluate the system as a whole in a given time step. For instance, what is the average waiting time? How much time are the cashiers busy? How is the queue length distributed?

The system could be represented by the following Python code as a minimal variant.

```
import heapq, random

# event = (time, type, customer_id)
event_list = []
heapq.heappush(event_list, (first_arrival_time, 'arrival', 1))

while event_list and time < sim_end:
    time, ev_type, cid = heapq.heappop(event_list)
    if ev_type == 'arrival':
        if any_cashier_free():
            start_checkout(cid, time)
            heapq.heappush(event_list, (time + service_time(cid),
                'departure', cid))
        else:
            enqueue(cid, time)
        heapq.heappush(event_list, (time + next_interarrival(),
            'arrival', next_id()))
    elif ev_type == 'departure':
        finish_service(cid, time)
        if queue_not_empty():
            next_cid = dequeue()
            start_service(next_cid, time)
            heapq.heappush(event_list, (time + service_time(next_cid),
                'departure', next_cid))
```

This code assumes that customers arrive at regular subsequent intervals after each arrival event. The parameter `sim_end` defines how long (how many steps) we want to simulate in this case. The function `service_time` returns the time the cashier needs for checking out customer `cid`.

The next customer will arrive after a time given by the function `next_interarrival`, which can implement different stochastic behaviours.

We can represent this system graphically as shown in the following illustration:

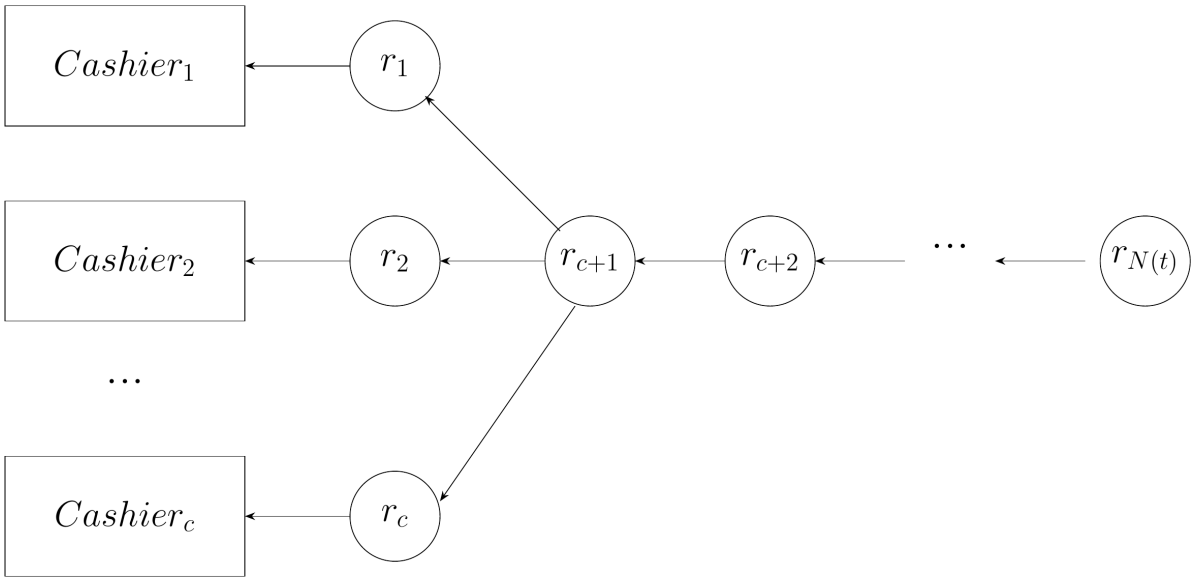


Figure 1.1: An illustration of the supermarket dynamics.

In this figure, customers are denoted by r_i, the amount of cashiers is c and the total number of customers in the supermarket at time t is denoted by $N(t)$.

Now let's try to refine the dynamics of this system. We will now write some equations to describe the system' dynamics according to the **infinite waiting room** $M/M/c$ model. Let's make the following assumptions:

- Arrivals follow a Poisson distribution with mean λ (arrivals per second), which for this case will be assumed to be stationary.
- The service times are assumed to be exponentially distributed with mean $1/\mu$.

What would be now the *traffic intensity per cashier*? That is, what is the mean customer flow that each cashier experiences from their own point of view? Let's call this number ρ and calculate it as follows:

$$\rho = \frac{\lambda}{c\mu} \tag{1.1}$$

In words, if customers arrive at a rate of $\lambda = 10$ customers/s and each cashier serves 2 customers/s (yes, it's a fast supermarket). With 5 cashiers, that means that $\rho = 10/5 \times 2 = 1$. This means that each cashier has quite a lot to do right now.

We are now interested in the probabilities of the states in this systems. In this case, we define a system by the number of customers currently present in the supermarket. So we can have $N = 1$ if there is currently 1 customer present, or any other number of customers (we assume the supermarket is so large, we can accomodate any number of them). Let's denote these probabilities by $p_n = \Pr\{N = n\}$. We have:

$$p_n = \lim_{t\to\infty} \int_0^t \mathbb{1}_{\{N(s)=n\}} ds \tag{1.2}$$

Intuitively, p_n represents the fraction of time where the supermarket has exactly n customers. As mentioned earlier, we will assume that arrivals do not depend of the current state n, so we write $\lambda_n = \lambda$ for all $n \geq 0$. However, note that the completion rates μ_n do *indeed* depend of the current state. To see this, imagine that there is only one customer in the supermarket ($n = 1$). The completion rate is then $\mu_1 = \mu$ since the only one cashier is needed to perform checkout. However, if there are $n = 2$ customers in the supermarket, two cashiers can serve those two customers in parallel, increasing the completion rate to $\mu_2 = 2\mu$. The same reasoning applies until $n = c$, the total number of cashiers. In this case, $\mu_c = c\mu$ and the next customer will have to wait in the queue. So we have:

$$\begin{aligned} \lambda_n &= \lambda \text{ for all } n \geq 0 \\ \mu_n &= \min(n, c)\mu \end{aligned} \tag{1.3}$$

Now we are going to state our **main modeling assumption**. Consider how we transition *between states*. Specifically, we transition from state n to state $n + 1$ when a new customer enters the supermarket, and there were already n customers in it. Similarly, we transition from state n to state $n - 1$ when a customer leaves the supermarket (in this case, all customers are served by the cashiers, so there is no way you leave the supermarket without paying first). Remember that the rate of customers arriving at the supermarket is always λ, and the rate of customers being served (i.e. leaving) when at state n is μ_n. In general, for each state we can define an *incoming* and an *outgoing flow*. This quantifies the transitions in resp. out of a given state.

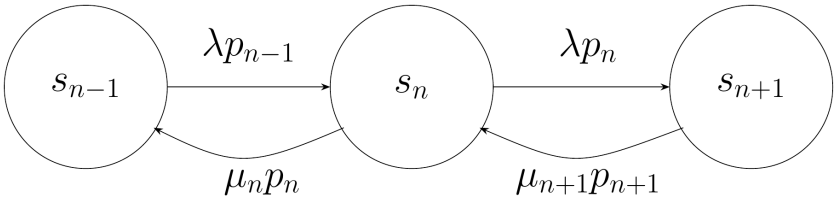


Figure 1.2: Transition dynamics between neighboring states

As can be seen in the previous figure, transitions flow away from state n in two ways: first, to state $n-1$ when a customer is served with a rate $\mu_n p_n$ and to state $n+1$ when a new customer

arrives with a rate λp_n. Similarly, one can transition from the other states to state n either by having a customer served in state $n+1$ with a rate $\mu_{n+1}p_{n+1}$ or when being in state $n-1$ and a new customer arrives with a rate λp_{n-1}. Our modeling assumption now is that, for each state n, the flow outwards balances out with the flow inwards (global balance):

$$\lambda p_{n-1} + \mu_{n+1}p_{n+1} = \lambda p_n + \mu_n p_n, \text{ for } n \geq 1 \tag{1.4}$$

We can also write that, in the long term, the rate of transitions from n to $n+1$ equals the transitions from $n+1$ to n, which results in the more simple form (local balance)

$$\lambda p_n = \mu_{n+1}p_{n+1}, \text{ for } n \geq 1 \tag{1.5}$$

This form follows from the global balance condition when only neighboring states are connected. Now, using this short form, we can provide closed-form expressions for different probabilities, using the following recursion (which directly follows from the above)

$$p_{n+1} = \frac{\lambda}{\mu_{n+1}} p_n \tag{1.6}$$

For instance, one can calculate that the probability of a customer having to wait (because all cashiers are busy at the moment) is

$$P_W = p_0 \frac{(\lambda/\mu)^c}{c!} \frac{1}{1-\rho} \tag{1.7}$$

This is also called the *Erlang-C* probability and we will delve deeper into the details in the coming chapters.

We can now write computer code that performs a step-by-step simulation of the system (i.e. in discrete time steps). This is specially useful if we are in a situation where there is no closed-form analytical solution, or the analytical solution is too complex to calculate. For instance, we can run the above simulator for a large number of steps (say $T = 10^6$) and then calculate specific metrics like:

- Mean queue length.
- Mean waiting time.
- Mean time in the system.
- Total fraction of time that the cashiers were busy.
- Overall utilization.

We will see examples of such simulators in the first part of the book.

1.2 Example 2: The traveling salesman

Let's not turn our attention to the other type of problems which are central to this book: *optimization problems*. Imagine you are a sales representative for a vaccum cleaner manufacturer. Your task is to visit potential customers in cities across your area and, at the end of the day, return to where you started your journey. As an environmental conscious employer and in order to save transport costs, your company introduces the restriction that each customer in the route has to be visited *exactly once*. So you need to think carefully before getting into your car and starting your route.

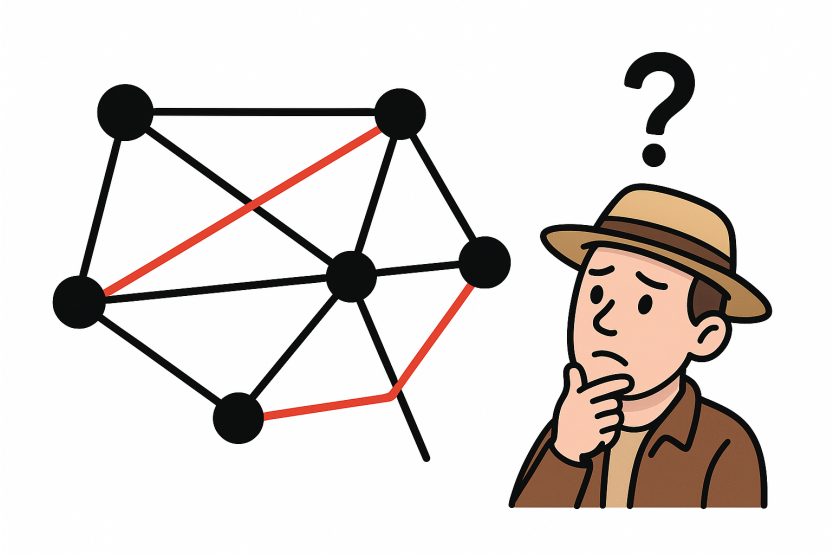


Figure 1.3: The traveling salesman has to find a good and practical solution

Now, the traveling salesman needs to consider two things:

- A method for *constructing valid* tours that start and end in the same location.
- A method for *evaluating* those tours so that we can quantitatively decide if a tour is better than another one.

Let's address each of these considerations in detail. Assume that $N = \{1, 2, \dots, n\}$ is our set of possible locations. Let's define the following variables:

$$x_{ij} = \begin{cases} 1 & \text{if the tour goes directly to location } i \text{ to location } j \\ 0 & \text{otherwise} \end{cases} \tag{1.8}$$

where $i, j \in N$. We call x_{ij} our *decision variables*. So if the traveling salesman specifies the value of each x_{ij}, we have a candidate route to consider. However, not every assignment of

the x_{ij} variables to $\{0, 1\}$ will make sense for the traveling salesman. For instance, imagine that we have in one assignment both $x_{23} = 1$ and $x_{43} = 1$. That would mean that location 3 is visited twice, once from location 2 and another time from location 4. That violates the requirement that each location is visited exactly once.

To model this situation, we need to introduce *constraints*. In optimization problems, constraints take usually the form of equalities or inequalities as functions of the decision variables. In our case, the requirement that each location is visited only once can be expressed by the following (linear) equalities:

$$\begin{aligned} \sum_{j=1}^{n} x_{ij} &= 1 \text{ for all } i \in N \\ \sum_{i=1}^{n} x_{ij} &= 1 \text{ for all } j \in N \end{aligned} \tag{1.9}$$

The first equality means that, fixed a location i, the sum of all *outgoing* edges is exactly one. Conversely, the second states that for a fixed location j, the sum of all *incoming* edges is also exactly one. This ensures that each location is visited exactly once.

We need another technical condition to guarantee that the tour is a single one and not composed of multiple sub-tours. There should be no subset $S \subset N$ such that is self contained, the number of visited cities equals exactly its size. Mathematically:

$$\sum_{i \in N} \sum_{j \in N} x_{ij} \leq |S| - 1 \text{ for all subsets } S \subset N \tag{1.10}$$

To sum up, we now have modeled how valid tours should look like. If we find a tour $T = \{x_{ij}\}$ that satisfies the constraints outlined before, we can be sure it is a valid tour.

But surely there are some tours that are better than others? This is where the second issue becomes important: we need an evaluation method to distinguish between good and bad solutions. In the optimization literature, we normally talk about **objective functions**. In our case, the traveling salesman would like the total distance to be *minimized*, meaning the sum of all distances between locations of the tour. Assume that $c_{ij} > 0$ is the distance between location i and j (for consistency assume $c_{ii} = 0$). Now we want to minimize the total distance traveled. For this we write:

$$\min \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij} \tag{1.11}$$

That is, if the traveling salesman visits location j from i, then $x_{ij} = 1$ and this activates the travel cost c_{ij} in the sum. Otherwise, $x_{ij} = 0$ and the cost does not count to the total sum, since that path is not traversed in the tour. Putting it all together, we have:

$$
\begin{aligned}
T^* = \text{argmin} & \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij} \\
\text{s.t.} & \sum_{j=1}^{n} x_{ij} = 1 \text{ for all } i \in N \\
& \sum_{i=1}^{n} x_{ij} = 1 \text{ for all } j \in N \\
& \sum_{i \in N} \sum_{j \in N} x_{ij} \leq |S| - 1 \text{ for all subsets } S \subset N \\
& x_{ij} \in \{0, 1\}
\end{aligned}
\tag{1.12}
$$

We call this set of expressions our *optimization model*. This will be the mathematical underpinning for all the methods and algorithms that we will use to find a solution to this problem. In the first line, we state our goal: to obtain a tour T^* that is optimal in the sense of minimizing the total cost (the expression argmin means "the argument that minimizes", so find the x_{ij} that minimize the total cost function). The subsequent lines state the *constraints* that we listed before. In the last line we specify the *domain* of the decision variables, i.e. what are the possible values these variables can take.

We will see that, depending on the form of the optimization model we will be able to choose from a toolbox of algorithms capable to solve the problem at hand, either exactly (*exact methods*) or approximately (*heuristic methods*).

1.3 Structure of the book

In this first chapter, we have introduced the concept of a *model* and have applied it successfully to a simulation and an optimization problem. The rest of the book is structured in two parts: Part I will be dedicated to simulation approaches, including the $M/M/c$ model we have seen in this chapter in Chapter 2. Monte Carlo methods are the main topic of Chapter 3. After that, Chapter 4 focuses on the handling of discrete events, while **?@sec-agent-based** concludes with considerations about agent-based modeling and simulation.

Part II is dedicated to optimization problems. In Chapter 5 we introduce the mathematical basics of optimization. Chapter 6 is dedicated to exact optimization methods like the simplex method for linear programming. Approximate methods for complex optimization problems like metaheuristics and evolutionary algorithms are presented in Chapter 7. Finally, we review the importance of optimization methods for machine learning in Chapter 8.

1.4 Exercises

1. Prove that in the supermarket example the local balance condition follows from the global balance condition (Hint: use induction).
2. What happens to the optimization model in presented in Equations 1.12 if we remove Equation 1.10? Find an example of a tour that is valid according to the model but invalid for the traveling salesman.

Part I

PART I: SIMULATION

2 Simulation basics

2.1 What is Simulation?

In science and engineering, it is of paramount importance to develop reliable quantitative models that capture the essential behavior of real systems. Simulation provides a controlled, repeatable, and cost-effective way to

- predict system behavior under varied conditions,
- explore "what-if" scenarios and design alternatives,
- quantify uncertainty and sensitivity to inputs,
- validate hypotheses when experiments are impractical or expensive,
- and support optimization and decision making.

A simulation study typically involves the following steps:

1. Construct a mathematical or computational model.
2. Specify inputs and assumptions.
3. Run experiments (often many replications with different parameters).
4. Analyze outputs and comparing them with data or theoretical expectations.

Proper validation and uncertainty quantification are critical to ensure that simulation results are trustworthy and useful for engineering practice.

Simulation can be defined as the methods and procedures to define models of a system of interest and execute it to get raw data (Osais 2017). In normal simulation studies, we are not interested in the raw data by itself, but use it to calculate measures of interest regarding the system's performance. For instance, in the example shown in Chapter 1, we saw that measures of interest include the average time that a customer has to way in the checkout queue. We sometimes also call these raw data *synthetic data*, since this is not the actual data that we would collect in the physical world. Synthetic data has by itself sparked interest in recent years due to its potential to enhance how we train and validate machine learning models, especially regarding data privacy and robustness, or when training data is expensive or scarce (Jordon et al. 2022; Breugel, Qian, and Schaar 2023).

In the rest of this chapter, we will introduce the basic principles and notions needed to understand how simulation works. We start with a gentle reminder of random numbers and distributions, and introduce standard methods of random number generation. We then move

on into stochastic processes and how discrete-event simulation works. After that, we present common statistical techniques to deal with the output data of simulations and conclude the chapter with considerations about verification and validation of simulation studies.

2.2 Dealing with Random Numbers

We refer to *random numbers* as realizations of random variables that follow probability distributions. The following elements completely determine the statistical behaviour of randon numbers:

- Their **type**: discrete or continuous?
- The form of their **probability distribution**: binomial, normal, exponential, Poisson, etc.
- The **joint** or **conditional** distributions associated with the phenomenon at hand.
- The specific **parameters** used for each probability distribution.

In this book, we will mainly deal with parametric probability distributions, although everything applies to non-parametric distributions as well. We will hint at specific differences when appropriate.

2.2.1 Pseudorandom Number Generators

In general, any procedure to generate random numbers is called a *pseudorandom number generator* (PRNB). A PRNB can be defined as a deterministic algorithm that, given an initial seed, produces a long sequence of numbers that mimic the statistical properties of truly random samples. Although the sequence is fully determined by the seed (so it is not truly random, hence *pseudorandom*), a good PRNG yields values that are uniformly distributed, have minimal serial correlation, and pass standard statistical tests. Important PRNG properties include period length, equidistribution, independence, speed, and reproducibility (the same seed reproduces the same sequence). For simulation work we typically prefer generators with very long periods and strong statistical quality while cryptographic applications require cryptographically secure PRNGs. PRNGs are used to produce uniform variates that are then transformed into other distributions via methods such as inverse transform sampling, acceptance–rejection, or composition.

Let's explore the properties of a specific PRNG, the Linear Congruential Generator (LCG) using the following Python code.

```
import numpy as np
from scipy.stats import chisquare
from collections import defaultdict
```

```python
class LCG:
    """
    X(n+1) = (a * X(n) + c) mod m
    """
    def __init__(self, seed, a, c, m):
        self._state = seed
        self.a = a
        self.c = c
        self.m = m
        self.seed = seed

    def next_int(self):
        """Generates the next pseudo-random integer
        in the sequence."""
        self._state = (self.a * self._state + self.c) % self.m
        return self._state

    def generate(self, size):
        """Generates a sequence of integers and
        normalizes them to [0, 1)."""
        sequence_int = []
        sequence_float = []
        # Reset state to seed for sequence generation
        self._state = self.seed

        for _ in range(size):
            next_val = self.next_int()
            sequence_int.append(next_val)
            # Normalize to a float in [0, 1) by dividing by the modulus
            sequence_float.append(next_val / self.m)

        return np.array(sequence_int), np.array(sequence_float)
```

The LCG is one of the oldest and best known PRNG which are used to date. As can be seen in the code, it uses three integer parameters a, c and the modulo m and computes the next random number using the recurrence:

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.1}$$

Starting at $n = 0$, we initialize X_0 to the random seed provided.

We can now use the generator as follows:

```
# LCG Parameters (a 'poor' LCG to highlight the deterministic nature)
# A small modulus (m) leads to a short period and visible patterns.
SEED = 42
A = 65  # Multiplier
C = 1   # Increment
M = 2**10  # Modulus (1024) - A small M is used for demonstration purposes
SEQUENCE_SIZE = 100000

# 1. Initialize and Generate Sequence
prng = LCG(SEED, A, C, M)
int_sequence, float_sequence = prng.generate(SEQUENCE_SIZE)
int_sequence[:10]
```

```
array([683, 364, 109, 942, 815, 752, 753, 818, 947, 116])
```

We have now generated 100000 random numbers using LCG (only first 10 are shown). But how can we ensure if this PRNM works well in practice? We will look now at the **period length**, how to check for **uniformity** and how to assert if there is **serial correlation**.

Period length

The period length assesses the number of values generated before the sequence of states returns to the first value (the starting state) for the first time. Note that in general the longer, the better. Note that in this case, the maximum possible period is m, the modulo of the generator. We can calculate this with a simple Python function as follows:

```
def calculate_period(lcg_generator):
    """
    Calculates the period (cycle length) of the LCG.
    The period is the number of values generated before the sequence repeats.
    """
    initial_state = lcg_generator.seed
    current_state = initial_state

    # Check for the next state immediately after the seed to start the loop
    current_state = (lcg_generator.a * current_state + lcg_generator.c) % lcg_generator.m
    period = 1

    # Loop until the state returns to the initial seed
    while current_state != initial_state:
        current_state = (lcg_generator.a * current_state + lcg_generator.c) % lcg_generator.m
        period += 1
```

```
        # Safety break for potentially infinite loops in case of a non-standard LCG
        if period > lcg_generator.m:
            return f"Period is greater than modulus m ({lcg_generator.m}). Check parameters."

    return period

period = calculate_period(prng)
period
```

```
1024
```

So in this case, our generator reaches the maximum period (1024), which is the best we can do.

Tests for uniformity

We want the generated random numbers to be uniformly generated (we will see later how generate numbers with different distributions started with uniformly generated random numbers). For this, we use the χ^2 test for uniformity:

- Null Hypothesis (H_0): The generated numbers are uniformly distributed.
- Alternative Hypothesis (H_1): The generated numbers are not uniformly distributed.

The main idea of this test is to divide the generated numbers in intervals, and check whether those intervals contain roughly the same number of generated values (e.g. a flat histogram). Like in the classical χ^2 test, we calculate the expected E_i and the observed O_i frequencies for each range and calculate the χ^2 statistic as usual:

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

We can use the following Python function:

```
def chi_squared_uniformity_test(data_float, num_bins=10):
    """
    Statistical Test: Chi-Squared Goodness-of-Fit Test for Uniformity.
    """
    N = len(data_float)

    # 1. Bin the data to get observed frequencies
    # The bins are equal-sized intervals in [0, 1).
    observed_frequencies, _ = np.histogram(data_float, bins=num_bins, range=(0, 1))
```

```
    # 2. Calculate expected frequencies for a perfectly uniform distribution
    expected_frequency = N / num_bins
    expected_frequencies = np.full(num_bins, expected_frequency)

    # 3. Perform the Chi-Squared test
    # The 'chisquare' function compares observed and expected frequencies.
    # A small p-value (e.g., < 0.05) leads to rejection of H0, meaning non-uniformity.
    chi2_stat, p_value = chisquare(f_obs=observed_frequencies, f_exp=expected_frequencies)

    return chi2_stat, p_value, num_bins

chi2_stat, p_value_uniformity, num_bins = chi_squared_uniformity_test(float_sequence)
print(f'Chi2 statistic: {chi2_stat}, p-value: {p_value_uniformity}, number of bins: {num_bins
```

```
Chi2 statistic: 2.2074, p-value: 0.9877471315220641, number of bins: 10
```

In this case, the p-value is much higher than $\alpha = 0.05$ and we **cannot** reject H_0, so the numbers appear to be uniformly random.

Serial correlation

The next possible measure to check is the serial correlation between the numbers generated. The **Serial Correlation Check**, also known as **Autocorrelation at Lag 1**, is a diagnostic measure used to characterize and detect a fundamental weakness in simple Pseudorandom Number Generators (PRNGs), such as the Linear Congruential Generator (LCG). The main idea is that the correlation between immediately adjacent numbers (hence lag 1) should be zero.

To calculate this, we form two sequences: the generated numbers and the same sequence moved by one place:

$$S_n = \{X_1, X_2, X_3, \dots, X_{n-1}\}$$
$$S_{n+1} = \{X_2, X_3, X_4, \dots, X_n\}$$

And now we calculate the Pearson correlation coefficient between S_1 and S_2.

$$r = \frac{\sum(S_1 - \bar{S_1})(S_2 - \bar{S_2})}{\sqrt{\sum(S_1 - \bar{S_1})^2(S_2 - \bar{S_2})^2}}$$

Our goal is that r is as close to zero as possible (note that $r \in [-1, 1]$). Let's use the following code:

```python
def serial_correlation_check(data_float):
    """
    Characterization: Autocorrelation (Serial Correlation) Check.
    """
    # X_n: all values except the last one
    X_n = data_float[:-1]
    # X_{n+1}: all values except the first one
    X_n_plus_1 = data_float[1:]

    # Calculate the Pearson correlation coefficient (r)
    # The result is an array, we take the correlation between the two sequences (index 0, 1)
    correlation_matrix = np.corrcoef(X_n, X_n_plus_1)
    lag_1_correlation = correlation_matrix[0, 1]

    return lag_1_correlation

lag_1_correlation = serial_correlation_check(float_sequence)
print(f'The lag 1 correlation coefficient is {lag_1_correlation}')
```

```
The lag 1 correlation coefficient is 0.008943629579226285
```

While the value is low, it's not as close to zero as it should, which is a known weakness of the LCG (the generated numbers tend to fall onto a number of parallel hyperplanes). This is the reason why PRNM like the LCG are not normally used in practice. The de-facto standard for pseudorandom number generation in practice is the algorithm known as the **Mersenne Twister**. This is the default generator used in Python or MATLAB, and the preferred one for simulation purposes (but *not* for cryptographic purposes). The basic idea is to use a highly non-linear twisted generalized feedback shift register. Apart from being much faster than LCG, it passes the serial correlation chek with flying colors:

```python
import random

random.seed(SEED)

# Generate a sequence of random floats in the range [0.0, 1.0)
float_sequence_mt = np.array([random.uniform(0, 1) for _ in range(SEQUENCE_SIZE)])

# Serial Correlation Check
lag_1_correlation_mt = serial_correlation_check(float_sequence_mt)
print(f'The lag 1 correlation coefficient is {lag_1_correlation_mt}')
```

```
The lag 1 correlation coefficient is -0.000962673758206564
```

which is an order of magnitude better than the LCG.

2.3 Sampling Methods

We have now a method for generating *uniformly distributed* random numbers. But what about other widely used distributions, like normal, exponential, Poisson, etc? In this section, we will review three popular methods for this purpose: the **inversion method**, the **rejection sampling** method, the **Box-Muller transform** and the **mixture method**. For all three methods, the general problem is as follows: we start with a random variable $U \sim$ Uniform$(0, 1)$. We want to convert U into $X \sim f(x)$, where f is the target PDF of X.

Inversion method

Suppose that we know the CDF of the target distribution $F(x) = P(X \leq x)$, and assume that we can invert it to $F^{-1}(u)$. With this function, we can simply obtain X by

$$X = F^{-1}(U) \tag{2.2}$$

For instance, imagine our target distribution is the exponential, with density function $f(x) = \lambda e^{-\lambda x}$. Elementary calculus tells us that $F(x) = 1 - e^{\lambda x}$. It can be shown that the inverse is

$$F^{-1}(u) = -\frac{1}{\lambda} \ln(1 - u) \tag{2.3}$$

Since $U' = 1 - U$ is also uniform in $[0, 1]$, we can simply write $X = -\frac{1}{\lambda} \ln(U')$.

Rejection-sampling method

But what if our CDF is not easily invertible, or worse, we don't have any analytical expression for it? Suppose that, although we don't have f, we have a proposal distribution $g(x)$ so that it "envelopes" the target distribution in the sense that there is a constant c so that $f(x) \leq cg(x)$ (i.e., we **do** know the PDF). In this case, we can do the following:

1. Sample x from the proposal distribution $g(x)$.
2. Sample a uniform $U(0, 1)$ random variable u.
3. If $u < \frac{f(x)}{cg(x)}$, the candidate number x is accepted since it follows $f(x)$.
4. Otherwise, we repeat the procedure until we get a candidate accepted.

The trick now is to take a *bounded* uniform distribution as g that contains our target distribution f. Once we have this, we can generate samples from virtually any probability distribution without requiring its CDF or inverse.

Box-Muller transform

The next method is specialized towards generating values for the **normal distribution**, and is widely used in practice. We start by generating two uniform random numbers $u_1, u_2 \sim U(-1, 1)$.

- First, we calculte the sum of their squares $S = u_1^2 + u_2^2$.
- If $S \geq 1$ or $S = 0$, we reject both and return to the first step.
- Otherwise, we calculate $C = \sqrt{\frac{-2\ln(S)}{S}}$.
- We output two normally distributed random numbers as $z_1 = u_1 C$ and $z_2 = u_2 C$.

The random numbers generated follow a standard normal distribution $N(0, 1)$. For an arbitrary normal distribution $N(\mu, \sigma^2)$ we just scale using the standard transformation $X = \mu + \sigma Z$.

Mixture method

In the case that the target distribution can be expressed as a mixture of k different PDFs

$$f(x) = \sum_{i=1}^{k} p_i f_i(x), \text{ with } p_i \geq 0, \text{ and } \sum_{i=1}^{k} p_i = 1 \tag{2.4}$$

Then we can use the following methods to sample from $f(x)$:

- Choose randomly an index $i \in I$ from the set of indices $I = \{1, 2, \ldots, k\}$. This is done by generating a random number $u \sim U(0, 1)$ and choosing the least index j so that $\sum_{i=1}^{j} p_i < u$.
- Generate a random variable x from $f_i(x)$ by using any of the aforementioned methods.

This is a suitable method, for instance, to generate random numbers that follow a **Gaussian Mixture Model (GMM)**. In this case, we just sample an index and generate a random number according to the Box-Muller method, scaling accordingly if necessary.

2.4 Stochastic Processes

Now that we know how to generate random numbers and sample from different distributions, we need to understand how they interact over time in a simulation study. This is the realm of *stochastic processes*.

A stochastic process $\{X(t), t \in T\}$ is a collection of random variables indexed by time. The set T can be:

- **Discrete**: $T = \{0, 1, 2, ...\}$ or $T = \{t_0, t_1, t_2, ...\}$
- **Continuous**: $T = [0, \infty)$ or $T = [a, b]$

For each fixed time t, $X(t)$ is a random variable. For each sample path (realization) of the process, $X(t)$ is a function of time. The nature of the state space (the set of possible values of $X(t)$) leads to different classifications:

- **Discrete state space**: The process can only take certain values (e.g., number of customers in a queue)
- **Continuous state space**: The process can take any value in an interval (e.g., temperature in a reactor)

Understanding stochastic processes is crucial for simulation because they model how random events unfold over time, which is exactly what we need to simulate complex systems with uncertainty.

Stationary and non-stationary processes

A stochastic process $\{X(t)\}$ is said to be **stationary** if its statistical properties do not change over time. More formally:

- The mean function is constant: $E[X(t)] = \mu$ for all $t \in T$.
- The variance function is constant: $Var[X(t)] = \sigma^2$ for all $t \in T$.
- The autocovariance function depends only on the time difference: $Cov[X(t), X(s)] = C(|t - s|)$.

In contrast, a **non-stationary** process has statistical properties that vary with time. For example:

- A random walk is non-stationary because its variance increases with time.
- A seasonal time series with repeating patterns is non-stationary.
- A process with a trend component is non-stationary.

Stationarity is an important property in simulation because it allows us to:

- Make meaningful predictions about future behavior.
- Estimate parameters from historical data.
- Apply many statistical techniques that assume stationarity.

In the following we will see examples of different stochastic processes and how to simulate them efficiently.

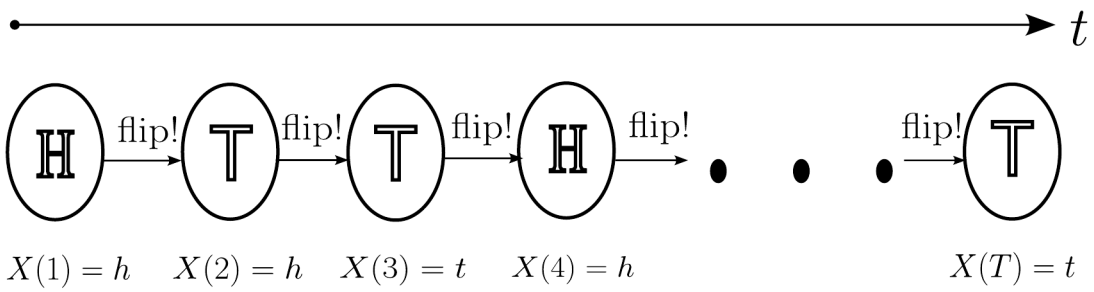


Figure 2.1: A typical Bernoulli process.

2.4.1 Bernoulli and Binomial Processes

We call a Bernoulli process a sequence of independent trials with two possible outcomes ("success/failure"). The classical example is flipping a coin independently for n times.

We formally define a **Bernoulli process** as follows:

- Each variable takes values from the set $\{0, 1\}$. In our example, the value 0 could stand for heads, and 1 for tails.
- The probability of sucess (per convention, $P(success) = P(X(t) = 1)$) is the same for every trial.
- The outcome for a given trial is independent of any other trial. So in our case, each coin flip is independent of all the others.

$$P(X(t) = 1) = p \text{ and } P(X(t) = 0) = 1 - p$$

The **Bernoulli counting process** $\{N(t)\}$ is just the sum of the outcomes of the first t trials.

$$N(t) = \sum_{i=1}^{t} X(i)$$

where $X(i)$ is a Bernoulli random variable with parameter p. So $N(t)$ counts the number of successes occurred in trials 1 to t. Some properties of the counting process are:

- Each variable $N(t)$ takes values in the set $\{0, 1, 2, ..., t\}$.
- The **increment** of this process $N(t) - N(t-1)$ is equals to the result of the t-th trial, $X(t)$.
- $N(t)$ follows a **binomial distribution**:

$$P(N(t) = k) = \binom{t}{k} p^k (1-p)^{z-k} \tag{2.5}$$

Let's see how to simulate a Bernoulli process:

```
import random

def simulate_bernoulli_process(num_trials, success_probability):
    """
    Simulates a Bernoulli process for a given number of trials and success probability.
    """
    if not 0 <= success_probability <= 1:
        raise ValueError("Success probability must be between 0 and 1.")

    bernoulli_outcomes = [
        1 if random.random() < success_probability else 0
        for _ in range(num_trials)
    ]

    return bernoulli_outcomes
```

Here we are just using Python's default random number generator (the Mersenne Twister) to check if the generated number is below or above the sucess probability, in the former case we count a success, otherwise a failure. We can now run the simulation for both Bernoulli and Bernoulli counting processes and visualize the results.

```
import matplotlib.pyplot as plt
import numpy as np
P_SUCCESS = 0.3  # Probability of success (p)
NUM_TRIALS = 50  # Total number of trials
random.seed(42)  # Set seed for reproducibility
outcomes = simulate_bernoulli_process(NUM_TRIALS, P_SUCCESS)
counting_process = np.cumsum(outcomes)
```

In this case, the Bernoulli counting process is just the cumulative sum of the generated Bernoulli process.

```
trials = np.arange(1, NUM_TRIALS + 1)

# Subplot 1: The Bernoulli Process (Outcomes)
plt.subplot(2, 1, 1)
```

```
plt.step(trials, outcomes, where='post', color='blue', linewidth=2)
plt.yticks([0, 1], ['Failure (0)', 'Success (1)'])
plt.title('Bernoulli Process (Sequence of Trials)')
plt.xlabel('Trial Number (i)')
plt.ylabel('Outcome ($X_i$)')
plt.grid(axis='y', linestyle='--')
plt.ylim(-0.1, 1.1)

# Subplot 2: The Bernoulli Counting Process (Cumulative Sum)
plt.subplot(2, 1, 2)
plt.step(trials, counting_process, where='post', color='red', linewidth=2)
plt.title('Bernoulli Counting Process (Cumulative Successes)')
plt.xlabel('Trial Number (t)')
plt.ylabel('Count ($N(t)$)')
plt.axhline(NUM_TRIALS * P_SUCCESS, color='green', linestyle=':', label='Expected Count')
plt.legend()
plt.grid(True, linestyle='--')

plt.tight_layout()
plt.show()
```

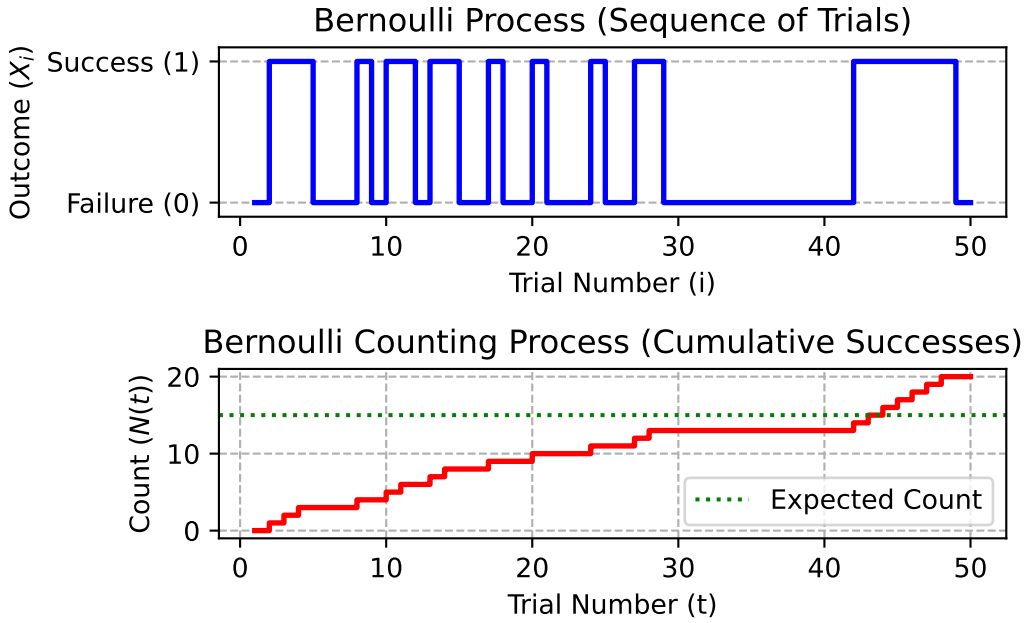


Figure 2.2: Plots of Bernoulli and counting processes

For this plot, we have generated $T = 50$ trials with a success probability of $p = 0.3$. The Bernoulli process outcomes are depicted in the upper part of the figure. In the lower part, the evolution of the counting process $\{N(t)\}$ is shown. The expected count is just calculated by multiplying the number of trials with the success probability, which results in 15 as the expected number of successes.

2.4.2 Poisson Processes

We now switch from *continuous* to *discrete* time. The **Poisson process** is a canonical continous time counting process. It tracks the number of occurrences of a given event up to a certain time t. Formally, let $\lambda > 0$ be the rate of occurrences of the event in question. We define the Poisson process $\{N(t), t \geq 0\}$ as follows:

- As initial condition, we set $N(0) = 0$.
- If we consider two non-overlapping intervals $[s, t] \cap [u, v] = \emptyset$, then the increments on those intervals are **independent** of each other:

$$N(t) - N(s) \text{ is independent of } N(v) - N(u), [s, t] \cap [u, v] = \emptyset \tag{2.6}$$

- At any interval of length $\tau = t - s$, the increments follow a Poisson distribution with parameter $\lambda\tau$.

$$N(t) - N(s) \sim \text{Poisson}(\lambda\tau) \tag{2.7}$$

$$P(N(t) - N(s) = k) = e^{-\lambda\tau}\frac{(\lambda\tau)^k}{k!} \tag{2.8}$$

The Poisson process is one of the most widely used stochastic models in science, engineering, and finance due to its ability to accurately model random, independent, and rare events occurring over time or space. Its practical relevance stems directly from the three conditions of its definition (independent, stationary increments, and the resulting Poisson distribution). For instance, the arrival of customer calls in a call center can be accurately modeled using Poisson processes. In engineernig, the number of defects found per area or volume (e.g. on a silicon wafer) can be also modeled this way (in this case, the rate is *spatial* rather than *temporal*, but the concept remains the same). In general, customer arrivals (at a hospital, a bank, a supermarket) can be also modeled using a Poisson process.

One of the main properties that makes the Poisson process useful in practice is the fact that is *memoryless*: the time until the next event is independent of how long it has been since the previous event, which aligns well with phenomena found in practice. Another advantage is that since the process is based in the Poisson (for the number of events per time interval) and the

exponential distributions (for the time between events), the process is mathematically well tractable and closed-form solutions exists for many situations of interest.

The next code snippet demonstrates how to simulate a Poisson process.

```
import numpy as np
import matplotlib.pyplot as plt
import random

def simulate_poisson_process(rate_lambda, max_time):
    if rate_lambda <= 0 or max_time <= 0:
        raise ValueError("Rate and max_time must be positive.")

    # Set up the event simulation
    current_time = 0.0
    event_times = [0.0]  # Start at time 0 with the first "event"

    while current_time < max_time:
        # 1. Generate the next inter-arrival time
        time_until_next_event = random.expovariate(rate_lambda)

        # 2. Update the cumulative time
        current_time += time_until_next_event

        # 3. Record the event time if it's within the simulation duration
        if current_time < max_time:
            event_times.append(current_time)

    # Calculate the event count at each recorded time
    num_events = len(event_times) - 1 # N(0)=0 is the first entry
    num_events_at_t = list(range(num_events + 1))

    return event_times, num_events_at_t
```

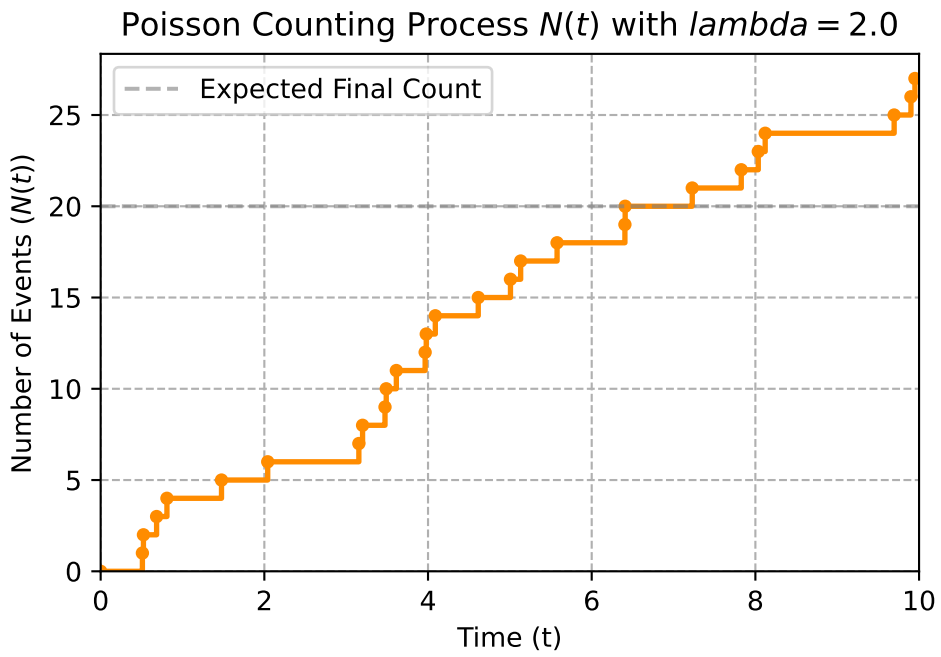
In this code we exploit the fact that the interarrival times are exponentially distributed, so we sample a time and count it as a new event. We can now use the code to simulate the process:

```
RATE_LAMBDA = 2.0  # Average rate of 2 events per unit of time
MAX_TIME = 10.0    # Simulate over 10 time units

random.seed(42) # Set seed for reproducibility
event_times, cumulative_counts = simulate_poisson_process(RATE_LAMBDA, MAX_TIME)
event_times[:5]
```

```
[0.0,
 0.5100301436374005,
 0.5226945631587699,
 0.6835065951962527,
 0.8097996880313094]
```

Like in the previous case, we can perform some visualizations to help understand the evolution of the process.

```
# Plot the step function of the counting process N(t)
# We use drawstyle='steps-post' to create the classic step-function look
plt.plot(event_times, cumulative_counts, drawstyle='steps-post', color='darkorange', linewid

# Plot the expected count line
plt.axhline(RATE_LAMBDA * MAX_TIME, color='gray', linestyle='--', alpha=0.6, label='Expected

plt.title(f'Poisson Counting Process $N(t)$ with $lambda = {RATE_LAMBDA}$')
plt.xlabel('Time (t)')
plt.ylabel('Number of Events ($N(t)$)')
plt.xlim(0, MAX_TIME)
plt.ylim(bottom=0)
plt.grid(True, linestyle='--')
plt.legend()
plt.show()
```



2.4.3 Markov Processes

The next type of process is fundamental for many practical applications and for simulation approaches that we will see in the rest of the book: **Markov processes**. A Markov process is a stochastic process that satisfies the Markov property, which states that the future state of the process depends only on the current state and not on the sequence of states that preceded it. Formally, a discrete-time Markov chain $\{X_t, t \geq 0\}$ satisfies:

$$P(X_{t+1} = j \mid X_t = i, X_{t-1} = i_{t-1}, \ldots, X_0 = i_0) = P(X_{t+1} = j \mid X_t = i)$$

This property is known as the **memoryless** or **Markov property**. The process is fully characterized by its **state space** (the set of possible states) and the **transition probabilities** between states. Both the state space and the time parameter can be either *discrete* or *continuous*. In the discrete-time case, we normally call these processes **Markov chains**.

Transition Matrix

For a Markov chain with a finite state space $S = \{1, 2, \ldots, N\}$, the transition probabilities can be represented as a matrix $\mathbf{P}$, where the entry $\mathbf{P}_{ij}$ represents the probability of transitioning from state i to state j:

$$\mathbf{P}_{ij} = P(X_{t+1} = j \mid X_t = i)$$

The rows of the matrix P must sum to 1, as they represent probability distributions:

$$\mathbf{P} = \begin{pmatrix} P_{11} & P_{12} & \dots \\ P_{21} & P_{22} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

$$\sum_{j \in S} P_{ij} = 1, \quad \forall i \in S$$

Phenomena that can be naturally modeled using Markov chains include those where the state of the system in a given moment only depends on the moment immediately before, not on those before. For instance, in genetics, the frequency of a given allele in a generation only depends on the frequencies recorded in the previous generation. Or in inventory management, where decisions about the inventory levels are solely based on the current stock level.

Simulation example

Let's see how to simulate a simple Markov chain with a discrete state space. Assume we want to model a heavily simplified weather system with three states: sunny (1), cloudy (2) and rainy (3). For the state transition matrix, we have:

$$\mathbf{P} = \begin{pmatrix} 0.8 & 0.15 & 0.05 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.4 & 0.5 \end{pmatrix}$$

So for instance, the probability of a rainy day given that yesterday was cloudy is $P_{23} = 0.2$.

```
import numpy as np
import matplotlib.pyplot as plt
import random

def simulate_dtmc(start_state, transition_matrix, num_steps):
    # 1. Initialize the chain
    num_states = transition_matrix.shape[0]
    current_state = start_state
    state_history = [current_state]

    # 2. Iterate for the specified number of steps
    for _ in range(num_steps):
        # Get the probability distribution for the next state, based on the current state (r
        probabilities = transition_matrix[current_state, :]

        # Select the next state using the probabilities
        # np.random.choice selects a state from the possible states (0 to num_states-1)
```

```
        # based on the corresponding probabilities.
        next_state = np.random.choice(
            a=np.arange(num_states),
            p=probabilities
        )

        # Update and record
        current_state = next_state
        state_history.append(current_state)

    return state_history
```

We now setup the simulation for our example:

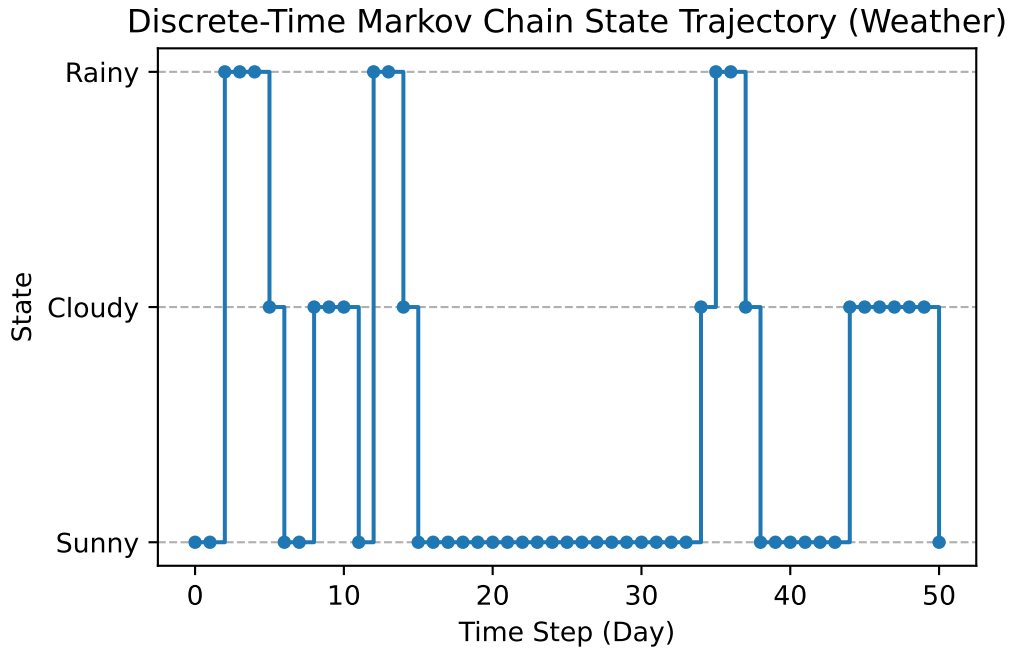
```
# Define the Transition Probability Matrix (P)
P = np.array([
    [0.80, 0.15, 0.05],  # From Sunny (0)
    [0.20, 0.60, 0.20],  # From Cloudy (1)
    [0.10, 0.40, 0.50]   # From Rainy (2)
])

# Define the State Space for interpretation
STATE_MAP = {0: 'Sunny', 1: 'Cloudy', 2: 'Rainy'}

START_STATE_INDEX = 0  # Start on a Sunny day
NUM_SIMULATION_STEPS = 50 # Simulate 50 days

# 3. Run the Simulation
np.random.seed(42) # Seed for reproducibility
simulated_states = simulate_dtmc(START_STATE_INDEX, P, NUM_SIMULATION_STEPS)
simulated_weather = [STATE_MAP[s] for s in simulated_states]
simulated_weather[:10]
```

```
['Sunny',
 'Sunny',
 'Rainy',
 'Rainy',
 'Rainy',
 'Cloudy',
 'Sunny',
 'Sunny',
```

```
 'Cloudy',
 'Cloudy']
```

Now let's plot the sequence of states against time:

```
time_steps = np.arange(NUM_SIMULATION_STEPS + 1)

plt.plot(time_steps, simulated_states, marker='o', linestyle='-', drawstyle='steps-post', ma
plt.yticks([0, 1, 2], [STATE_MAP[0], STATE_MAP[1], STATE_MAP[2]])
plt.title('Discrete-Time Markov Chain State Trajectory (Weather)')
plt.xlabel('Time Step (Day)')
plt.ylabel('State')
plt.grid(True, axis='y', linestyle='--')
plt.show()
```



Continuous-Time Markov Processes

In our previous example, we have assumed that the time index takes values in a discrete set. Let's now consider the situation where time is continuous: $\{X(t), t \geq 0\}$. In this case, the system might change its state at any time, instead of being governed by a fixed transition probability P_{ij}. We now assume that the time spent in the current state i is **itself a random variable** which is continuous and exponentially distributed. Let's denote this **holding time** by τ_i. We have:

$$\tau_i \sim \text{Exponential}(q_i) \tag{2.9}$$

where the parameter q_i is called the **total exit rate** or intensity from state i. The Markov property is now given by the fact that the exponential distribution is memoryless: the time remaining in state i is independent on how long the system has been there already.

So how do we decide where which state to jump next? For that, let q_{ij} denote the *instantaneous rate of moving* from state i to j. Our parameter q_i becomes then $q_i = \sum_{i \neq j} q_{ij}$. We can convert this into a transition probability P_{ij} using $P_{ij} = \frac{q_{ij}}{q_i}$. Instead of talking about a transition matrix $\mathbf{P}$, we define a **rate** or **Q-Matrix Q** where $Q_{ij} = q_{ij}$ for $i \neq j$ and $Q_{ii} = -q_i$ (the diagonal elements are defined with a minus sign because we want the sum of each row -the sum of leaving and entering state i- to be 0).

We can simulate a continuous-time Markov chain in the following way:

```
import numpy as np
import matplotlib.pyplot as plt

def simulate_ctmc(Q_matrix, start_state, max_time):
    current_time = 0.0
    current_state = start_state

    times = [0.0]
    states = [current_state]

    num_states = Q_matrix.shape[0]

    while current_time < max_time:
        exit_rate = -Q_matrix[current_state, current_state]

        if exit_rate <= 0:
            break

        time_step = np.random.exponential(scale=1/exit_rate)

        if current_time + time_step > max_time:
            break

        rates_row = Q_matrix[current_state, :].copy()

        rates_row[current_state] = 0
```

```
        jump_probabilities = rates_row / exit_rate

        next_state = np.random.choice(np.arange(num_states), p=jump_probabilities)

        current_time += time_step
        current_state = next_state

        times.append(current_time)
        states.append(current_state)

    times.append(max_time)
    states.append(states[-1])

    return times, states
```

This function first enters a classical simulation loop where time advances until a stopping time has been reached. On each iteration, we first sample a time using the inverse of the exit rate (the minus diagonal element of the Q-matrix) as the parameter of an exponential distribution. Then we determine the state we are goint to jump to by calculating the jump probabilities. The next state is then chosen according to the jump probabilities.

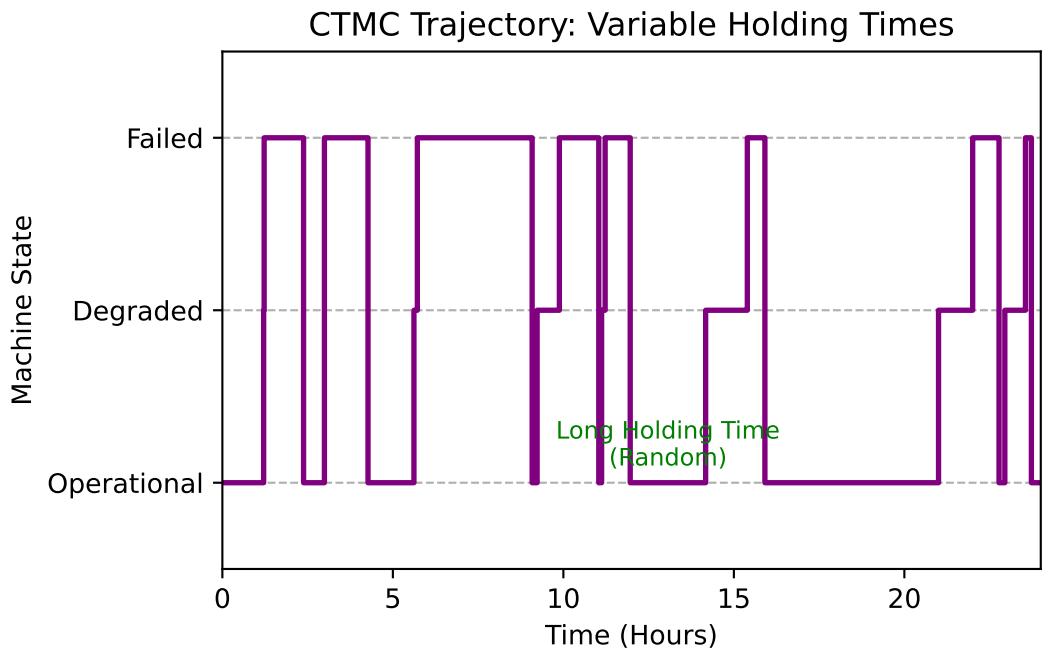
Let's now simulate a scenario where we want to study machine reliability. We define the state space as Operational (1), Degraded (2) or Failed (3). In the operational state, the machine works as expected, however it transitions to degraded often, but rarely to failed. In the degraded state, the machine transitions to failed quickly, since it is already not working properly (and never transitions to operational). Finally, in the failed state, the machine is broken and it will only transition to operational if the machine is repaired.

```
# Define the Generator Matrix (Q)
# Rows must sum to 0.
# Units: events per hour (rates)
Q = np.array([
# To:   Op(0)   Deg(1)  Fail(2)
    [-0.6,   0.5,    0.1],  # From Operational (Exit rate = 0.6)
    [ 0.0,  -2.0,    2.0],  # From Degraded    (Exit rate = 2.0 -> Very fast exit)
    [ 1.0,   0.0,   -1.0]   # From Failed      (Exit rate = 1.0 -> Repair time)
])

STATE_NAMES = {0: 'Operational', 1: 'Degraded', 2: 'Failed'}
START_STATE = 0
DURATION = 24.0  # Simulate 24 hours
```

```
# --- Run Simulation ---
np.random.seed(101)
sim_times, sim_states = simulate_ctmc(Q, START_STATE, DURATION)
```

Note that the units of **Q** are *rates* or events per unit of time. Here the unit of time can be freely chosen, what is important is the fact that each number represents the number of events that happen, on average, in the unit of time, and that each row has to sum up to 0.

```
plt.step(sim_times, sim_states, where='post', color='purple', linewidth=2)

plt.yticks([0, 1, 2], [STATE_NAMES[0], STATE_NAMES[1], STATE_NAMES[2]])
plt.xlabel('Time (Hours)')
plt.ylabel('Machine State')
plt.title('CTMC Trajectory: Variable Holding Times')
plt.grid(True, axis='y', linestyle='--')
plt.xlim(0, DURATION)
plt.ylim(-0.5, 2.5)

for i in range(len(sim_times)-1):
    if sim_states[i] == 0 and (sim_times[i+1] - sim_times[i] > 2.0):
        mid_point = (sim_times[i] + sim_times[i+1]) / 2
        plt.text(mid_point, 0.1, "Long Holding Time\n(Random)", ha='center', fontsize=9, colo
        break

plt.show()
```



In this visualization, an exemplary long holding time (the first longer than 2 hours in the operational state) is highlighted. As can be seen, the holding times are highly variable (compare to the discrete-time Markov chain previously). Note also that the system spends very little time in the degraded state. This is because the exit rate (the diagonal element) is very high, so the machine passes this state quickly on its way to the failed state. Since the exit rate is quite low in the operational state, the machine stays there for longer periods.

Continuous-State Markov Processes

The last important case we need to take a closer look at is when the state space S is **continuous**. a continuous range of values, such as the position of a particle in space, the temperature of a system, or the level of a reservoir. In this case, the process is often described by a **stochastic differential equation (SDE)**, which governs the evolution of the state over time. The time parameter can be in this case either discrete or continuous, so the corresponding considerations apply. The main challenge is how to deal with state transition probabilities. For this purpose, the most general solution is to switch from transition matrices to **transition probability density functions** that we denote by $f(x_t|x_{t-1})$.

One particularly relevant example of Markov process is the **random walk**. The main idea is to let the process evolve from an initial state by applying random, independent perturbations:

$$X_t = X_{t-1} + \epsilon_t,\ \epsilon_t \sim N(0, \sigma^2) \tag{2.10}$$

It is important to stress the fact that the noise terms ϵ_t and the process steps X_{t-1} are independent. We can see that for random walks, the Markov property is fulfilled because X_t only depends on X_{t-1} and not on any values before. Random walks are used when modeling non-stationary time series, like stock prices without a drift.

Another important example is the **Brownian motion** or **Wiener process**. The Brownian motion is a stochastic process commonly used for modeling random movements of particles. Let's denote this process by $W(t)$. The defining characteristic of a Brownian motion is that its increments are **normally distributed** and **independent of the past**.

$$W(t+s) - W(s) \sim N(0, t-s) \tag{2.11}$$

There are many practical applications of this type of process. In finance, the Brownian motion is used to model option pricing (the **Black-Scholes** equation). In polymer physics, the random, wriggling moving of a long-chain polymer molecule in a solvent can be well modeled using the Brownian motion. In audio engineering and signal processing, it can be used for random noise generation (white and pink noise).

Simulation example

We will now see how to simulate a Brownian motion. In order to simulate this process (and like it, any continuous-time Markov process), we resort to the **discrete approximation method**. The main idea is to model a continuous path by adding small, independent, normally distributed random increments at small time steps (Δt). We use the following equation:

$$W(t_{i+1}) = W(t_i) + \sqrt{\Delta t} \cdot Z_i \tag{2.12}$$

The square root of the increment follows from the increments property of the Brownian motion: the variance of the increments has to be proportional to the difference between the time steps.

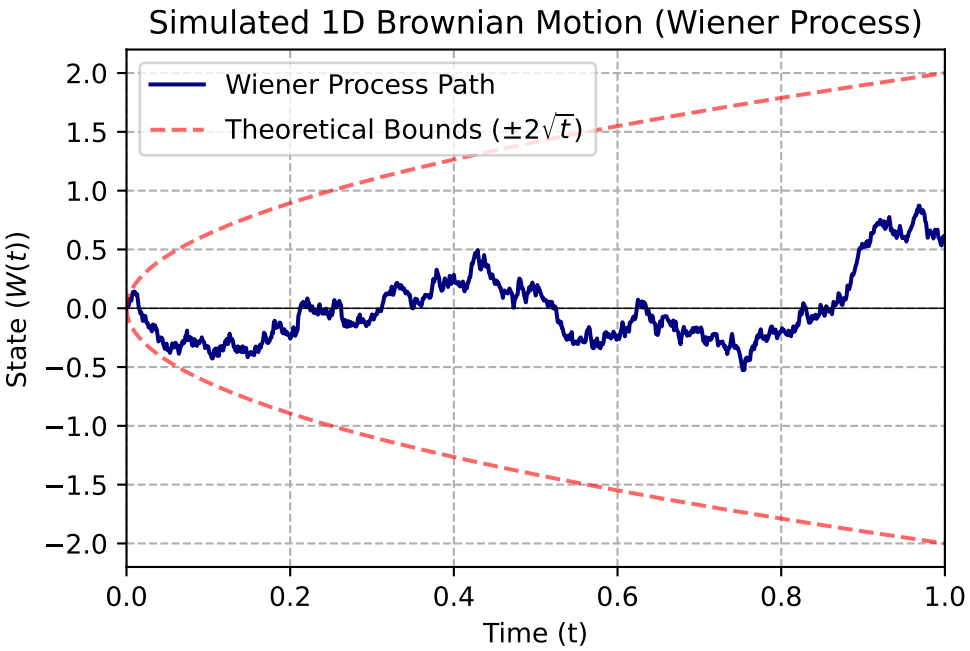
```
import numpy as np
import matplotlib.pyplot as plt

def simulate_brownian_motion(total_time, num_steps):
    if num_steps <= 0 or total_time <= 0:
        raise ValueError("num_steps and total_time must be positive.")

    dt = total_time / num_steps
    time_points = np.linspace(0, total_time, num_steps + 1)

    standard_deviation = np.sqrt(dt)

    Z_increments = np.random.normal(loc=0.0, scale=1.0, size=num_steps)
```

```
    dW_increments = standard_deviation * Z_increments

    brownian_path = np.concatenate(([0.0], np.cumsum(dW_increments)))

    return time_points, brownian_path
```

In the code, we first discretize the time by dividing the total time by the number of simulation steps. Then we calculate the standard deviation as the square root of the increment and proceed to simulate the increments Z_i. The final Brownian path is taken to be the cumulative sum of the increments.

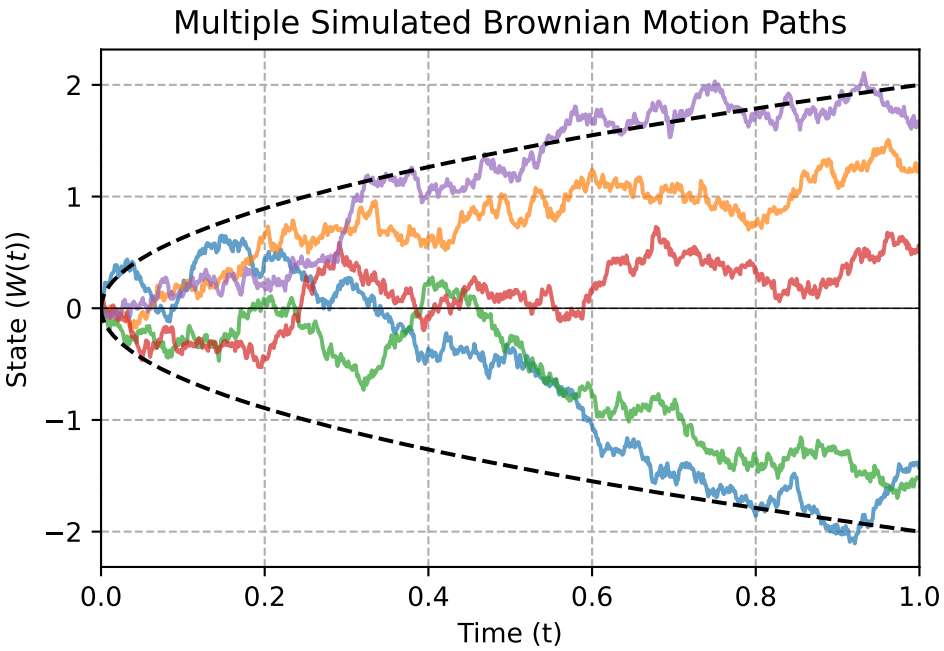
Let's now perform a simulation a visualize the results:

```
TOTAL_TIME = 1.0     # Simulate up to time T=1
NUM_STEPS = 1000     # Use 1000 steps for a smooth path

np.random.seed(42) # Seed for reproducibility
time, path = simulate_brownian_motion(TOTAL_TIME, NUM_STEPS)

plt.plot(time, path, label='Wiener Process Path', color='navy')

# The theoretical standard deviation at time t is sqrt(t)
std_dev = np.sqrt(time)
plt.plot(time, 2 * std_dev, 'r--', label=r'Theoretical Bounds ($\pm 2\sqrt{t}$)', alpha=0.6)
plt.plot(time, -2 * std_dev, 'r--', alpha=0.6)

plt.title('Simulated 1D Brownian Motion (Wiener Process)')
plt.xlabel('Time (t)')
plt.ylabel('State ($W(t)$)')
plt.axhline(0, color='black', linewidth=0.5)
plt.xlim(0, TOTAL_TIME)
plt.grid(True, linestyle='--')
plt.legend()
plt.show()
```



In this visualization, we plot one realization of a Brownian motion bounded by its theoretical bounds (using a standard deviation of $\sqrt{t}$ up and down). The next visualization shows several realizations of the same Brownian motion:

```
NUM_PATHS = 5
for i in range(NUM_PATHS):
    np.random.seed(i) # Use a different seed for each path
    _, multi_path = simulate_brownian_motion(TOTAL_TIME, NUM_STEPS)
    plt.plot(time, multi_path, alpha=0.7)

plt.plot(time, 2 * np.sqrt(time), 'k--', label='Theoretical Bounds', linewidth=1.5)
plt.plot(time, -2 * np.sqrt(time), 'k--', linewidth=1.5)
plt.title(f'Multiple Simulated Brownian Motion Paths')
plt.xlabel('Time (t)')
plt.ylabel('State ($W(t)$)')
plt.axhline(0, color='black', linewidth=0.5)
plt.xlim(0, TOTAL_TIME)
plt.grid(True, linestyle='--')
plt.show()
```



2.5 Chapter Summary

In this chapter, we introduced the fundamental concepts of simulation, focusing on random number generation, sampling methods, and stochastic processes. We explored pseudorandom number generators, including the Linear Congruential Generator and the Mersenne Twister, and discussed their properties and limitations. Sampling methods such as the inversion method, rejection sampling, and the Box-Muller transform were presented for generating random variables from various distributions. We then delved into stochastic processes, covering discrete and continuous-time processes like Bernoulli, Poisson, Markov chains, and Brownian motion, along with their simulation techniques. These concepts form the foundation for modeling and analyzing systems with inherent randomness.

2.6 Exercises

1. Implement a simulation for a geometric random walk, where the state evolves as:

$$X_{t+1} = X_t \cdot e^{\mu + \sigma Z_t}$$

   Here, $Z_t \sim N(0,1)$, μ is the drift, and σ is the volatility. Simulate the process for 1000 steps with $\mu = 0.01$ and $\sigma = 0.2$, starting at $X_0 = 1$. Plot the resulting trajectory.

2. Modify the Poisson process simulation to handle a time-dependent rate $\lambda(t)$. For example, use $\lambda(t) = 2 + \sin(t)$ over the interval $[0, 10]$. Plot the resulting event times and the cumulative count process.

3. Show that the exponential distribution is memoryless, i.e., for $T \sim \text{Exponential}(\lambda)$:

$$P(T > s + t \mid T > t) = P(T > s).$$

   Provide a detailed derivation.

4. Prove that the Poisson process has stationary increments, i.e., the number of events in any interval of length τ follows the same distribution, regardless of the starting time. Use the definition of the Poisson process and its properties in your proof.

3 Monte Carlo

3.1 What is Monte-Carlo Simulation?

We call **Monte-Carlo (MC) simulation methods** to a broad class of computational methods used to approximate solutions to complex problems which can't often be obtained by traditional analytical methods. The main idea of MC methods is to repeatedly sample randomly from pre-specified probability distributions to obtain numerical results that can be used to approximate the problem or phenomenon of interest.

Usually, each MC method comprises the following steps:

1. **Define a model:** The first step is to define a mathematical model of the phenomenon of interest, for instance stock prices, project duration, interest rates, etc.
2. **Assign probability distributions:** For any of the inputs defined in the mathematical model, we assign a **probability distribution** to each of them. These distributions can take any particular form which may be suited to the problem at hand (normal, exponential, Poisson, etc) and represent the range of possible values and their likelihood.
3. **Simulate and aggregate:** We run simulations of the model for large sample sizes and record the results for each run.

The final result is a **distribution of possible outcomes** and the probability of each outcome occurring, which provides a clear picture of the risk and potential variability inherent in the system.

The Monte Carlo method was developed in the 1940s by scientists working on the Manhattan Project, including Stanislaw Ulam and John von Neumann. The method was named after the Monte Carlo Casino in Monaco, reflecting the element of chance central to the technique. Initially used to solve problems in nuclear physics, the method has since been widely adopted across various fields, including finance, engineering, and computer science, due to its versatility and effectiveness in solving complex probabilistic problems.

3.1.1 The intuition behind Monte-Carlo simulation

Imagine that you want to know the probability of a particular outcome in a complex system with many uncertain variables that possibly interact in complicated ways. For instance, what

is the probability that a new product launch will be successful, given uncertain factors like manufacturing costs, consumer demand and competitor pricing?

Instead of defining a mathematical equation to calculate that specific probability, we just simulate the event a large number of times (which can be thousands or even millions). Think of it like running an experiment for a number of times, the only difference being that in MC simulation the experiments are not physical but *virtual*. Therefore, the element of chance and repeated trials is central to MC simulation, and hence its connection to casinos and related games like the roulette wheel.

By running experiments a repeated number of times, we build representative samples that we can use to aggregate and calculate probabilities and confidence intervals. For instance, in our example if it turns out that 20% of the simulations result in success, we can confidently say that the probability of achieving that goal is 20%.

Example: Estimating the value of π

Let's look at an example where we use MC simulation to approximate the value of the constant π, which is depicted in Figure 3.1. Imagine you are throwing darts into this figure. When a dart hits the inside of the circle, we color it blue. Otherwise, we color it green. From basic arithmetic, we know that the ratio between the area of the circle (which is $\pi r^2 = \pi$ because $r = 1$) and the area of the square (a 2×2 square with area 4) is $\pi/4$. This means that if we had an approximation for these two areas, we could also approximate the value of π by dividing the circle's area by the square's area and multiplying by 4.

As you start throwing darts and counting which darts hit the inside and which ones the outside, you can estimate the ratio between the areas by just counting the number of blue dots and dividing by the total number of dots. However, for this to work, we need to assume that you are very bad at darts: concretely, that your darts will be uniformly distributed inside of the square. Under this conditions, if you continue throwing more and more darts and count accordingly, your approximation for π will get better and better. That's MC simulation in a nutshell!

Let's use some code to make this more concrete:

```
import random
import math

def estimate_pi_monte_carlo(num_iterations):
    points_inside_circle = 0

    inside_x, inside_y = [], []
    outside_x, outside_y = [], []

    for _ in range(num_iterations):
        x = random.uniform(0, 1)
```

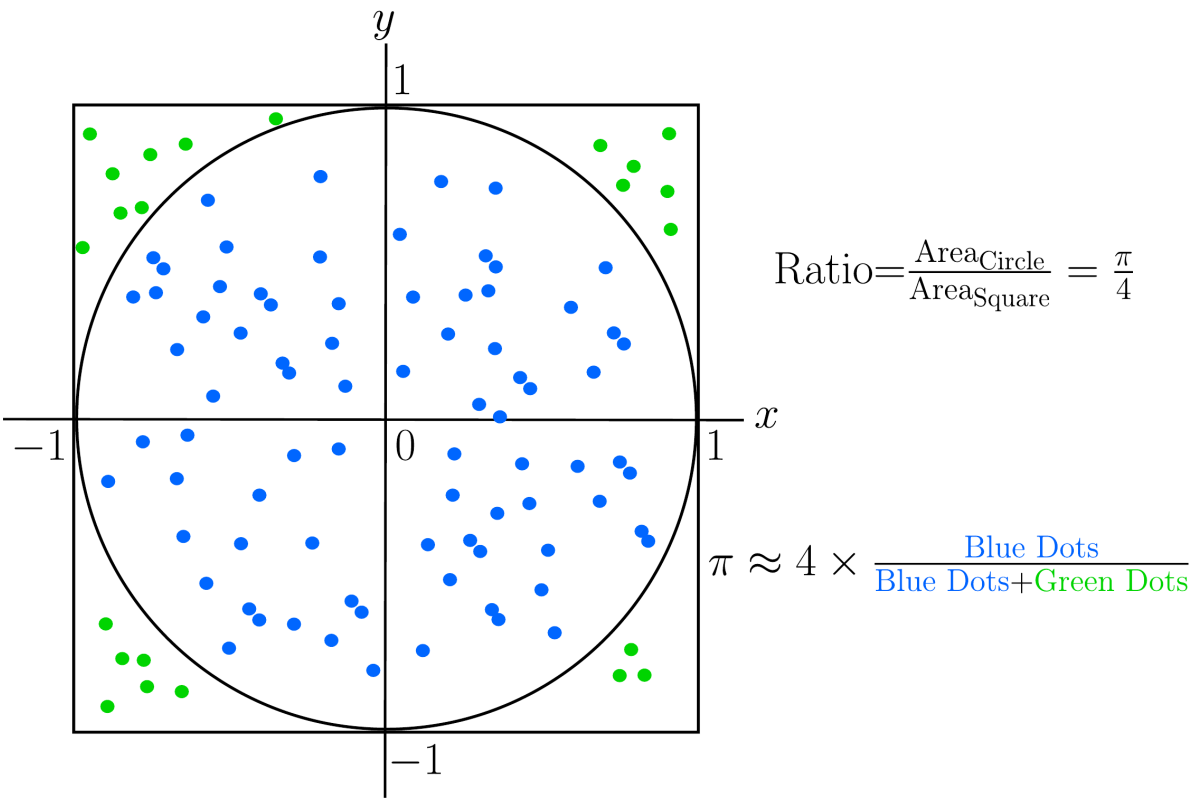


Figure 3.1: Example of Monte-Carlo simulation to calculate π.

```
        y = random.uniform(0, 1)

        distance_squared = x**2 + y**2

        if distance_squared <= 1:
            points_inside_circle += 1
            inside_x.append(x)
            inside_y.append(y)
        else:
            outside_x.append(x)
            outside_y.append(y)

    pi_estimate = 4 * (points_inside_circle / num_iterations)
    return pi_estimate

N = 100000  # 100,000 darts
estimated_pi = estimate_pi_monte_carlo(N)
estimated_pi
```

```
3.14212
```

In this code, we make use of the modeling fact that points (x, y) inside of the circle satisfy $x^2 + y^2 \leq 1$.

3.1.2 Limitations

Albeit powerful, MC simulation has a number of limitations that need to be taken into account.

- **Variance:** Since MC is an inherently probabilistic method, the estimates have a degree of statistical uncertainty. Therefore different simulation runs will typically produce different estimates.
- **Slow convergence:** In order to reduce this variance, the number N of iterations (samples) has to be dramatically increased. In general, the standard error decreases like $1/\sqrt{N}$. So to halve the error, it's not enough to *double* the number of samples, but it needs to be *four times larger*.
- **Random number generators:** Here the quality of the PRNG used (see Chapter 2) plays a critical role. The used generator has to produce samples which are as random and independent as possible or otherwise MC won't work.

In the next sections, we will uncover the foundations of MC simulation. First we will explore the core concepts and mathematical foundations behind it. After that, we will study classical MC techniques and how to calculate confidence intervals for their outputs. We will conclude this chapter with Markov Chain Monte Carlo (MCMC), one essential technique widely used in machine learning.

3.2 Core Concepts

The core idea of MC simulation is to leverage the **Law of Large Numbers** to approximate complex quantities: the average of a random variable over a large number of samples will converge to its expected value. This allows us to approximate quantities that are difficult or impossible to compute analytically by simulating random samples and calculating their average.

Mathematically, the goal of MC simulation is to estimate the expected value of a function of a random variable $E[g(X)]$. The true expected value is defined by the integral:

$$E[g(x)] = \int_{-\infty}^{\infty} g(x)f(x)dx \tag{3.1}$$

Instead of solving this (often intractable) integral analytically, the MC method provides an approximation:

1. **Sampling:** We start by generating N independent and identically distributed (iid) random samples $X_1, X_2, ..., X_N$ from the probability distribution of X (using any of the methods seen in Chapter 2).
2. **Evaluation**: We now evaluate the target function g on each of the samples: $g(X_1), g(X_2), ..., g(X_N)$.
3. **Estimation**: The MC estimate $\hat{E}[g(X)]$ is calculated as the sample average:

$$\hat{E}[g(X)] = \frac{1}{N}\sum_{i=1}^{N} g(X_i) \tag{3.2}$$

We call $\hat{E}[g(X)]$ the **Monte Carlo estimator**. Note that we can approximate *any integral* with MC simulation using a simple trick. Assume we want to calculate the following integral:

$$I = \int_a^b h(x)dx$$

We can reframe this integral as the expectation of a uniform random variable in the interval $[a, b]$. Such a random variable has a pdf $f(x) = 1/(b-a)$. Now we have:

$$
\begin{aligned}
I &= \int_a^b h(x)dx \\
&= (b-a)\int_a^b h(x)\frac{1}{b-a}dx \\
&= (b-a)\int_a^b h(x)f(x)dx \\
&= (b-a)E[h(X)] \quad \text{where } f(x) = \frac{1}{b-a}
\end{aligned}
$$

So we can approximate I by:

$$
\hat{I} = (b-a)\hat{E}[h(X)] = (b-a) \times \left(\frac{1}{N} \sum_{i=1}^{N} h(X_i) \right) \tag{3.3}
$$

3.2.1 Statistical properties of the MC estimator

Recall that an estimator $\hat{\theta}$ of a parameter θ is said to be *unbiased* iff its expected value is equals to its true value $E[\hat{\theta}] = \theta$. The MC estimator is an **unbiased** estimator of the expected value of a function of a random variable:

$$
\begin{aligned}
E[\hat{E}[g(X)]] = E\left[\frac{1}{N} \sum_{i=1}^{N} g(X_i) \right] = \frac{1}{N} \sum_{i=1}^{N} E[g(X_i)] = \\
\overset{iid}{=} \frac{1}{N} \sum_{i=1}^{N} E[g(X)] = \frac{1}{N} N \cdot E[g(X)] = E[g(X)]
\end{aligned}
$$

By the law of large numbers, we know that $\hat{E}[g(X)]$ converges in probability to $E[g(X)]$. Additionally, by the Central Limit Theorem (CLT), the standard error is proportional to $1/\sqrt{N}$.

$$
\text{Standard error} \propto \frac{\sigma}{\sqrt{N}} \tag{3.4}
$$

where σ is the standard deviation of $g(X)$, which explains the slow convergence in general of MC simulation.

3.2.2 The Law of Large Numbers

The correctness guarantee for MC simulation comes from the **law of large numbers**, as mentioned before. We now give a more formal justification of why MC simulation does indeed arrive at the correct answer.

Let $X_1, X_2, \ldots, X_n$ be a sequence of iid random variables, and let $\mu = E[X]$ denote the true mean of the distribution of the X_i. We calculate the *sample average* of the sequence as:

$$\bar{X}_n = \frac{X_1 + X_2 + \cdots + X_n}{n}$$

The law of large numbers states in its strong version that

$$P\Big(\lim_{n\to\infty} \hat{X}_n = \mu\Big) = 1 \tag{3.5}$$

That is, in the limit $n \to \infty$, the probability of the limit converging to the true mean μ is equals to 1. In our case, the true expectation μ corresponds to the quantity of interest that we wish to calculate, and the random variables X_i correspond to one iteration of the simulation. The sample average $\hat{X}_n$ represents then our current estimate of the true value μ.

We can model the simulation using the following probabilistic model. We can imagine each realization X_i to be the true value μ perturbed by a random noise term ϵ_i with $E[\epsilon_i] = 0$: $X_i = \mu + \epsilon_i$. Now we average over n realizations:

$$\hat{X}_n = \frac{1}{n}\sum_{i=1}^{n}(\mu + \epsilon_i) = \mu + \frac{1}{n}\sum_{i=1}^{n}\epsilon_i$$

But because the noise terms cancel out over time, we have $\frac{1}{n}\sum \epsilon_i \to 0$ as $n \to \infty$, and we end up in the limit with $\bar{X}_n \approx \mu$.

Note that the independece assumption of the X_i is *critical*, since otherwise the error terms would compound rather than cancel out. This is the reason why only high-quality PRNG are used with MC simulation.

3.2.3 Confidence intervals

In order to quantify the uncertainty given by a MC estimate, confidence intevals represent an appropriate tool. For the calculation, we rely on the **Central Limit Theorem** already mentioned above. By this theorem, we know that the distribution of the *sample mean* tends towards a **normal distribution** when the number of iterations becomes large. Formally:

$$\sqrt{n}(\bar{X}_n - \mu) \sim N(0, \sigma^2)$$

Where $\text{Var}(X_i) = \sigma^2$. The consequence is that we can use Z-values of the standard normal distribution to calculate the confidence intervals:

1. Calculate the sample standard deviation:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X}_n)}$$

2. Calculate the standard error $SE = s/\sqrt{n}$.
3. Choose a confidence level $(1 - \alpha)$ (usually, 95%, so $\alpha = 0.05$).
4. The confidence interval is then:

$$CI = \bar{X}_n \pm \left(z_{\alpha/2} \times \frac{s}{\sqrt{n}}\right)$$

where $z_{\alpha/2}$ represents the $\alpha/2$-level quantile of the standard normal distribution. As an example, consider the following implementation for the estimate of π using Python code:

```
import numpy as np
import scipy.stats as stats

def monte_carlo_with_ci(num_iterations):
    x = np.random.uniform(0, 1, num_iterations)
    y = np.random.uniform(0, 1, num_iterations)

    inside_circle = (x**2 + y**2) <= 1
    values = inside_circle * 4.0

    mean_estimate = np.mean(values)
    std_dev = np.std(values, ddof=1) # ddof=1 for sample standard deviation
    standard_error = std_dev / np.sqrt(num_iterations)

    # 4. Calculate 95% Confidence Interval
    # z-score for 95% is 1.96, or strictly: stats.norm.ppf(0.025)
    z_score = stats.norm.ppf(0.025)
    margin_of_error = z_score * standard_error

    lower_bound = mean_estimate + margin_of_error
    upper_bound = mean_estimate - margin_of_error
```

```
    return mean_estimate, lower_bound, upper_bound

# Run
mu, lower, upper = monte_carlo_with_ci(10000)

print(f"Estimate: {mu:.4f}")
print(f"95% CI:   [{lower:.4f}, {upper:.4f}]")
```

```
Estimate: 3.1572
95% CI:   [3.1252, 3.1892]
```

3.3 Variance Reduction Techniques

Until now, we have considered situations where sampling the probability space is effective. For instance, in the π calculation example, we are sampling points (x, y) from a well-defined 2-dimensional space. By sampling a large number of points we expect our estimate to converge to the true value as the number of samples tends to infinity. However, what happens when we need to sample from a much higher dimensional space? For instance, sampling 100 numbers in a one-dimensional space might be enough for a good estimation. In a $2D$ space, we would need to sample $100 \times 100 = 10^4$ points to reach a similar coverage, which would be still doable. However, what if our sampling space has 10 dimensions? That would amount to sampling $(100)^{10}$ points! This is called the **curse of dimensionality** and is one of the central problems in practical, high-dimensional problems.

One of the main advantages of MC simulation is that the error depends primarily on the number of samples and the variance of the simulation, and not on the dimension of the problem (see Equation 3.4). The problem is that the square root in the denominator forces us to increase the number of samples quadratically to achieve a specific error level. For example, if a simulation takes 1h to run, making it 10 times more accurate would take not 10, but 100 hours (more than 4 days).

However, there is another knob in Equation 3.4 that we can use: the *standard deviation* σ. If we manage to reduce this standard deviation (or equivalently, the variance) of the simulation, we can increase the accuracy of the simulation as well. We will now see some methods for reducing the variance effectively in MC simulation: antithetic variates, control variates, importance sampling and stratified sampling.

3.3.1 Antithetic variates

Imagine the draw a large value for a random variable X_i during our simulation (large compared to the true mean μ). Automatically, this value will increase the sample variance significantly.

One simple way of compensating for this large value is to generate a "mirror" value that makes the large value cancel out. This is the main idea of the **antithetic variates** method. For instance, if we sample a random variable U in $[0, 1]$, we do include also the value $1 - U$ in our sample. Therefore, instead of sampling n times independently, we sample $n/2$ pairs $(U, 1 - U)$. This typically already reduces the variance by a significant margin, improving the effectiveness of the MC simulation.

3.3.2 Control variates

Another highly effective method for reducing the variance is to use **control variates**. The main intuition behind this method is the following: imagine that you want to weigh an object Y, however the scale has unknown accuracy. Let's say we put it on the scale and it shows 10.5 kg. Now we have another object X for which we know, for sure, that it weighs exactly 10 kg. After putting it on the scale, we get a measurement of 10.2 kg. This means that the *scale is off by 0.2 kg*, so we can now correct our previous measurement and correctly conclude that Y weighs 10.3 kg.

In our case, we want to estimate $E[Y]$ using MC simulation, for which there is no analytical solution. However, there is a control X for which we can calculate $E[X]$ exactly and is highly correlated with Y. Then, we can define a corrected estimator as:

$$Y_{cv} = Y - c(X - E[X]) \tag{3.6}$$

with a specific value for the constant c. Note that this estimator is unbiased:

$$E[Y_{cv}] = E[Y] - c(E[X] - E[X]) = E[Y]$$

We are interested in minimizing the variance of this estimator, which is

$$\mathrm{Var}(Y_{cv}) = \mathrm{Var}(Y) + c^2 \,\mathrm{Var}(X) - 2c \,\mathrm{Cov}(X, Y) \tag{3.7}$$

If we take the value c^* that minimizes this variance, it turns out to be

$$c^* = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Var}(X)}$$

Plugging this value into Equation 3.7 to calculate the ratio between the new and the old variance, we get

$$\frac{\mathrm{Var}(Y_{cv})}{\mathrm{Var}(Y)} = 1 - \rho_{xy}$$

where ρ_{xy} is the correlation coefficient between X and Y. Since we want this ratio to get as close to 0 as possible, we have to choose X and Y so that ρ_{xy} is as close to 1 as possible.

Let's solve an example to illustrate this technique. Assume that we want to estimate the value of the following integral:

$$I = \int_0^1 e^{x^2} dx$$

which is analytically untractable. However, there is an obvious proxy $(1 + x^2)$, which represents the first two terms of the Taylor expansion of e^{x^2} around 0. We can calculate the integral for the control function easily:

$$\int_0^1 (1 + x^2) dx = \left[x + \frac{x^3}{3} \right]_0^1 = 1 + \frac{1}{3} = 1.3333 \ldots$$

We will now estimate both quantities in the same loop to calculate the controlled estimate.

```
import numpy as np
import matplotlib.pyplot as plt

def simulation_control_variates(n_iterations):
    u = np.random.uniform(0, 1, n_iterations)
    y_samples = np.exp(u**2)
    x_samples = 1 + u**2
    expected_x = 4 / 3

    # Calculate Optimal 'c'
    cov_matrix = np.cov(x_samples, y_samples)
    covariance_xy = cov_matrix[0, 1]
    variance_x = cov_matrix[0, 0]

    c_optimal = covariance_xy / variance_x

    # Formula: Y_cv = Y - c * (X - E[X])
    cv_samples = y_samples - c_optimal * (x_samples - expected_x)

    naive_estimate = np.mean(y_samples)
    naive_variance = np.var(y_samples)

    # Control Variates Estimate
    cv_estimate = np.mean(cv_samples)
```

```
    cv_variance = np.var(cv_samples)

    return {
        "naive_est": naive_estimate,
        "cv_est": cv_estimate,
        "naive_var": naive_variance,
        "cv_var": cv_variance,
        "correlation": np.corrcoef(x_samples, y_samples)[0,1],
        "c_optimal": c_optimal
    }
```

Let's now run the code for 10000 iterations and compare the results:

```
N = 10000
results = simulation_control_variates(N)

# The "True" value (calculated via high-precision numerical integration for comparison)
# integral of e^(x^2) from 0 to 1 is approx 1.4626517459...
true_value = 1.4626517459

print(f"--- Results with N={N:,} ---")
print(f"True Value:                {true_value:.6f}")
print(f"Naive MC Estimate:         {results['naive_est']:.6f} (Error: {abs(results['naive_est
print(f"Control Variate Estimate: {results['cv_est']:.6f} (Error: {abs(results['cv_est'] - t

print(f"\n--- Variance Analysis ---")
print(f"Correlation (X, Y):        {results['correlation']:.4f}")
print(f"Naive Variance:            {results['naive_var']:.6f}")
print(f"CV Variance:               {results['cv_var']:.6f}")

reduction_factor = results['naive_var'] / results['cv_var']
print(f"Variance Reduction:        {reduction_factor:.1f}x lower variance")
```

```
--- Results with N=10,000 ---
True Value:                1.462652
Naive MC Estimate:         1.461151 (Error: 0.001500)
Control Variate Estimate: 1.462047 (Error: 0.000605)

--- Variance Analysis ---
Correlation (X, Y):        0.9916
Naive Variance:            0.222303
```

```
CV Variance:              0.003735
Variance Reduction:       59.5x lower variance
```

As can be seen, the variance reduction achieved by the control variates method is in this case around 60x, which means that running 10,000 iterations using the control variates method is *as good as running classical MC with 600,000 iterations.*

3.3.3 Importance sampling

The next method is used in cases where the region of interest in the sampling space is very small, i.e. the events we try to simulate are very rare. Imagine we want to estimate the probability for a market crash. If we simulate the market for 1,000,000 times, we might still miss all crash events (e.g. if the market crash is considered to be a five-sigma event, its probability amounts to about 1 in 3.5 million). Therefore, standard MC would come up with a probability of 0, which is incorrect.

Instead of that, we sample from a probability distribution where market crashes are common. The good news is that now, we will have plenty of points to work with. However, by doing this we introduce a significant bias in the simulation, so our result will not be correct. The trick used by **importance sampling** is to "un-bias" the simulation by assigning a *weight* to each sample which is basically proportional to its *likelihood*. Samples which are more likely according to the original distribution will get a higher weight, whereas rare samples will be assigned a low weight.

More formally, we want to calculate the expectation of a function of a random variable $f(x)$ under a probability distribution characterized by its pdf $p(x)$.

$$E_p[f(x)] = \int f(x)p(x)dx$$

We now introduce a *proposal distribution* $q(x)$, which is biased towards the rare events, by multiplying and dividing by it:

$$\int f(x)p(x)dx = \int f(x)\frac{p(x)}{q(x)}q(x)dx$$

This is equivalent to the expectation of a new function of x under the new distribution q:

$$\int f(x)\frac{p(x)}{q(x)}q(x)dx = E_q\left[f(x)\frac{p(x)}{q(x)}\right]$$

which is the result of *weighting* $f(x)$ by a factor given by the quotient $p(x)/q(x)$. This results in the estimator:

$$\hat{I} = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \cdot \frac{p(x_i)}{q(x_i)} = \frac{1}{n}\sum_{i=1}^{n} w_i f(x_i) \tag{3.8}$$

Note that our samples now come from $q(x)$ instead of $p(x)$ (because we sample from the biased distribution). Let's now consider as an example a five-sigma event under a standard normal distribution governed by the probability $P(X > 5)$, where the true probability is about 2.87×10^{-7}. In the real world, the true probability distribution is $N(0, 1)$. Now we can sample from a biased distribution by moving the mean: $N(5, 1)$. In this distribution, events $X > 5$ happen roughly 50% of the time.

```
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

def importance_sampling_demo(n_samples, threshold=5):
    samples_naive = np.random.normal(0, 1, n_samples)

    hits_naive = samples_naive > threshold
    prob_naive = np.mean(hits_naive)

    var_naive = (prob_naive * (1 - prob_naive)) / n_samples

    shift_mean = threshold
    samples_is = np.random.normal(loc=shift_mean, scale=1, size=n_samples)

    hits_is = samples_is > threshold # This will be True for roughly 50% of samples
    f_x = hits_is.astype(float)

    p_x = stats.norm.pdf(samples_is, loc=0, scale=1)
    q_x = stats.norm.pdf(samples_is, loc=shift_mean, scale=1)

    weights = p_x / q_x

    weighted_values = f_x * weights
    prob_is = np.mean(weighted_values)

    var_is = np.var(weighted_values) / n_samples

    return prob_naive, var_naive, prob_is, var_is
```

In this code, we compare the probability estimates and the variances obtained by standard MC and importance sampling.

```python
N = 10000 # Only 10,000 samples (Small for a rare event!)
target = 5

naive_p, naive_var, is_p, is_var = importance_sampling_demo(N, target)
true_p = 1 - stats.norm.cdf(target) # Analytical solution

print(f"Target: P(X > {target})")
print(f"True Probability: {true_p:.10f}")
print(f"Naive MC Estimate: {naive_p:.10f}")
print(f"Naive Variance:    {naive_var:.10f} (Likely zero if no hits occurred)")
print(f"Imp. Samp Estimate:{is_p:.10f}")
print(f"Imp. Samp Variance:{is_var:.20f}")

# Check the ratio of variance reduction (avoid div by zero)
if naive_var == 0:
    print("Naive method failed completely (0 hits). Importance Sampling is infinitely better
else:
    print(f"Variance Reduction Factor: {naive_var / is_var:.1f}x")
```

```
Target: P(X > 5)
True Probability: 0.0000002867
Naive MC Estimate: 0.0000000000
Naive Variance:    0.0000000000 (Likely zero if no hits occurred)
Imp. Samp Estimate:0.0000002893
Imp. Samp Variance:0.00000000000000004712
Naive method failed completely (0 hits). Importance Sampling is infinitely better here.
```

As can be seen, the estimate obtained by importance sampling is quite close to the true probability. With only 10,000 iterations, standard MC fails to sample a single extreme event, and therefore both the estimate and the variance are 0.

How to choose a good $q(x)$

We have seen that, in general, we should choose $q(x)$ so that it is biased towards regions where the events of interest are more common. Concretely this means that $q(x)$ should have high density where $|f(x)|p(x)$ is also high, since this will significantly steer sampling towards the "important" region and reduce the variance. On the other side, if $q(x)$ is zero in regions where $p(x)$ is not, we might stop sampling in regions that matter. In this case, the answer will definitely be biased. As a rule of thumb, the tails of $q(x)$ need to be heavier than those of $p(x)$, since we do not want $q(x)$ to decrease faster than $p(x)$.

3.3.4 Stratified sampling

Let's assume we want to integrate a monotonic function (that is, a function that either *always* increases or *always* decreases). Think e.g. about $f(x) = e^x$ in $[0, 1]$. If we start sampling random numbers, we might get trapped either in the low or the high region of the function. This might well happen because PRNM might accidentally "cluster" samples in a specific region. Alas, that would be bad, because it would significantly bias our estimate. In this case, we need to make sure that we sample equally well from all regions. This is the main idea of **stratified sampling**.

The mathematical trick now is to divide the sampling space into disjoint **strata**, which represent separate regions that cover the whole space. Depending on the form of the target function, the strata might be different in size. However, if possible, choosing strata of equal size might simplify the calculations.

Let's denote by H the number of strata and p_h the probability that a random sample falls into stratum h. We denote by S the random variable that represents the index of the stratum. Therefore, $p_h = P(S = h)$ and $\sum p_h = 1$. Each stratum has its own mean $\mu_h = E[Y|S = h]$ and variance $\sigma_h^2 = \mathrm{Var}(Y|S = h)$. The total expected value that we are trying to estimate with stratified MC becomes then:

$$\mu = E[Y] = \sum_{h=1}^{H} p_h \mu_h$$

In order to calculate the variance, we use the *law of total variance* to decompose it into:

$$\sigma^2 = \mathrm{Var}(Y) = E[\mathrm{Var}(Y|S)] + \mathrm{Var}(E[Y|S]) \tag{3.9}$$

In this decomposition, the first term on the right-hand side $E[\mathrm{Var}(Y|S)]$ represents the *mean variance* of the strata, or the variation *within strata*. The second term $\mathrm{Var}(E[Y|S])$ represents the *variance* of the mean values of the strata, which can be seen as a measure of the variation *between* strata.

$$\begin{aligned}
\sigma_W^2 &= E[\mathrm{Var}(Y|S)] = \sum_{h=1}^{H} p_h \sigma_h^2 \\
\sigma_B^2 &= \mathrm{Var}(E[Y|S]) = \sum_{h=1}^{H} p_h (\mu_h - \mu)^2 \\
\sigma^2 &= \sigma_W^2 + \sigma_B^2
\end{aligned}$$

Let's now compare the variance of the stratified estimator compared to standard MC. Recall that for the standard MC estimator $\hat{Y}_{MC} = \frac{1}{n} \sum f(x_i)$:

$$\sigma^2_{MC} = \text{Var}(\hat{Y}_{MC}) = \text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} f(x_i)\right) = \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(f(x_i)) =$$
$$= \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2 = \frac{\sigma^2}{n} = \frac{1}{n}(\sigma^2_W + \sigma^2_B)$$

For the stratified MC estimator, we have the sample means of each stratum $\hat{\mu}_h$, and we calculate $\hat{Y}_{SE} = \frac{1}{n}\sum p_h\hat{\mu}_h$. The variance then becomes the sum of the variances, since the covariance between strata is 0:

$$\text{Var}(\hat{Y}_{SE}) = \sum_{h=1}^{H}\text{Var}(p_h\hat{\mu}_h) = \sum_{h=1}^{H} p_h^2\,\text{Var}(\hat{\mu}_h) = \sum_{h=1}^{H} p_h^2\frac{\sigma^2_h}{n_h}$$

If we now choose the number of samples in each stratum to be proportional to the stratum's weight $n_h = p_h \times n$, we have:

$$\sigma^2_{SE} = \text{Var}(\hat{Y}_{SE}) = \sum_{h=1}^{H} p_h^2\frac{\sigma^2_h}{(p_h \cdot n)} = \sum_{h=1}^{H} p_h\frac{\sigma^2_h}{n} = \frac{1}{n}\sum_{h=1}^{H} p_h\sigma^2_h$$

This should look familiar: it corresponds exactly to $\frac{1}{n}$ times the sum of the individual within-strata variances!

$$\sigma^2_{MC} = \frac{1}{n}(\sigma^2_W + \sigma^2_B)$$
$$\sigma^2_{SE} = \frac{1}{n}\sigma^2_W$$

And since $\sigma^2_B \geq 0$, we have our variance reduction $\text{Var}(\hat{Y}_{SE}) \leq \text{Var}(\hat{Y}_{MC})$.

In summary, stratified sampling using proportional strata is always equal to or better than standard sampling because it eliminates the variance caused by the variability in sampling proportions across strata with different means.

Example

As an example, let's integrate $f(x) = e^x$ in the interval $[0, 1]$. For this, we will define H equally-spaced strata in this interval.

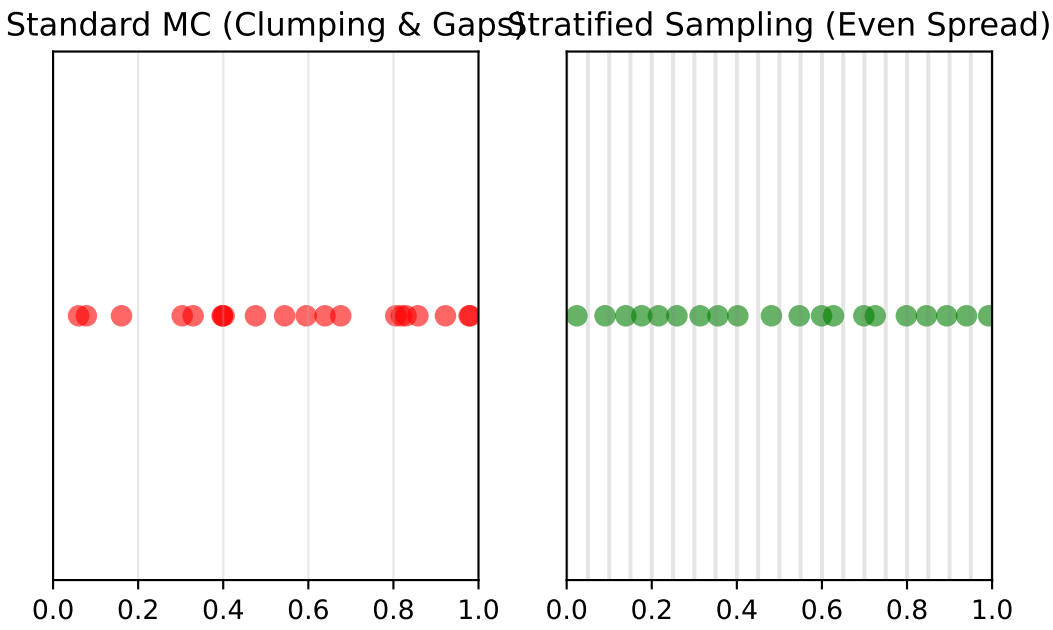
```
import numpy as np
import matplotlib.pyplot as plt

def simulation_stratified(n_samples):
    u_naive = np.random.uniform(0, 1, n_samples)
    y_naive = np.exp(u_naive)

    est_naive = np.mean(y_naive)
    var_naive = np.var(y_naive)

    strata_starts = np.arange(n_samples) / n_samples
    offsets = np.random.uniform(0, 1/n_samples, n_samples)
    x_stratified = strata_starts + offsets

    y_stratified = np.exp(x_stratified)
    est_stratified = np.mean(y_stratified)

    return est_naive, est_stratified
```

We now run the experiment multiple times to determine the variance of the method. Because stratified samples are not iid, we need to repeat the experiment to get an estimate of its variance:

```
experiments = 1000
N = 100 # Samples per experiment

naive_results = []
stratified_results = []

for _ in range(experiments):
    n_res, s_res = simulation_stratified(N)
    naive_results.append(n_res)
    stratified_results.append(s_res)

true_value = np.e - 1

# Calculate statistics of the *Estimators*
var_of_naive_method = np.var(naive_results)
var_of_stratified_method = np.var(stratified_results)

print(f"True Value: {true_value:.6f}")
print("-" * 30)
```

```
print(f"Naive MC Variance (across {experiments} runs):      {var_of_naive_method:.8f}")
print(f"Stratified MC Variance (across {experiments} runs): {var_of_stratified_method:.8f}")
print("-" * 30)
print(f"Variance Reduction Factor: {var_of_naive_method / var_of_stratified_method:.1f}x")

# --- Visualization (First 50 points) ---

# Plot Standard
plt.subplot(1, 2, 1)
u_naive = np.random.uniform(0, 1, 20)
plt.scatter(u_naive, np.zeros_like(u_naive), color='red', alpha=0.6, s=50)
plt.title("Standard MC (Clumping & Gaps)")
plt.yticks([])
plt.xlim(0, 1)
plt.grid(axis='x', alpha=0.3)

# Plot Stratified
plt.subplot(1, 2, 2)
strata_starts = np.arange(20) / 20
offsets = np.random.uniform(0, 1/20, 20)
x_stratified = strata_starts + offsets
plt.scatter(x_stratified, np.zeros_like(x_stratified), color='green', alpha=0.6, s=50)
# Draw grid lines to show strata
for i in range(21):
    plt.axvline(i/20, color='gray', alpha=0.2)
plt.title("Stratified Sampling (Even Spread)")
plt.yticks([])
plt.xlim(0, 1)

plt.tight_layout()
plt.show()
```

```
True Value: 1.718282
------------------------------
Naive MC Variance (across 1000 runs):      0.00229289
Stratified MC Variance (across 1000 runs): 0.00000027
------------------------------
Variance Reduction Factor: 8554.8x
```



As can be seen, the variance could be dramatically reduced and, comparing the plots, we can see why: stratified sampling obtains a much more balance distribution for the sampled points than standard MC, where the random numbers tend to clump and form gaps.

3.3.5 Latin Hypercube Sampling

The last variance reduction technique we will see is a modification of stratified sampling that comes to the rescue in high-dimensional problems: **Latin Hypercube Sampling**. The problem of stratified sampling in higher dimensions is that the number of strata required grows exponentially with the dimension of the problem. For instance, if we wish to stratify each dimension using 100 bins, the number of strata becomes already prohibitive for e.g. 10 dimensions (100^{10} points would be needed).

Latin hypercube sampling is a clever way to organize the sampling space. The main idea is to stratify every single dimension **simultaneously**, but without filling the entire grid. Our goal is to sample n points such that, when looking from the perspective of any single dimension, there is always exactly one point in every bin. To visualize this, imagine a chess board with 8 rooks, and try to place each rook so that *no rook can attack another*.

TODO: visualization

If we now look at each dimension in isolation, there is always exactly one rook in each row/column. The advantage is that samples are now spread out maximally across the range of each variable individually, thus effectively reducing the variance in higher dimensions.

The method to build a latin hypercube for sampling n points is as follows:

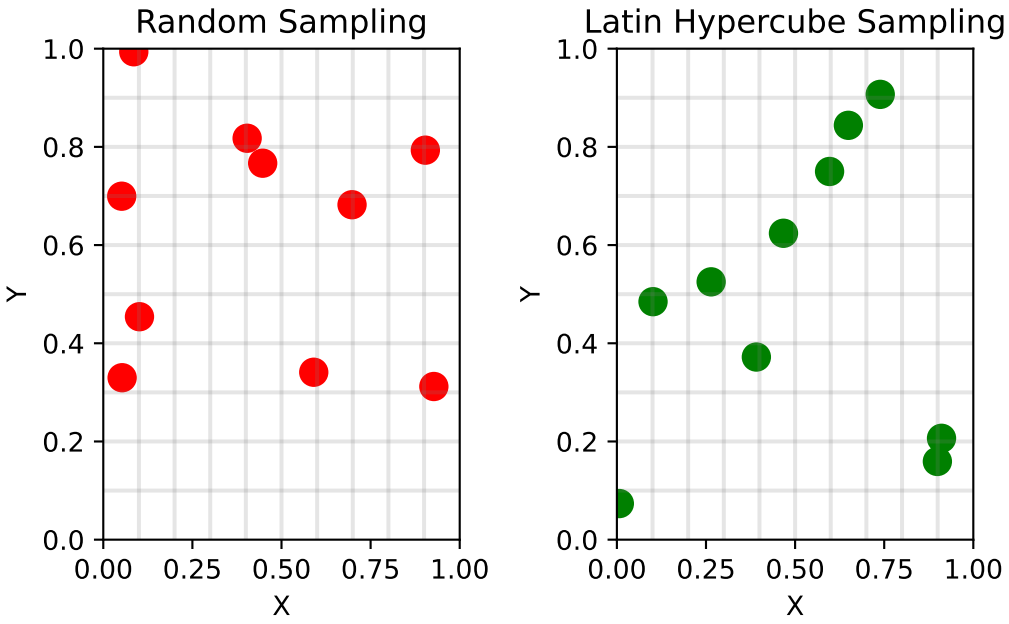
1. Divide the range $[0, 1]$ into n equal intervals.
2. For each dimension, generate a random permutation of the indices $[0, 1, \ldots, N-1]$.
3. Based on these permutations, we assign the intervals to the samples.
4. To avoid each sample to be exactly at the center, we add a random jitter to each point.

```
import numpy as np
import matplotlib.pyplot as plt

def latin_hypercube_sampling(n_samples, n_dim):
    samples = np.zeros((n_samples, n_dim))

    for d in range(n_dim):
        permutation = np.random.permutation(n_samples)

        jitter = np.random.uniform(0, 1, n_samples)

        samples[:, d] = (permutation + jitter) / n_samples

    return samples
```

In two dimensions, this would look like the following:

```
N = 10 # 10 samples
D = 2  # 2 dimensions

# Generate Data
random_samples = np.random.uniform(0, 1, (N, D))
lhs_samples = latin_hypercube_sampling(N, D)

# Plot 1: Random Sampling
plt.subplot(1, 2, 1)
plt.scatter(random_samples[:, 0], random_samples[:, 1], color='red', s=100)
plt.title("Random Sampling")
plt.xlim(0, 1); plt.ylim(0, 1)
# Draw grid to show bins
for i in range(1, N):
    plt.axvline(i/N, color='gray', alpha=0.2)
    plt.axhline(i/N, color='gray', alpha=0.2)
plt.xlabel("X")
plt.ylabel("Y")
```

```
# Plot 2: Latin Hypercube Sampling
plt.subplot(1, 2, 2)
plt.scatter(lhs_samples[:, 0], lhs_samples[:, 1], color='green', s=100)
plt.title("Latin Hypercube Sampling")
plt.xlim(0, 1); plt.ylim(0, 1)
# Draw grid to show bins
for i in range(1, N):
    plt.axvline(i/N, color='gray', alpha=0.2)
    plt.axhline(i/N, color='gray', alpha=0.2)
plt.xlabel("X")
plt.ylabel("Y")

plt.tight_layout()
plt.show()
```



As can be seen, the points are clearly better organized than in the random sampling case. The jitter adds some variability in order not always to fall in the center of each bin.

To conclude, we demonstrate how to calculate a high-dimensional integral using latin hypercube sampling. Let's calculate the integral of the following 5-dimensional function:

$$f(\mathbf{x}) = \sum_{i=1}^{5} x_i^2$$

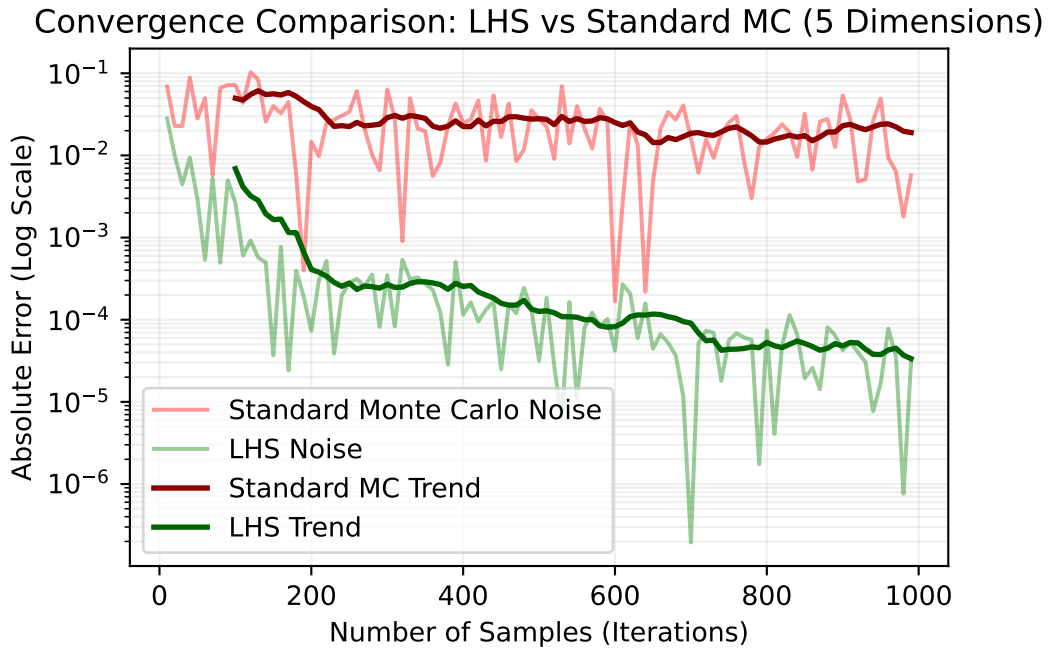
in the interval $[-1, 1]$ for each dimension x_i. For this, we will use the standard latin hypersquare sampling implementation of the SciPy library `scipy.stats.qmc`.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import qmc

def target_function(x):
    return np.sum(x**2, axis=1)

dim = 5
sample_sizes = np.arange(10, 1000, 10)

errors_mc = []
errors_lhs = []
true_mean = dim * (1/3)

for n in sample_sizes:
    mc_samples = np.random.uniform(low=-1, high=1, size=(n, dim))
    mc_results = target_function(mc_samples)
    mc_estimate = np.mean(mc_results)
    errors_mc.append(abs(mc_estimate - true_mean))

    sampler = qmc.LatinHypercube(d=dim)
    lhs_unit = sampler.random(n=n)

    lhs_samples = qmc.scale(lhs_unit, l_bounds=[-1]*dim, u_bounds=[1]*dim)

    lhs_results = target_function(lhs_samples)
    lhs_estimate = np.mean(lhs_results)
    errors_lhs.append(abs(lhs_estimate - true_mean))
```

Note that the `LatinHypercube` sampler outputs samples in the interval $[0, 1]$ which need so be scaled to $[-1, 1]$ using the scaler `qmc.scale`. Let's now visualize the results:

```
plt.semilogy(sample_sizes, errors_mc, color='red', alpha=0.4, label='Standard Monte Carlo No
plt.semilogy(sample_sizes, errors_lhs, color='green', alpha=0.4, label='LHS Noise')

# Add trendlines (moving average) to make it clearer
def moving_average(a, n=10) :
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
```

```
    return ret[n - 1:] / n

plt.semilogy(sample_sizes[9:], moving_average(errors_mc), color='darkred', linewidth=2, label
plt.semilogy(sample_sizes[9:], moving_average(errors_lhs), color='darkgreen', linewidth=2, la

plt.title(f"Convergence Comparison: LHS vs Standard MC ({dim} Dimensions)")
plt.xlabel("Number of Samples (Iterations)")
plt.ylabel("Absolute Error (Log Scale)")
plt.legend()
plt.grid(True, which="both", ls="-", alpha=0.2)
plt.show()

# Final comparison for the largest N
print(f"Final Error (N={sample_sizes[-1]}):")
print(f"Standard MC Error: {errors_mc[-1]:.6f}")
print(f"LHS Error:         {errors_lhs[-1]:.6f}")
print(f"Improvement Factor: {errors_mc[-1]/errors_lhs[-1]:.1f}x more accurate")
```




```
Final Error (N=990):
Standard MC Error: 0.005729
LHS Error:         0.000031
Improvement Factor: 187.1x more accurate
```

In this plot, we have used a logarithmic scale on the y-axis to better see small errors. Additionally, a trend line is provided which is the moving average of the errors. As can be seen, the LHS errors are clearly lower than the standard MC errors and the method is about 100x more accurate than standard MC using the same number of samples.

3.3.6 Summary of Variance Reduction Techniques

Variance reduction techniques are essential in Monte Carlo simulations to improve the accuracy of estimates without requiring a significant increase in the number of samples. Below is a summary of the techniques discussed and when to use each:

1. **Antithetic Variates**:
    - **When to use**: Use when the random variables are symmetric, and you can generate negatively correlated pairs (e.g., sampling U and $1 - U$).
    - **Advantage**: Reduces variance by ensuring that large deviations in one direction are counterbalanced by deviations in the opposite direction.
2. **Control Variates**:
    - **When to use**: Use when you have a control variable that is highly correlated with the target variable and whose expected value is known.
    - **Advantage**: Significantly reduces variance by leveraging the known expected value of the control variable to correct the estimate.
3. **Importance Sampling**:
    - **When to use**: Use when the region of interest in the sampling space is rare or has low probability (e.g., rare events like market crashes).
    - **Advantage**: Focuses sampling on the important regions of the space, improving efficiency and reducing variance for rare events.
4. **Stratified Sampling**:
    - **When to use**: Use when the sampling space can be divided into distinct strata, and you want to ensure proportional representation from each stratum.
    - **Advantage**: Reduces variance by eliminating variability caused by uneven sampling across strata.
5. **Latin Hypercube Sampling (LHS)**:
    - **When to use**: Use in high-dimensional problems where stratified sampling becomes computationally expensive.
    - **Advantage**: Ensures that samples are evenly distributed across each dimension, reducing variance in high-dimensional spaces.

Each technique has its strengths and is suited for specific scenarios. By selecting the appropriate method, a significant variance reduction and improved efficiency of the MC simulation can be achieved.

3.4 Markov Chain Monte Carlo

Until now, all methods for standard Monte Carlo that we have seen so far rely explicitly on the However, in many practical scenarios, generating independent samples directly from the target distribution is either computationally expensive or infeasible. This is where **Markov Chain Monte Carlo (MCMC)** methods come into play. MCMC methods allow us to sample from complex distributions by constructing a Markov chain whose stationary distribution matches the target distribution.

The key idea is to design a Markov chain that "explores" the target distribution efficiently, even if the samples are not independent. Over time, the chain converges to the target distribution, and we can use the samples generated to approximate expectations.

In the next sections, we will explore the foundations of MCMC, including the **Metropolis-Hastings** algorithm and **Gibbs sampling**, two of the most widely used MCMC techniques.

3.4.1 Motivation for MCMC

Imagine that we want to map the mountain peaks on a mountain range that is covered in thick fog. We have basically two options:

1. A helicopter drops thousands of hikers at completely random coordinates in a 100 km area. As a result, 99% of them land in the valleys or the ocean. Only a few luck out and land on a peak. This is the approach used by standard MC.
2. We drop only **one** hiker, which follows the following rule:
    - Pick a random direction and check the altitude of that spot.
    - If the new spot is higher, **move there**.
    - If it's lower, move there only with a given **probability**.

The last approach encompasses the main idea of MCMC: If the hiker only moved up, they would get stuck on the very top of the first tiny hill they found (local maximum). By occasionally accepting downward steps, the hiker can traverse valleys to find the massive mountains on the other side. After 10,000 (simulation) steps, if we look at a map of where the hiker has been, the density of their footprints will perfectly match the elevation map of the mountain range.

Note that it's not just that the mountain pikes are concentrated in a rather small volume: the main issue in this case is rather that it is

challenging to sample from the target distribution directly. MCMC methods provide a way to approximate the target distribution by constructing a **Markov chain** that explores the space iteratively. Over time, the chain converges to the desired distribution, allowing us to estimate expectations and probabilities effectively.

3.4.2 When do we apply MCMC?

As mentioned early, MCMC is particularly useful in scenarios where direct sampling from the target distribution is challenging or computationally expensive. Some common situations where MCMC is applied include:

1. **High-dimensional distributions**: When the target distribution exists in a high-dimensional space, direct sampling becomes infeasible due to the curse of dimensionality. We already mentioned similar situations for some variance reduction methods in Section 3.3.
2. **Complex or unknown normalizing constants**: In Bayesian inference, posterior distributions often involve a normalizing constant that is difficult to compute. That is a distribution of the form:

$$p(x) = \frac{f(x)}{Z}$$

where Z is a normalization constant required to make probabilities sum up to 1. In essence, Z represents the integral of $f(x)$ over the entire domain. However this integral might be impossible to calculate. This is especially relevant in Bayesian inference when calculating the posterior distribution of a parameter of interest given the available data:

$$p(\theta|D) = \frac{p(D|\theta)p(\theta)}{p(D)}$$

where the normalization constant $p(D)$ is usually written as:

$$p(D) = \int p(D|\theta)p(\theta)d\theta$$

This integral is usually intractable, either because of the high dimensionality of θ (think e.g. of neural network weights) or because there is no closed form solution. MCMC allows sampling without explicitly calculating this constant, as we will see shortly.

3. **Non-standard distributions**: When the target distribution does not have a closed-form expression or does not belong to a standard family of distributions.

4. **Rare event probabilities**: When estimating probabilities of rare events, MCMC can efficiently explore the regions of interest.

5. **Integration over complex spaces**: MCMC is used to approximate integrals in cases where the integrand is defined over a complex or irregular domain.

Applications of MCMC include cryptography (deciphering substitution ciphers), detecting gerrymandering in political science, protein folding in computational biology, Bayesian inference in machine learning, and epidemiology.

3.4.3 Foundations of MCMC

The main idea of MCMC is the following: we want to sample the **target distribution** using a Markov chain where the **stationary distibution** is itself the distribution we want to sample from. Let's denote by $\pi(x)$ our target distribution. The Markov chain we want to find is characterized by a **transition kernel** $T(x'|x)$. The transition kernel is a generalization for the transition matrix $\mathbf{P}$ that applies for both continuous and discrete-state Markov chains. In the discrete case, $T(j|i) = P_{ij}$. In the continuous case, $T(x'|x)$ is just the integral of the conditional pdf $f(x'|x)$ over the appropriate domain.

Now we want to build our kernel in such a way that $\pi(x)$ is the stationary distribution of the chain. This is expressed by:

$$\pi(x') = \int T(x'|x)\pi(x)dx \tag{3.10}$$

This means that when drawing a sample x from $\pi(x)$, applying the transition rules of the chain also results in a point x' that is distributed according to π. Therefore, once that the chain enters the target distribution, it will stay there forever. However, working directly with Equation 3.10 can be challenging, therefore in practice we work with a simpler equilibrium condition (which is stronger than mere stationarity):

$$T(x|x')\pi(x') = T(x'|x)\pi(x) \tag{3.11}$$

This means that the probability flow between any two points x, x' should remain equal. In summary, when this condition holds, then this implies Equation 3.10 and π becomes our stationary distribution as desired. However, we still haven't any guarantees that the chain, indeed, will converge to π. As soon as it does, it will stay there forever, but will it reach that point of no return? To answer this question affirmatively, we need to pose another condition and that is **ergodicity**. In essence, this boils down to the following two critical requirements:

1. The chain is **irreducible**: there must be a non-zero probability of reaching a state x from any other state x'. This means that the chain cannot get trapped forever in an isolated environment.
2. The chain is **aperiodic**: it must not get stuck in a cycle (i.e. a fixed loop).

If these two requirements are met, the following theorem (**Ergodic Theorem**) applies: Let $\{X_t\}$ be a Markov chain that is irreducible, aperiodic and has a stationary distribution π. Then, for any starting point X_0, we have:

$$\lim_{T\to\infty} \frac{1}{T}\sum_{t=1}^{T} g(X_t) = E_\pi\left[g(X)\right] \tag{3.12}$$

That is, in the limit the time average of the chain equals the mean of the target distribution.

3.4.4 Metropolis-Hastings

We now focus on a specific algorithm for building a transition kernel T for a Markov chain that fulfills the equilibrium condition Equation 3.11 and the conditions for ergodicity. For this, a seminal approach is the **Metropolis-Hastings** algorithm. The main idea is to decompose the kernel $T(x'|x)$ into two parts:

1. A **proposal** distribution $q(x'|x)$, which is usually a distribution we know how to sample from (e.g. usually a Gaussian centered at the current state $N(x, \sigma)$).
2. An **acceptance probability** $\alpha(x'|x)$ that expresses the probability that we accept the proposed move.

So we have:

$$T(x'|x) = \alpha(x'|x) \cdot q(x'|x)$$

Let's substitute this into Equation 3.11:

$$\alpha(x|x')q(x|x')\pi(x') = \alpha(x'|x)q(x'|x)\pi(x) \tag{3.13}$$

Metropolis-Hastings proposes the following rule to satisfy the previous condition:

$$\alpha(x'|x) = \min\left(1, \frac{q(x|x')\pi(x')}{q(x'|x)\pi(x)}\right) \tag{3.14}$$

Because, in principle, this ratio is unbounded, we need to set a ceiling of 1 so that we get a valid probability. One of the main advantages of Equation 3.14 is that, since π appears both on

the numerator and the denominator, any normalization constant Z appearing as $\pi(x) = f(x)/Z$ cancels out, so we don't need to estimate this constant anymore.

The algorithm itself involves the following steps:

1. Pick an arbitrary strating point x_0.
2. For $t = 0$ to T (simulation length):
   - Sample a candidate x' from the proposal distribution $x' \sim q(x'|x_t)$.
   - Calculate the acceptance probability α according to Equation 3.14.
   - Generate a uniform random number u in the interval $[0, 1]$.
   - If $u \leq \alpha$, **accept**: $x_{t+1} = x'$.
   - Otherwise, **reject**: $x_{t+1} = x_t$.

Why Metropolis-Hastings works

Let's take a closer look at Equation 3.14, and why it satisfies Equation 3.11. We can divide the ratio into the following parts:

$$\alpha = \min\left(1, \underbrace{\frac{\pi(x')}{\pi(x)}}_{\text{Likelihood Ratio}} \times \underbrace{\frac{q(x|x')}{q(x'|x)}}_{\text{Correction Factor}}\right)$$

The likelihood ratio $\pi(x')/\pi(x)$ measures how much better the new sampled point x' is in relation to the current one x. When this ratio is greater than 1, this means that the new point has a higher probability density than the previous one, and we would normally accept.

The correction factor $q(x|x')/q(x'|x)$ ensures that, in case the proposal is not symmetric, the simulation is not biased towards specific high-probability regions by correcting for the imbalance (i.e. $q(x'|x)$ is very large).

Now let's denote the combined ratio by r, so $\alpha = \min(1, r)$ and focus on the detailed balance condition (Equation 3.13), which states the the flow from $x \to x'$ should be equal than the flow $x' \to x$.

- If $r \geq 1$, we get $\alpha(x'|x) = 1$. Additionally, $\alpha(x|x')$ is $1/r$, which amounts to $\pi(x)q(x'|x)/\pi(x')q(x|x')$. Substituting in Equation 3.13 we get:

$$\frac{\pi(x)q(x'|x)}{\pi(x')q(x|x')} \cdot q(x|x')\pi(x') = 1 \cdot q(x'|x)\pi(x)$$

which is clearly satisfied.

- If $r < 1$, we have that the minimum is less than 1, and therefore $\alpha(x'|x) = \pi(x')q(x|x')/\pi(x)q(x'|x)$. For $\alpha(x|x')$, we have $1/r > 1$ and therefore $\alpha(x|x') = 1$. Substituting in Equation 3.13 we get:

$$1 \cdot q(x|x')\pi(x') = \frac{\pi(x')q(x|x')}{\pi(x)q(x'|x)} \cdot q(x'|x)\pi(x)$$

which is also satisfied.

Example

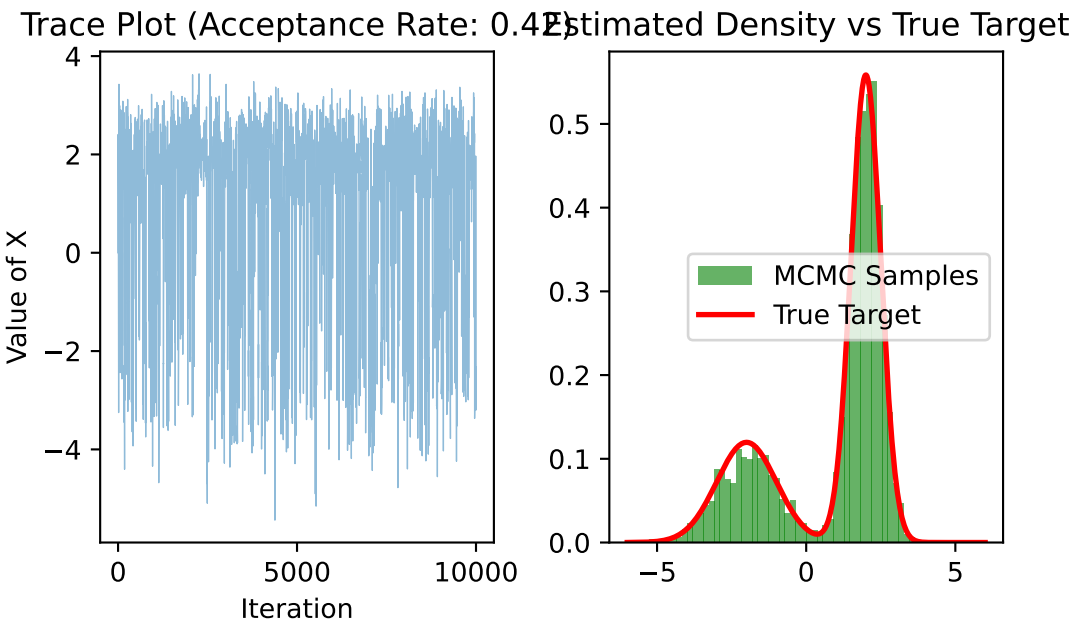
Let's sample from a bimodal distribution which is a mixture of two Gaussians. Without MCMC, classical algorithms would frequently get stuck in one peak or the other. For instance, $\pi(x) = 0.3N(-2, 1) + 0.7N(2, 1)$.

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as stats

def target_pdf(x):
    return 0.3 * stats.norm.pdf(x, loc=-2, scale=1) + \
           0.7 * stats.norm.pdf(x, loc=2, scale=0.5)

def proposal_pdf(x, current_x, step_size):
    return stats.norm.pdf(x, loc=current_x, scale=step_size)

def propose(current_x, step_size):
    return np.random.normal(current_x, step_size)

def metropolis_hastings(n_samples, initial_x, step_size):
    samples = [initial_x]
    current_x = initial_x
    accepted_count = 0

    for _ in range(n_samples):
        proposed_x = propose(current_x, step_size)

        p_current = target_pdf(current_x)
        p_proposed = target_pdf(proposed_x)

        correction = proposal_pdf(current_x, proposed_x, step_size) / \
            proposal_pdf(proposed_x, current_x, step_size)

        if p_current == 0:
            ratio = 1
        else:
            ratio = (p_proposed / p_current) * correction
```

```
        alpha = min(1, ratio)

        u = np.random.uniform(0, 1)
        if u <= alpha:
            current_x = proposed_x
            accepted_count += 1
        else:
            pass

        samples.append(current_x)

    return np.array(samples), accepted_count / n_samples
```

Now let's run a simulation for 10,000 steps.

```
N = 10000
start_x = 0
step_size = 2.0

samples, acc_rate = metropolis_hastings(N, start_x, step_size)

plt.subplot(1, 2, 1)
plt.plot(samples, alpha=0.5, lw=0.5)
plt.title(f"Trace Plot (Acceptance Rate: {acc_rate:.2f})")
plt.xlabel("Iteration")
plt.ylabel("Value of X")

# Histogram vs True Density
plt.subplot(1, 2, 2)
# Histogram of MCMC samples
plt.hist(samples, bins=50, density=True, alpha=0.6, color='g', label='MCMC Samples')
# True curve
x_range = np.linspace(-6, 6, 1000)
plt.plot(x_range, target_pdf(x_range), 'r-', lw=2, label='True Target')
plt.title("Estimated Density vs True Target")
plt.legend()

plt.tight_layout()
plt.show()
```



3.5 Chapter Summary

In this chapter, we have explored Monte Carlo simulation in detail, including its rationale and several variance reduction techniques to improve simulation accuracy. These techniques include antithetic variates, importance sampling, stratified sampling, and latin hypercube sampling. Additionally, we explored Markov Chain Monte Carlo as a way of simulating specially challenging probability distributions, and took a closer look at the Metropolis-Hastings algorithm as one special way of implementing MCMC.

3.6 Exercises

1. Prove that for the control variates variance reduction approach, the constant that minimizes the variance is indeed $c^* = \text{Cov}(X, Y)/\text{Var(X)}$. Hint: Use calculus.

2. Consider the importance sampling variance reduction technique and let $f(x) > 0$ be strictly positive. Let the proposal distribution be defined as $q^*(x) = f(x)p(x)/Z$, where Z is the normalization constant required to make q^* a probability distribution. Note that $Z = \int f(x)p(x)dx$ is exactly the integral we are trying to estimate. Substitute this proposal distribution q^* into Equation 3.8 and explicitly calculate its variance. Explain why this choice of q^* is theoretically optimal.

3. Write a Python script that compares Standard Monte Carlo with stratified sampling for estimating the integral $I = \int_0^1 \frac{1}{1+x} dx$, whose analytical solution is $\ln(2) \approx 0.693147$. For stratified sampling, use 10 equally sized strata and print the "Variance Reduction Factor" (Variance of Standard / Variance of Stratified). Do the results align with the theoretical expected result?

4. Implement the Metropolis-Hastings method for sampling from a distribution proportional to the Gamma(3,1) distribution $\pi(x) \propto x^2 e^{-x}$ for $x > 0$ and $\pi(x) = 0$ for $x \leq 0$. Plot a histogram of your samples against the theoretical curve (normalized) to visually verify convergence and calculate the empirical mean.

4 Discrete events and Queuing Theory

4.1 Introduction

In the previous chapters, we have considered mostly continuous simulation approaches. These approaches are well suited for situations where systems can be described using simple equations. However, there are situations that are either too complex to be modeled in such a way, or too expensive or dangerous to experiment with in the real world. Whether it is a hospital emergency room attempting to reduce patient wait times, a manufacturing plant trying to optimize throughput, or a computer network managing data packets, these systems share a common trait: they change state at specific, irregular points in time.

Unlike other modeling approaches that view the world as a continuous flow or a static snapshot, **Discrete Event Simulation (DES)** views the world as a sequence of distinct events—arrivals, departures, breakdowns, and completions—that occur at specific instants, triggering changes in the system's state. DES becomes necessary when a system exhibits the following properties:

- The state of the system **evolves over time**. A decision made at time t affects the resources available at time $t + 1$.
- Entities (like customers, parts, data) compete for **limited resources** (counters, machines, bandwidth) leading to the formation of *queues*.
- The system is driven by **randomness**. Customers do not arrive at fixed intervals; machines do not take the exact same amount of time to process every part.

We utilize DES when analytical solutions are mathematically intractable. For example, calculating the average wait time in a bank with one line and one teller is a simple mathematical exercise. However, if that bank has varying arrival rates depending on the time of day, tellers who take lunch breaks, and customers who may leave (balk) if the line is too long, the mathematical formulas break down. DES allows us to mimic this complex reality inside a computer to answer "What if?" questions without disrupting actual operations.

Note that Monte Carlo simulation can be considered in general as **static**. It evaluates a problem at a single point in time or across a timeless dimension. For instance, estimating the value of π by dropping distinct pins on a grid is a Monte Carlo experiment. It provides a probabilistic outcome of a specific scenario but does not model how that scenario evolves. By contrast, DES is **dynamic**, creating a history of the system operation over a duration.

The mathematical foundation that represents the basis for DES is called **Queuing Theory**. It underpins the computational engine used by DES to solve complex simulation problems. Queuing Theory provides the analytical framework for understanding waiting lines. It offers exact mathematical formulas to predict performance measures such as server utilization or and average waiting times under specific, idealized assumptions (e.g., purely random arrivals and service times).

In this chapter, we will introduce the fundamentals of Queuing Theory that will serve as a framework to understand the principles of DES. While Queuing Theory provides the "laws of physics" for the system at hand, DES provides the laboratory in which we experiment with those laws under complex, realistic conditions.

4.2 Queuing Theory

Before we can build a valid discrete simulation model, we must understand the mathematical logic that governs the flow of entities. Queuing Theory is the study of waiting lines. It provides a set of analytical techniques used to describe the behavior of systems where entities arrive, wait for a resource, receive service, and depart. While we often associate queues with negative experiences—standing in line at a grocery store or waiting on hold for customer support—queuing theory views the queue not as a nuisance, but as a buffer. It is a necessary component that manages the mismatch between the irregular demand for service and the limited capacity to provide it. The fundamental goal of queuing theory is to quantify the trade-off between the cost of providing service (adding more servers) and the cost of waiting (lost time or unhappy customers).

4.2.1 Anatomy of a Queuing System

We now decompose a queueing system into its basic constituent processes: the arrival, the queue and the service mechanism.

- The **arrival process** defines how entities appear. In most queuing models, we assume that arrivals are random and measure them using the *interarrival time*, which is the time elapsed between two consecutive arrivals.
- The **queuing mechanism** defines how the holding area looks like where entities wait for service. This queue might have finite or infinite capacity, and function according to a specific principle or set of rules. The most common is a First-In, First-Out (FIFO) mechanism, where the first entity that arrived at the queue will get service first, and all other entities will advance by one place on the waiting line. However, other (less frequent) mechanisms might be used like LIFO (Last-In, Fist-Out), SIRO (Service In Random Order) or some other Priority schemes.

- The **service mechanism** defines how entities are processed, e.g. in a serial way (only one server) or with several concurrent servers. Additionally, this mechanism is characterized by a *service time*, which is usually stochastic as well.

Revisiting the Supermarket

Coming back to the supermarket example of Chapter 1, we can now identify the system's components as follows:

- **Arrival:** Customers arrive randomly following exponentially distributed interarrival times.
- **Queueing mechanism:** Customers are put into an infinite waiting room (the queue has infinite capacity) and they are served sequentially, using a FIFO mechanism.
- **Service:** Once that the customers arrive at checkout, they are serviced by the next available of c cashiers. Service times are exponentially distributed.

In general, Queuing Theory provides a mathematical framework for ideal systems which can be handled analytically. However, this is not always the case. For these cases, DES offers a mechanistic approach to perform realistic system simulations. Nevertheless, Queuing Theory can also be used in this cases to provide a calibration benchmark or a "sanity check" for a simulation study by simplifying the model and checking that the results of the DES are the same than the ones predicted by the theory.

4.2.2 Kendall's Notation

Queuing systems can vary in some specific ways: for instance, different arrival patterns, number of servers, etc. In order to describe these different modalities, we will use a formal notation called **Kendall's Notation**. This notation is based on three components in the format $A/B/c$. Each of these components represent the following characteristics:

- A**:** Denotes the probability distribution of the *arrival* times.
- B**:** Denotes the probability distribution of the *service* times.
- c**:** Denotes the number of parallel channels available (e.g. number of servers).

Some symbols for common distributions used in practice for A and B include:

- M**:** The *memoryless* (or Markovian) distribution. This implies for times the use of the exponential distribution, and for counts the Poisson distribution.
- D**:** Denotes a deterministic mechanism where times are constant (zero variance). For instance, the service time in an automated car wash could take always the same 5 minutes per car.
- G**:** Denotes any general distribution like Gaussian, Gamma, etc.

Some examples include the $M/M/1$ model (exponential arrivals, exponential service times, and 1 server), $M/M/c$ (same as before but with c servers in parallel) and $M/D/1$ (exponential arrivals, fixed service times, and 1 server).

4.2.3 Key Performance Measures

We now describe some measures used to quantify how well the system performs. These metrics measure, in general, two competing objectives: the efficiency of the resource and the waiting time of the customer.

Let our input parameters be λ (the mean arrival rate, e.g. 10 customers per hour) and μ (the mean service rate per server, e.g. a bank teller handling on average 15 customers per hour). In the following, we will use the term *queue* to refer to the waiting line itself, whereas *queueing system* refers to the whole system (the queue plus the service).

Utilization factor

We denote the fraction of time each server is busy, or utilization factor by ρ. It measures how efficiently the servers are being used:

$$\rho = \frac{\lambda}{c\mu}$$

Note that if the arrival rate exceeds the service rate, the queue size diverges to infinity (i.e. the condition $\rho < 1$ is a sufficient condition for stability).

State metrics

We now assume that the system is in a steady state, i.e. the system has been running for long enough so that the initial conditions no longer influence current state probabilities. The state of the system is modeled as a random variable. Let P_n denote the steady state probability that there are exactly n entities in the system. We seek to minimize the following state metrics for any queuing system:

- L: The length of the system, or average total number of waiting entities and those being served. This is the expected value of the number of entities:

$$L = \sum_{n=0}^{\infty} n \cdot P_n$$

- L_q: The length of the queue, or average number of entities in the waiting line. We define this quantity as the expected value of entities waiting in the line, excluding the ones being served:

$$L_q = \sum_{n=c+1}^{\infty} (n-c) \cdot P_n$$

- W: The total time in the system, or the average time an entity spends in the system (waiting + served). This is the sum of the expected time waiting in the queue plus the expected service time.

$$W = E[\text{Waiting Time}] + E[\text{Service Time}]$$

- W_q: The waiting time in queue, or average time an entity spends in the waiting line before being served. This can be expressed by:

$$W_q = W - \frac{1}{\mu}$$

or the time waiting until one of the servers becomes free.

Little's Law

One of the best known results in queuing theory is known as **Little's law**, which established a fundamental relationship between the number of items in the system L, the arrival rate λ and the average time spent in the system W:

$$L = \lambda \times W \tag{4.1}$$

This relation holds true regardless of the probability distributions involved. Figure 4.1 shows a geometric visualization of Little's Law.

In this plot, we represent a realization of a $M/M/1$ process where the cumulative arrivals are denoted by $A(t)$ and the cumulative departures by $D(t)$. From the horizontal perspective (the customer's view), each horizontal section represents the arrival and departure of an entity (e.g. in the plot the green line represents the total time W_5 that Entity 5 spent in the system). If we sum for all N customers, and we denote by W the average total time per customer, we get an estimation of the total system time (the grey area) of $\approx N \times W$.

From the vertical perspective, each vertical section represents a moment in time, with its length equal to the number of entities in the system. If we now sum for all times, we also get an estimation of the total system time of $\approx T \times L$, where T is the duration of the simulation and L is the average number of entities in the system. Equating both, we get:

$$N \times W = T \times L$$

and therefore

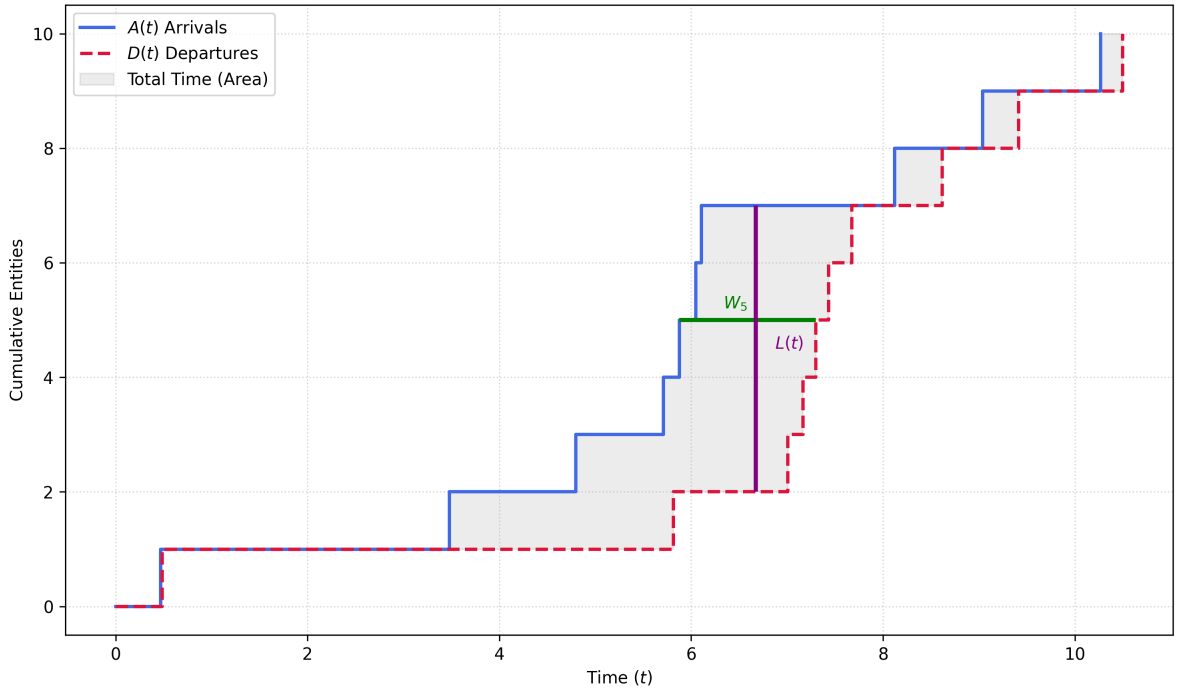


Figure 4.1: Geometric Visualization of Little's Law

$$L = \frac{N}{T} \times W$$

where N/T is exactly the arrival rate λ, and we thus get Equation 4.1.

4.2.4 Example: The M/M/1 Queue

As an example, we now calculate L, L_q, W and W_q explicitly for the case of an $M/M/1$ queue. For the arrivals, we assume a Poisson process with rate λ, exponential service times with rate μ and 1 server. This results in an utilization of $\rho = \lambda/\mu$ and we assume $\rho < 1$ so that the queue remains stable.

The steady state probability P_n becomes then

$$P_n = (1 - \rho)\rho^n$$

where $(1 - \rho)$ is the probability that the system is empty. The length of the system L becomes then:

$$L = \sum_{n=0}^{\infty} n \cdot P_n = \sum_{n=0}^{\infty} n \cdot (1-\rho)\rho^n = (1-\rho) \sum_{n=0}^{\infty} n\rho^n$$

because this is a geometric summation, we get

$$L = (1-\rho)\frac{\rho}{(1-\rho)^2} = \frac{\rho}{(1-\rho)} = \frac{\lambda}{\mu - \lambda}$$

Using Little's Law, we can now easily find W:

$$W = \frac{L}{\lambda} = \frac{1}{\mu - \lambda}$$

Similarly, the total time in the queue W_q can be calculated as follows:

$$W_q = W - \frac{1}{\mu} = \frac{1}{\mu - \lambda} - \frac{1}{\mu} = \frac{\lambda}{\mu(\mu - \lambda)}$$

Applying Little's Law for the queue ($L_q = \lambda W_q$), we get:

$$L_q = \frac{\lambda^2}{\mu(\mu - \lambda)}$$

4.2.5 Summary

We have now seen the basic definitions behind Queuing Theory and the main performance measures that we seek to optimize. Depending on the actual form of the system ($M/M/1$, $M/M/c$, etc.) each measure takes a specific analytical form that can be calculated explicitly. However, with increasing complexity it becomes more difficult to get exact analytical expressions for these quantities. For these scenarios, we use DES as a simulation tool as we will explore in the next sections.

4.3 Discrete Event Simulation

While Queuing Theory provides the laws of physics for waiting lines, it has a significant limitation: it requires the world to be well-behaved. The moment we introduce real-world messiness—dynamic arrival rates (e.g., the lunch rush), complex routing logic (e.g., a part returning to a machine for rework), or resource dependencies (e.g., a machine needs both an operator and a tool)—the analytical formulas of Queuing Theory become mathematically unsolvable.

When the assumptions of Queuing Theory aren't valid anymore, we turn to **Discrete Event Simulation (DES)**. Discrete Event Simulation is a computational technique used to model the operation of a system as a discrete sequence of events in time. The defining characteristic of DES is how it handles state changes. In a continuous system (like water flowing into a tank), the state changes continuously every millisecond. In a discrete system, the state is piecewise constant. The number of people in a line remains exactly the same for a period of time, then changes instantaneously when a person arrives or departs. Therefore, a DES model assumes that nothing interesting happens between events. This assumption allows the computer to skip over periods of inactivity, making the simulation highly efficient.

4.3.1 Components of a DES Model

In order to define a DES model, we need to map the physical elements of the system to simulation components.

1. **Entities:** The dynamic objects that flow through the system (e.g. customers of a bank, data packets in a computer network, etc.). Entities arrive, move through the system, queue for resources, and leave the system (depart).
2. **Attributes:** Local data tags that describe the unique characteristics of a specific entity (e.g. the arrival time of an entity, the severity of a patient, etc). With attributes, we can treat in the simulation different entitites differently.
3. **Resources:** Stationary elements that provide service to entities and have limited capacity (e.g. tellers, machines, doctos, etc). Resources have normally at least two states: idle (free) and busy (occupied).
4. **Global variables:** Data tags that belong to the whole system instead of individual entities. They normally track the aggregated state of the system (e.g. the current length of the queue, number of busy servers, etc).
5. **Events:** An instantaneous occurrence that changes the state of the system. Events like arrivals and departures are considered as primary events, whereas other types of events (breakdowns, shift changes, etc) are considered as secondary events.

4.3.2 The DES Worldview

In a DES simulation, time is not considered to flow like a clock does. Between events, nothing interesting (from the point of view of the simulation) happens, so the next "tick" of the clock actually happens with the next event. In order to be able to manage these discrete jumps, it is usual for DES to maintain a **Future Event List (FEL)**. Intuitively, this is a sort of "To-Do List" where all known future events are sorted chronologically.

The general simulation loop then becomes as follows:

1. Check the FEL to find the event with the smalles time stamp.
2. Advance the simulation clock to that time.
3. Execute the logic associated with that event (e.g., free the server, update statistics, etc).
4. Generate possible new events and place them into the FEL.
5. Repeat until simulation end is reached.

4.3.3 Building a Simulation Study

A widely held misconception about DES is that "simulation" is synonymous with "coding." In reality, writing the code is often the shortest phase of a project. A simulation study is a structured systems engineering process. If the conceptualization is flawed or the data is inaccurate, the most sophisticated code in the world will yield useless results.

To ensure reliability, a simulation study generally follows a standard five-step life cycle.

1. **Problem formulation:** Before we write a single line of code, we must define the objectives and scope of the simulation. What specific question are we trying to answer? (e.g. if we add a new machine, will that bring the total waiting time to under 5 seconds?). What is inside the model, and what is outside? (e.g. do we need to model that materials need some time for disposal, or is this done instantly?).
2. **Data collection:** This is typically the most time-consuming and difficult phase, where the goal is to gather real-world data to drive the model. This involves e.g. stopwatch studies, analyzing historical logs, and interviewing subject matter experts.
3. **Model translation** This is the phase where the conceptual model is translated into a computer representation (the "coding" phase). This involves defining the entities, resources, and event logic within the chosen simulation software (e.g., SimPy, Arena, AnyLogic, or C++).
4. **Verification and validation:** This is the quality control phase. Verification answers the question whether the code does what is intended to do. Comparing the model against Queuing Theory formulas is a primary method of verification. On the other hand, validation is concerned with accuracy, i.e. does the model behave like the real system?

5. **Experimentation and analysis:** Once the model is valid, it becomes a virtual laboratory. We run scenarios (e.g., "Increase arrival rate by 20%") and analyze the resulting output statistics to make specific recommendations.

The Markovian Assumption

In Section 4.2, we relied heavily on the Exponential Distribution (Markovian) because it makes the mathematics solvable. However, in a simulation study, we are not bound by analytical convenience. Instead, we should and must use distributions that closer match reality.

Specifically, while arrivals do mostly follow a Poisson process (exponential interarrivals), scheduled arrivals do not. Regarding service times, humans are rarely exponential. A highly skilled worker might be consistent with low variance (Normal distribution), while machine repair times often follow a Weibull or Lognormal distribution. In this case, the main recommendation is to fit historical data to a probability curve. If we simply guess the distribution (e.g., assuming a machine always takes exactly 5 minutes when it actually varies between 2 and 12), the simulation results will underestimate the congestion caused by variance.

4.3.4 Analyzing the Output

Analyzing the output of a simulation is fundamentally different from reading a single number. Because DES is driven by random numbers, the output is also a random variable. Therfore, we need to express the results in the language of statistics and probability theory. Specifically, we need to use **confidence intervals**. E.g., instead of saying "The wait time is 10 minutes", the simulation result should read something like: "We are 95% confident that the true average wait time lies between 9.2 and 10.8 minutes." This requires running multiple replications (iterations) of the model to generate a statistically significant sample size, and using standard confidence interval tools to calculate the intervals.

It is also important to define a **Warm-Up Period**. In a steady-state simulation, the model usually starts "empty and idle." This creates of course an initial bias. If we simulate a busy factory but start with zero parts in the system, the first few simulated hours will show misleadingly low queue times while the factory fills up. To correct this, the Warm-up Period (or transient period) is used to run the simulation for a specific time (e.g., 24 hours) to let the queues build up, delete the statistics from this period, and only collect data once the system reaches a stable state.

4.4 Comparative Study

To consolidate our understanding, we will now perform a comparative study. We will define a simple queuing system, solve it using the mathematical formulas derived in Section 4.2, and then build a Discrete Event Simulation (DES) in Python to solve it numerically.

Consider a small 24-hour IT Help Desk with a single technician. Support tickets arrive randomly following a Poisson process with a mean rate of 5 tickets per hour ($\lambda = 5$). The technician resolves tickets with service times that are exponentially distributed with a mean rate of $\mu = 6$ tickets per hour. The queuing discipline is FIFO. Under these conditions, the system can be well described by a $M/M/1$ queuing model.

Our goal is to determine the average waiting time in the queue (W_q) and the average number of tickets in the system (L). Using the formulas derived in Section 4.2, we get:

$$\rho = \frac{5}{6} = 0.8333\ldots$$

So the technician is busy on average 83.3% of the time.

$$L = \frac{\rho}{1-\rho} = \frac{0.8333}{1-0.8333} = 5$$
$$W_q = \frac{\lambda}{\mu(\mu-\lambda)} = \frac{5}{6(6-5)} = \frac{5}{6}$$

The average number of tickets in the system is 5, and the average waiting time is 5/6 hours or 50 minutes. Let's now build an equivalent DES system to replicate these results computationally. We start by defining the system parameters and some helpers:

```
import random
import heapq

# --- Configuration ---
RANDOM_SEED = 42
TOTAL_TIME = 20000.0  # minutes
ARRIVAL_RATE = 5.0 / 60.0  # lambda (customers per minute)
SERVICE_RATE = 6.0 / 60.0  # mu (customers per minute)
WARMUP_TIME = 100.0   # Time to discard (Transient State)

clock = 0.0
queue = []  # Stores arrival times of waiting customers
server_busy = False
fel = []  # Future Event List

total_wait_time = 0.0
total_customers_served = 0
area_q_t = 0.0  # Area under Queue Length curve (for calculating Lq)
last_event_time = 0.0
warmup_cleared = False # Flag to track if we have reset the stats
```

```
def schedule_event(time, event_type):
    """Adds an event to the FEL, sorted by time."""
    heapq.heappush(fel, (time, event_type))
```

The main simulation loop can be implemented as follows:

```
def run_simulation():
    global clock, server_busy, total_customers_served, area_q_t, last_event_time, warmup_clea

    random.seed(RANDOM_SEED)
    interarrival = random.expovariate(ARRIVAL_RATE)
    schedule_event(clock + interarrival, 'ARRIVAL')

    while clock < TOTAL_TIME and fel:
        event_time, event_type = heapq.heappop(fel)
        time_delta = event_time - last_event_time

        queue_len = len(queue)
        if server_busy:
            queue_len += 1 # System Length = Queue + Service

        area_q_t += queue_len * time_delta
        clock = event_time
        last_event_time = clock

        if not warmup_cleared and clock > WARMUP_TIME:
            area_q_t = 0.0
            total_customers_served = 0
            warmup_cleared = True

        if event_type == 'ARRIVAL':
            next_arrival = clock + random.expovariate(ARRIVAL_RATE)
            schedule_event(next_arrival, 'ARRIVAL')

            if not server_busy:
                server_busy = True
                service_time = random.expovariate(SERVICE_RATE)
                schedule_event(clock + service_time, 'DEPARTURE')
            else:
                queue.append(clock)
```

```
        elif event_type == 'DEPARTURE':
            if warmup_cleared:
                total_customers_served += 1

            if len(queue) > 0:
                _ = queue.pop(0) # Entity enters service
                service_time = random.expovariate(SERVICE_RATE)
                schedule_event(clock + service_time, 'DEPARTURE')
            else:
                server_busy = False

    collection_duration = clock - WARMUP_TIME
    L_sim = area_q_t / collection_duration
    W_sim = L_sim / ARRIVAL_RATE
    Wq_sim = W_sim - (1 / SERVICE_RATE)

    print(f"--- Simulation Results (Steady State Only) ---")
    print(f"Total Time: {TOTAL_TIME} | Warm-up: {WARMUP_TIME}")
    print(f"Valid Data Duration: {collection_duration:.1f} min")
    print(f"Total Customers Served (in valid period): {total_customers_served}")
    print(f"-" * 30)
    print(f"Simulated L (Avg in System): {L_sim:.4f} tickets")
    print(f"Simulated Wq (Avg Wait in Queue): {Wq_sim:.4f} min")
```

The FEL is implemented using the `fel` variable. We start by scheduling the first arrival and entering the main loop. We then extract the first element from the FEL. If the server happens to be busy, we queue the element and schedule the next arrival. Otherwise, we serve the element and schedule its departure from the system. The departure event schedules the next departure if there are still elements waiting in the queue, otherwise it sets the state variable `server_busy` again to `False`. Note that we are calculating `L_sim` (the simulated length of the system) by just calculating the area under the curve that describes the current length of the system and dividing by the total time to get the average. The other quantities are given by Little's Law.

We now run the simulation

```
run_simulation()
```

```
--- Simulation Results (Steady State Only) ---
Total Time: 20000.0 | Warm-up: 100.0
Valid Data Duration: 19927.2 min
Total Customers Served (in valid period): 1666
------------------------------
```

```
Simulated L (Avg in System): 5.5395 tickets
Simulated Wq (Avg Wait in Queue): 56.4742 min
```

As can be seen, the result are close to the theoretical 5 tickets and 50 minutes. The more steps we simulate, the closer will the system get to the true analytical results. Note that we are discarding the events that happened before `WARMUP_TIME` so that the system has enough time to leave the initial, empty state and reach a steady state.

However, what is then the point of DES, if we can calculate everything analytically? The reason is that, in practice, very often we can't. For instance, imagine that additionally we have the rule that the technician takes a 10-minute break every 2 hours, but only if no high-priority tickets are waiting. Queuing Theory can't handle this problem, so we need to resort to DES instead. In the simulation code, it would be enough to add some additional logic to model this case accurately.

4.5 Summary

In this chapter, we have explored the complementary relationship between the mathematical elegance of Queuing Theory and the computational power of Discrete Event Simulation (DES). We began by establishing the fundamental vocabulary of systems engineering: entities, resources, and queues. We learned that while Queuing Theory provides exact, closed-form solutions for performance metrics like L and W_q, it is bounded by the assumptions. When a system involves complex routing, time-varying arrival rates, or resource dependencies, analytical formulas become intractable. This is where Discrete Event Simulation takes over. By modeling a system not as a continuous flow but as a sequence of distinct events managed by a Future Event List (FEL) and a Next-Event Time Advance clock, we can reconstruct complex reality inside a computer. However, we also demonstrated that simulation is not a replacement for theory, but an extension of it. Through the $M/M/1$ Case Study, we showed that Queuing Theory serves a critical role as a verification tool. By simplifying a simulation model to match standard theoretical assumptions, we can mathematically validate the underlying code before re-introducing real-world complexities.

4.6 Exercises

1. Consider a cloud computing server that processes requests (jobs). The arrivals follow a Poisson distribution with a mean rate of λ jobs per minute, and the service times are exponentially distributed with a mean rate of μ jobs per minute. Assume $\lambda = 4$ jobs/min and $\mu = 5$ jobs/min. Calculate the utilization factor ρ, the average waiting time in the queue W_q and the average number of jobs in the system L.

2. Due to a marketing campaign, the arrival rate increases slightly to $\lambda = 4.8$ jobs/min. The server speeds remains at $\mu = 5$. Calculate the new average waiting time in the queue W_q. By what percentage did the waiting time increase? Compare it to the traffic increase and explain the difference.

3. Modify the provided Python code to simulate a $M/M/3$ model instead (i.e. with 3 parallel servers). What needs to be changed in the code? Compare the simulated Average Waiting Time W_q in the case $\mu = 5$ with 3 servers and $M/M/1$ with $\mu = 15$. Are they the same? Why?

Part II

PART II: OPTIMIZATION

5 Optimization basics

5.1 Introduction

We now switch our focus to the theory and practice of optimization as one of the building blocks of modern AI. *Optimization* is the mathematical discipline concerned with finding the "best" solution from a set of available alternatives. It is the invisible utility that powers the modern world. When you type a query into a search engine, an optimization algorithm determines the most relevant ranking of results. When a packet travels across the internet, routing protocols solve shortest-path optimization problems to minimize latency. When a logistics company schedules thousands of deliveries, they are solving complex integer programming problems to minimize fuel consumption and time.

Optimization is not only a new toolbox of solvers: it's a fundamental language for modeling the world. Beyond the digital realm, optimization bridges software with the physical world. This is the domain of **Operations Research**, where mathematical models dictate the efficiency of e.g.

- Energy grids, where the goal is to balance electricity generation with demand in real-time to prevent blackouts while minimizing costs.
- Suppy chains, where the goal is to determine how many units of a product to ship from a specific set of warehouses to a set of retailers.
- Finance, where the goal is to construct investment portfolios that maximize returns for a specific level of risk.

However, one of the most compelling reasons for a modern computer scientist to master optimization lies in the current revolution in Artificial Intelligence and Machine Learning (ML). It is no exaggeration to say that **Machine Learning is optimization**. While the architecture of a model (e.g., a Convolutional Neural Network or a Transformer) dictates how information flows, optimization dictates *what* the model actually learns. When we "train" a model, we are mathematically traversing a high-dimensional landscape, searching for a specific set of parameters (weights and biases) that minimizes a loss function. Whether it is a simple Linear Regression minimizing mean squared error or a Large Language Model (LLM) minimizing next-token prediction error, the mechanism is an optimization algorithm—typically a variant of Stochastic Gradient Descent (SGD). In Reinforcement Learning (RL), agents are explicitly designed to maximize a cumulative reward function over time, often requiring complex policy optimization techniques.

As we proceed through this part of the book, you will learn that finding the true best solution is not always computationally feasible. In Linear Programming, we can often find the exact global optimum efficiently. In Integer Programming, the problem becomes NP-hard, forcing us to rely on clever search techniques like Branch and Bound. In the rugged, non-convex landscapes of Deep Learning, we often settle for local optima, relying on Heuristics to guide us.

Understanding the mathematical foundations of optimization—convexity, gradients, duality, and complexity—allows you to distinguish between a problem that can be solved perfectly in milliseconds and one that requires an approximation to solve before the heat death of the universe. This chapter provides the roadmap for making those distinctions.

5.2 General Problem Formulation

We will now start developing the mathematical machinery needed to understand this part of the book. To treat optimization as a computational discipline, we must abstract away the specific details of the application (whether it is protein folding or portfolio management) and map the problem into a standardized mathematical structure.

Almost all optimization problems can be expressed in the following canonical form:

$$\begin{aligned} &\min_x f(x) \\ \text{subject to: } &g_i(x) \leq 0, \text{ for } i = 1, \dots, m \\ &h_j(x) = 0, \text{ for } j = 1, \dots, p \end{aligned}$$

This canonical structure contains several distinct components that dictate the difficulty and the strategy of the solution.

The **decision variables** $x \in \mathbb{R}^n$ represent the unknowns we wish to determine. In computer science contexts, these are often referred to as *parameters* or *weights*. In linear programming, x might represent the flow of traffic across n network edges. In deep learning, x might represent the billions of parameters of a deep neural network. In integer programming, x is restricted to integer values $\mathbb{Z}^n$ which might represent discrete choices (e.g. to select a specific server, whether to invest in an asset or not).

The **objective function** $f : \mathbb{R}^n \to \mathbb{R}$ represents the *cost* or *quality* of the solution. Usually, the convention is that we solve minimization problems, so if the problem requires to maximize a measure of utility $u(x)$ instead, we minimize the negative $\min_x -u(x)$. The specific form of the objective function is an important part of the difficulty of the problem. If f is just is a linear "plane", the direction of improvement is constant. However, if f is a rugged "mountain landscape" (common in non-convex neural network loss surfaces), the direction changes constantly depending on where you are in the space.

The **constraints** g_i and h_j represents limits or restrictions we are bound to in real systems, like physical, financial and logical limits. Inequality constraints ($g_i(x) \leq 0$) usually represent resource limits or thresholds. For instance, if we want to express that the latency has to be under 100 ms, we might write latency$(x) \leq 100$, or in the canonical form latency$(x) - 100 \leq 0$. On the other side, equality constraints ($h_j(x) = 0$) usually represent strict structural requirements. For instance, the conservation of flow equations we saw in **?@sec-sec-monte-carlo**. If both $m = 0$ and $p = 0$, we call the problem *unconstrained* and handle it usually by means of techniques like gradient descent.

The intersection of all constraints is usually referred to as the **feasible region**, denoted by Ω:

$$\Omega = \{x \in \mathbb{R}^n \mid g_i(x) \leq 0, \forall i, h_j(x) = 0, \forall j\} \tag{5.1}$$

Our task is then to find a point $x^* \in \Omega$, such that $f(x^*) \leq f(x)$ for all $x \in \Omega$. We then call x^* a **global optimum** of f to distinguish it from **local optima** where the point is merely better than its immediate neighbours.

If f and its constraints turn out to be **convex**, there are usually guarantees that any local minimum will be a global minimum as well, making the problem solvable efficiently. A function f is said to be convex if it satisfies the following property: For any $t \in [0, 1]$ and any two points x_1, x_2 in the domain of the function, we have:

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \tag{5.2}$$

Prominent examples of convex functions include linear, quadratic and exponential functions. Intuitively, this means that if we draw a line segment between any points in the function's curve $f(x_1)$ and $f(x_2)$, this line will always be either above or on the curve itself. When the function to optimize or any of the constraints are non-convex (e.g. deep learning), finding the global optimum becomes NP-hard (although usually reasonable approximations are good enough in practice).

5.3 Classification of Problems

Depending on the form of the function to be optimized, the constraints and the domains of the decision variables, we obtain different types of optimization problems. We will now explore the different classifications used for characterizing optimization problems.

5.3.1 Continuous vs. Discrete

One of the most significant criteria is whether the decision variables are continuous or discrete.

In **continuous optimization** problems, the decision variables live in a continuous space (usually $\mathbb{R}^n$). If the objective function is smooth (i.e. differentiable), we can then calculate gradients and use a method like gradient descent to find an optimum of the function. For example, training a neural network under the usual circumstances is a continuous optimization problem.

In **discrete optimization** problems, the decision variables belong to a discrete set like $\mathbb{Z}$ or $\{0, 1\}$ and therefore combinatorial in nature. In this case, we can't rely on gradients to optimize the function, and other (more computationally complex) are needed. For instance, the traveling salesman problem already presented in Chapter 1 is a typical example of a discrete optimization problem. These problems often face *combinatorial explosion*. As the number of variables increases, the number of possible configurations grows exponentially. Consequently, many discrete problems are NP-Hard.

In **mixed-integer optimization**, the decision variables contain both discrete and continuous variables. For instance, a power plant might optimize the continuous amount of power to generate (Megawatts) while making discrete decisions about which generators to turn on or off. Mixed-integer problems are therefore combinatorial in nature as well and face the same kind of problems than purely discrete optimization problems.

5.3.2 Unconstrained vs. Constrained

When the feasible region is equals to the domain of the objective function, we say that the problem is **unconstrained**. In this case, we just want to find the minimum of f anywhere in the domain of the function. This is usually the case in neural network training, where our goal is to find network weights which are normally not constrained in any specific way.

By contrast, **constrained optimization** problems place a special focus on finding optimal solutions that satisfy a specific set of conditions. For example, a self-driving car controller might minimize travel time (objective) subject to the strict constraint that the car must stay within the lane boundaries (constraints). Solving these requires specialized techniques like Lagrange Multipliers or projection methods.

5.3.3 Deterministic vs. Stochastic

In **deterministic optimization** problems, we assume that the objective function and constraints are known perfectly and do not change. There is no uncertainty when evaluating the objective function: any given input will deterministically produce the same output.

In **stochastic optimization**, the objective function involves some random quantity, so we usually resort to minimizing expectations instead. This is the native language of ML. Since we cannot calculate the loss over the true data distribution (which is unknown), we estimate it using finite, noisy batches of data. Stochastic Gradient Descent (SGD) is named precisely because it treats the true gradient as a random variable to be estimated.

5.3.4 Linearity and Convexity

Finally, it is the *geometry* of the function and the constraints which largely determines the computational complexity of the problem. There are three cases that we might face:

- Both the objective and the constraints are **linear**: We call this **linear programming (LP)** and there are efficient methods for solving this type of problems.
- All functions are convex, but some are **non-linear**: The objective is curved, but has only one minimum. These are solvable in polynomial time.
- There is some **non-convex** function: The landscape has hills, valleys, and saddle points. Finding the global minimum is generally intractable. Most modern DL falls into this category, forcing us to rely on heuristics and accept "good enough" approximations or local optima.

5.4 Mathematical Prerequisites

In this section, we will review some mathematical background knowledge that will be needed across this part of the book. We will frame some fundamental concepts in linear algebra and calculus though the lens of optimization theory to provide a strong foundation for the chapters to come.

In optimization, we treat data and parameters as vectors in space. We use norms to measure errors, gradients to find directions of improvement, and Hessians to understand the landscape's terrain.

5.4.1 Vector spaces and norms

Optimization algorithms operate in vector spaces like $\mathbb{R}^n$. To determine if one solution is "better" or "closer" to the target than another, we need a rigorous way to measure size and distance. This is the role of the **norm**, which is a function $\|\cdot\| : \mathbb{R}^n \to \mathbb{R}$ that takes a vector and returns a non-negative value that indicates its "size". Some typical examples of norms widely used in ML and optimization are the L_2, L_1 and L_∞ norms.

The L_2 norm is also known as **Euclidean norm** and is defined by:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x^T x}$$

This norm can be interpreted as the usual, "straight-line" distance. Geometrically, constraining the L_2 norm of a vector using an upper value e.g. $\|x\|_2 \leq 1$ results in a sphere.

The L_1 norm is also known as the **Manhattan norm** and is defined by:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

Instead of using the sums of squares like the L_2 norm, the L_1 norm just uses the absolute values. Geometrically, a region like $\|x\|_1 \leq 1$ can be interpreted as a polytope (i.e. a "diamond" form).

Lastly, the **infinity norm** is defined by:

$$\|x\|_\infty = \max_i |x_i|$$

which represents the magnitue of the largest single component of x. It is frequently used in robust optimization and analyzing worst-case scenarios (e.g., adversarial attacks on neural networks).

5.4.2 Matrix Calculus

In general, in order to miminize a function $f(x)$, we need to know how f changes with slight perturbations of x. In high dimensions, single-variable derivatives are replaced by vectors and matrices of derivatives.

The **gradient** is the vector of first-order partial derivatives:

$$\nabla f(x) = \begin{bmatrix} \dfrac{\partial f(x)}{\partial x_1} \\ \dfrac{\partial f(x)}{\partial x_2} \\ \vdots \\ \dfrac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

The gradient has a distinct geometric interpretation: it points in the direction of steepest ascent. Consequently, the direction of steepest descent, which is the path used by nearly all training algorithms in DL is $-\nabla f(x)$.

While the gradient tells us the slope, it tells us nothing about the curvature. This information is captured by the **Hessian**, the symmetric $n \times n$ matrix of second-order partial derivatives:

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots \\ \vdots & \ddots & \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Intuitively, ff the Hessian is "large" (high curvature), the gradient changes rapidly, and algorithms must take small steps to avoid overshooting. If the Hessian is "small" (flat curvature), algorithms can take larger steps.

5.4.3 Eigenvalues and Definiteness

In optimization, it is usually important to know whether the Hessian is *positive definite*. This is the matrix analogue of a number (scalar) being positive. In a nutshell, (semi-)positive definiteness of the Hessian will indicate convexity (in general) and also if a local optimum is a maximum or a minimum.

We say that a symmetric matrix A is **positive semidefinite (PSD)** if and only if $x^T A x \geq 0$ for all x different from zero. Alternatively, A is PSD if and only if all eigenvalues are non-negative $\lambda_i \geq 0$.

For a square matrix A, an eigenvector v and eigenvalue λ satisfy $Av = \lambda v$. The eigenvalues represent the "scaling factors" of the matrix along its principal axes. In the context of a loss landscape, the eigenvalues of the Hessian describe the curvature in different directions: large λ (in absolute value) indicates steep curvature, while small λ (in absolute value) indicates flat curvature.

Let x be such that $\nabla f(x) = 0$ (i.e. x is a singular point). In order to check if x is a maximum, minimum or a saddle point, we check the Hessian:

- If $\nabla^2 f(x)$ is positive definite, then x is a **local minimum**.
- If $\nabla^2 f(x)$ is positive negative, then x is a **local maximum**.
- If $\nabla^2 f(x)$ has mixed signs, then x is a **saddle point**.

Importantly, a function is convex if and only if *its Hessian is positive semidefinite everywhere.* This guarantees that any local minimum is **global**.

5.5 Why Convexity Is Good?

In mathematics in general, the distinction between easy and hard problems is done based on whether the problems are linear (easy) or non-linear (hard). However, in optimization, this distinction would be misleading. The defining property for optimization problems to be easily solvable or not is actually **convexity**. If a problem is convex, we can generally solve it efficiently and guarantee that our solution is the global optimum. If it is non-convex, we are entering a world of NP-hardness, local traps, and heuristics.

Optimization problems are constrained to a feasible region Ω (see Equation 5.1). The geometry of this region dictates how we can move through the search space. In general, a set $C \subseteq \mathbb{R}^n$ is said to be **convex** if and only if for any two points $x, y \in C$, the line segment that joins those two points is entirely contained in C. More formally:

$$\theta x + (1-\theta)y \in C, \text{ for all } x, y \in C, \text{ and } \theta \in [0,1]$$

Intuitively, a circle and a square are convex; a star shape or a crescent moon are not.

While convex sets describe the constraints, convex functions describe the objective $f(x)$. For the sake of completeness, we re-state the definition of a convex function: for all x, y in the domain of the function, we have:

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y),\ \forall \theta \in [0,1]$$

To analyze algorithms, we rely on differential definitions of convexity. The **first-order condition** states that

$$f(y) \geq f(x) + \nabla f(x)^T (y - x)$$

which means that the tangent plane (the first-order Taylor approximation of the function around any point x) is always *below* the function everywhere (for all y). So if we have that x is a singular point ($\nabla f(x) = 0$), this automatically means that x is a global minimum (since $f(y) \geq f(x)$).

The **second-order condition** states that the Hessian is positive semidefinite everywhere $\nabla^2 f(x) \succeq 0$, which means that the curvature is always "upwards". This implies that the function is convex everywhere.

Now if the objective function $f(x)$ is convex *and* the feasible region Ω is a convex set, we have:

1. Any local minimum is a **global minimum**.

2. The set of global minima is itself a **convex set** (so any convex combination of global minima is itself a global minimum).

This means that, in a convex optimization problem (e.g., linear regression, Support Vector Machines, Linear Programming), we do not need to worry about initialization. We can start the algorithm anywhere in the feasible region, follow the negative gradient, and we are mathematically guaranteed to converge to the best possible solution. However, in a non-convex optimization problem (e.g., DL), this guarantee vanishes. The loss landscape of a neural network is riddled with local minima, saddle points, and flat plateaus. A gradient descent algorithm might get stuck in a suboptimal valley, which is why training deep networks requires "tricks" like random restarts, momentum, and stochastic noise to escape local traps.

5.6 Optimality Conditions

We now turn our attention to the problem of certifying if a singular point x found by an optimization algorithm is indeed a global minimum (i.e. a stopping condition). In a sorting algorithm, we stop when the list is ordered. In optimization, we stop when we satisfy a specific set of mathematical conditions. These conditions differ depending on whether we are free to move anywhere (unconstrained) or are bounded by limits (constrained). Therefore, we will distinguish between unconstrained and constrained optimization problems, since the optimality conditions here differ most significantly.

5.6.1 Unconstrained optimization

In the case where we have no boundaries (constraints), we rely on calculus to identify "flat" spots in the landscape. The first condition is a necessary condition related to stationarity. If a candidate point $x*$ is a local minimizer and f is continuously differentiable, then the gradient must vanish: $\nabla f(x^*) = 0$. However, this condition is not enough, since x^* could now be a maximum, a minimum or a saddle point.

The sufficient condition is the previously mentioned second-order condition for positive semidefiniteness of the Hessian: if $\nabla f(x^*) = 0$ *and* $\nabla^2 f(x^*)$ is positive definite (i.e. all eigenvalues are positive), then x^* is a **strict local** minimizer.

In high-dimensional non-convex optimization (like DL), the Hessian often has both positive and negative eigenvalues. These are saddle points. In high dimensions, local minima are actually rare, and the vast majority of critical points are saddle points. Standard Gradient Descent can get "stuck" near saddle points because the gradient is zero, but the Hessian reveals there is still a direction to escape (which is the eigenvector corresponding to the negative eigenvalue).

5.6.2 Constrained Optimization: The KKT Conditions

When we add constraints, the gradient does not have to be zero. Imagine a ball rolling down a hill but hitting a fence. The ball stops at the fence, even though the hill continues downward on the other side. At this optimal point, the "force" of gravity (the objective gradient) is perfectly balanced by the "force" of the fence (the constraint gradient). To solve this problem, we use the method of **Lagrange Multipliers**.

The basic idea is to convert the constrained problem into an unconstrained one by means of a new objective function, the Lagrangian:

$$\mathcal{L}(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{p} \nu_j h_j(x)$$

We call the λ_i and ν_j the **dual variables**. They represent the cost or penalty of violating a constraint. Using these dual variables, we can state the general **Karush-Kuhn-Tucker (KKT)** conditions, which are fundamental conditions for a point (x^*, λ^*, ν^*) to be optimal in any general optimization problem:

1. **Stationarity:** The gradient of the Lagrangian with respect to x is 0 $\nabla_x \mathcal{L} = 0$.

$$\nabla_x \mathcal{L}(x^*, \lambda^*, \nu^*) = \nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla g_i(x^*) + \sum_{j=1}^{p} \nu_j^* \nabla h_j(x^*) = 0$$

2. **Primal Feasibility:** The solution x^* must satisfy all constraints.

$$g_i(x^*) \leq 0, \text{ for all } i,\ h_j(x^*) = 0, \text{ for all } j$$

3. **Dual Feasibility:** The Lagrange multipliers for the inequality constraints must be non-negative.

$$\lambda_i^* \geq 0, \text{ for all } i$$

4. **Complementary Slackness:** For each inequality constraint, we have:

$$\lambda_i^* g_i(x^*) = 0, \text{ for all } i$$

This last condition is the cornerstone of understanding how constraints interact with the objective function at an optimum. It's often the most counter-intuitive part of the KKT conditions, but it holds the key to why optimization algorithms work and how they identify the most critical elements of a problem. The core idea is that at an optimal solution x^*, an inequality constraint can behave in one of two possible ways:

- The constraint is **active**: this is the case when $g_i(x^*) = 0$, which means that x^* lies *exactly on the boundary* defined by the constraint, like the ball hitting the fence. This makes automatically the product $\lambda_i^* g_i(x^*) = 0$, so λ_i^* can take any non-negative value. Any non-zero value for λ_i^* indicates that the constraint is actively influencing the optimum. We can think of this multiplier as the *marginal cost* of the constraint. If we would relax the constraint by a small amount ϵ ($g_i(x^*) \leq \epsilon$), the optimal value would improve by about $\lambda_i^* \epsilon$. The constraint is "pushing back" so that x^* cannot move further.
- The constraint is **inactive**: this is the case when $g_i(x^*) < 0$, which means that x^* lies *inside the feasible region* specified by the constraint, like the ball away from the fence (but inside). In this case, for the product $\lambda_i^* g_i(x^*)$ to be zero, then λ_i^* must be zero as well. This means that the constraint is not actively influencing the optimal solution. The constraint is effectively irrelevant for finding the optimum.

5.7 Optimization in AI and ML

For many decades, the field of AI was dominated by logic, search, and symbolic reasoning. However, most current AI systems are data-driven, and the engine that drives learning from data is mathematical optimization. In this paradigm, "learning" is simply a synonym for "minimizing a loss function". Every time a ML model is trained, whether it is a simple linear regressor or a trillion-parameter Large Language Model (LLM), we are solving an optimization problem. The translation from a learning task to an optimization problem follows a standard recipe:

1. **The Hypothesis Space:** We define a model $f(x; \theta)$ parametrized by a vector θ which represents the decision variables of the corresponding optimization problem. For instance, in a neural network, θ represents all the weights and biases of the network.
2. **The Loss Function:** We define a scalar function $L(\theta)$ that measures the discrepancy between the model's predictions and the ground truth. In regression problems, this function takes often the form of the mean squared error (MSE). In classification problems, we use other functions like the cross-entropy loss.

The goal is then to find $\theta^* = \arg\min_\theta L(\theta)$.

5.7.1 Gradient descent

Since solving analytically (e.g. by setting $\nabla L(\theta) = 0$ and solving for θ) is most commonly not feasible, we rely on iterative algorithms like **Gradient Descent**. The most basic (and computationally efficient) variant is to move in the direction of steepest descent by following the negative gradient:

$$\theta_{t+1} = \theta_t + \eta \nabla L(\theta_t)$$

where η is called *step size*. The main idea is therefore to move in small steps towards the minimum so that we don't get lost on the way by taking too large steps.

In modern ML, calculating the gradient exactly requires summing over the entire dataset, which might be composed of millions of data points (images, documents, etc). This is computationally prohibitive. Instead, we estimate the gradient using a small random subset (also called a *minibatch*) of data.

$$\theta_{t+1} = \theta_t + \eta \hat{g}_t$$

where $E[\hat{g}_t] = \nabla L(\theta)$. This is called **Stochastic Gradient Descent (SGD)** and is a cornerstone of modern ML training procedures. Because the gradient is noisy, SGD does not settle at a single point but "bounces around" the minimum. This noise can actually be beneficial, helping the algorithm escape shallow local minima and saddle points.

5.7.2 Regularization

One of the biggest challenges in ML is overfitting: memorizing the training data rather than learning general patterns. We combat this using **regularization**, which is mathematically equivalent to constrained optimization. Consider for instance Ridge Regression, which involves adding a penalty term to the loss function:

$$\min_{\theta} L(\theta) + \lambda \|\theta\|_2^2$$

Does this ring a bell? This is, in fact, equivalent to using Lagrange multipliers to solve the equivalent constrained optimization problem:

$$\min_{\theta} L(\theta) \text{ subject to } \|\theta\|_2^2 \leq C$$

By constraining the Euclidean norm of the weights, we force the model to be smooth and more simple. Similarly, Lasso regression uses the L_1 norm instead. As discussed in Section 2.1, the geometry of the L_1 ball (a diamond) tends to touch the loss contours at the axes, forcing many weights to become exactly zero. This performs automatic feature selection, identifying the most important variables in the dataset.

5.8 Summary

Optimization is the mathematical tool that transforms the question of "is this possible?" into "what is the best way to do this?". This chapter established the mathematical framework necessary to understand and design the algorithms that power everything from network routing to Large Language Models.

We began by reviewing the essential tools of multivariate calculus and linear algebra, focusing on how gradients define the direction of improvement and how Hessians and eigenvalues characterize the curvature of the search landscape. We identified convexity as the fundamental concept in optimization theory. Unlike the distinction between linear and non-linear, the distinction between convex and non-convex determines tractability: convex problems guarantee that any local minimum is a global minimum, allowing for efficient polynomial-time solutions.

We stated the Karush-Kuhn-Tucker (KKT) conditions, the universal set of requirements (stationarity, primal/dual feasibility, and complementary slackness) that certify whether a solution is optimal in constrained environments. We also explored the computational complexity of these problems, noting how discrete variables (Integer Programming) and high-dimensional non-convex landscapes (Deep Learning) introduce NP-hardness and the "curse of dimensionality", forcing us to rely on approximations and heuristics.

Finally, we connected these abstract principles to AI and ML, demonstrating that training a machine learning model is fundamentally an optimization task. By framing learning as loss minimization, we saw how concepts like Stochastic Gradient Descent and regularization are direct applications of the calculus and geometric constraints discussed throughout the chapter. This foundation sets the stage for other specialized techniques covered in the remainder of this book.

5.9 Exercises

1. A data center has N servers. Each server i has a maximum CPU capacity C_i and a maximum RAM capacity R_i. It is needed to schedule M different jobs. Each job j requires c_i units of CPU and r_j units of RAM to run. Let x_{ij} be a binary variable indicating if job j is assigned to server i ($x_{ij} = 1$) or not ($x_{ij} = 0$). Write down the full mathematical optimization model. Identify the objective function, the decision variables, and the constraints. Is this a convex optimization problem? Why?

2. Consider the following function $f : \mathbb{R}^2 \to \mathbb{R}$:

$$f(x, y) = x^3 + y^3 - 3xy$$

   Compute the gradient $\nabla f(x, y)$ and the Hessian $\nabla^2 f(x, y)$. Find all critical points ($\nabla f(x, y) = 0$). Evaluate the Hessian at the critical points and calculate the eigenvalues

of the Hessian there. Then, classify each critical point as maximum, minimum or saddle point. Is $f(x, y)$ globally convex?

3. Solve the following constrained optimization problem using the KKT conditions:

$$\min_{x,y} f(x, y) = x^2 + y^2$$
$$\text{subject to } 2x + y \geq 5$$

Specifically, formulate the Lagrangian $\mathcal{L}(x, y, \lambda)$ and write down the four KKT conditions for this problem. Solve the system of equations and consider two cases based on complementary slackness: either $\lambda = 0$ (constraint is inactive) or $\lambda > 0$ (constraint is active). Verify which case yields a valid solution and state the optimal x^*, y^*.

4. In ML, Ridge Regression aims to fit a linear model $y = Xw$ while at the same time keeping the weights small to prevent overfitting. This can be formulated using the following objective function:

$$L(w) = \|y - Xw\|_2^2 + \lambda \|w\|_2^2$$

where $y \in \mathbb{R}^m$ is the vector of labels, $X \in \mathbb{R}^{m \times n}$ is the training data matrix, $w \in \mathbb{R}^n$ is the weight vector, and $\lambda > 0$ is a regularization hyperparameter. Compute the gradient $\nabla_w L(w)$ with respect to w (Hint: $\|v\|_2^2 = v^T v$) and equate it to zero to solve for w. If you solved correctly, you should get:

$$w^* = (X^T X + \lambda I)^{-1} X^T y$$

What does the λI term mean in terms of invertibility and convexity?

6 Exact methods

6.1 Introduction

Linear Programming (LP) is arguably the most successful algorithmic framework in history. Developed independently by Kantorovich and Dantzig in the 1940s, it remains the standard tool for allocating resources, from routing internet traffic to optimizing global supply chains. At the core of LP lies the assumption that the world behaves in a **linear** way. For instance, if it costs 10€ to run a server, it will costs 100€ to run 10 servers (proportionality). This also means that we discard scale effects or diminishing returns. Additionally, we assume that total costs are always the sum of individual costs. There are no second- or higher-order interactions between the variables. While these assumptions may seem restrictive, they approximate a vast number of real-world systems well enough to be useful.

A linear program in standard form is normally expressed as:

$$\begin{aligned} \min_{x} \ & c^T x \\ \text{subject to } & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{6.1}$$

In this model, $x \in \mathbb{R}^n$ are the decision variables, which we wish to determine. The vector $c \in \mathbb{R}^n$ is the **cost vector**. The matrix $A \in \mathbb{R}^{m \times n}$ contains all the linear constraints (one per row), describing how the variables interact with resources. The vector $b \in \mathbb{R}^m$ thus represents a **resource limit vector**, establishing an upper bound on resource usage. The non-negativity constraint $x \geq 0$ implies that we can't produce negative units of a product.

At first sight, this formulation seems somewhat restrictive. The reason is that it assumes a very specific interpretation in terms of minimimization of costs and limits on resources, and some solvers actually expect the problem to be described in this form. However, with some changes, we can express more general problems:

- Maximization instead of minimization: replace $\min_x c^T x$ with $\max_x -c^T x$.
- Use of equality constraints: a constraint $a^T x = b$ can be expressed by two inequality constraints $a^T x \leq b$ and $-a^T x \leq -b$.
- If x can be negative, we can replace x with two non-negative variables u, v such that $x = u - v$.

As an example, consider the following **Server Allocation Problem**: A cloud provider has two types of jobs: CPU-bound and memory-bound. Each job type requires a certain amount of CPU and memory resources, and generates a specific revenue. The provider has a limited amount of CPU and memory resources available. The goal is to determine how many jobs of each type to run in order to maximize the total revenue, without exceeding the available resources. Let's define the decision variables: let x_1 be the number of CPU-bound jobs to run, and x_2 be the number of memory-bound jobs to run. The revenue generated by each CPU-bound job is 5€, and each memory-bound job generates 8€. The cloud provider has a total of 100 CPU units and 80 memory units available. Each CPU-bound job requires 2 CPU units and 1 memory unit, while each memory-bound job requires 1 CPU unit and 3 memory units. We can formulate this problem as a linear program as follows:

$$\begin{aligned} \max_{x_1, x_2} \quad & 5x_1 + 8x_2 \\ \text{subject to } & 2x_1 + 1x_2 \leq 100 \quad \text{(CPU constraint)} \\ & 1x_1 + 3x_2 \leq 80 \quad \text{(Memory constraint)} \\ & x_1, x_2 \geq 0 \quad \text{(Non-negativity constraint)} \end{aligned}$$

In this formulation, the objective function $5x_1 + 8x_2$ represents the total revenue from running x_1 CPU-bound jobs and x_2 memory-bound jobs. The constraints ensure that the total CPU and memory usage does not exceed the available resources.

6.1.1 Using Geometry to Solve LPs

Linear programs can be solved using various algorithms, with the most common being the Simplex method and Interior Point methods. Both methods leverage the geometric properties of linear programs.

Think about a linear inequality in the form $a^T x \leq b$. This inequality describes a half-space in n-dimensional space, where a is the normal vector to the hyperplane defined by $a^T x = b$. The feasible region of a linear program is the intersection of all such half-spaces defined by the constraints. This feasible region is a convex polyhedron (or polytope), meaning that any line segment connecting two points within the region lies entirely within the region.

- In 2D, this feasible region appears as a polygon.
- In 3D, it appears as a polyhedron.
- In higher dimensions, it is referred to as a polytope (a "hyper-diamond" shape).

Because the intersection of convex sets is convex, the feasible region of an LP is always convex. This guarantees that if we find a local minimum, it is also the global minimum (see Chapter 5).

Now consider the objective function $c^T x$. This function defines a family of parallel hyperplanes in n-dimensional space, each corresponding to a different value of the objective function. The

goal of the LP is to find the point in the feasible region that lies on the hyperplane with the lowest (or highest, for maximization problems) value of $c^T x$. Now if the polytope is bounded (a polytope rather than a polyhedron), the optimal solution will always be located at one of the vertices (or corners) of the polytope. This is because the objective function is linear, and thus its maximum or minimum value over a convex set occurs at an extreme point of that set. This result is known as the **Fundamental Theorem of Linear Programming** and is the foundation for algorithms like the Simplex method, which systematically explores the vertices of the feasible region to find the optimal solution.

6.2 The Simplex Method

The Simplex method is an iterative algorithm that starts at a vertex of the feasible region and moves along the edges of the polytope to adjacent vertices, seeking to improve the objective function value at each step. The algorithm continues this process until it reaches a vertex where no adjacent vertices offer a better objective function value, indicating that the optimal solution has been found. Invented by George Dantzig in 1947, the Simplex method is widely considered one of the top 10 algorithms of the 20th century. It transforms the continuous optimization problem into a graph traversal problem.

The Simplex method can be summarized in the following steps:

1. **Initialization**: The algorithm starts at an arbitrary vertex of the polytope. If the origin ($x = 0$) is feasible, it can be used as the starting point; otherwise, an auxiliary sub-problem is solved to find an initial feasible vertex.
2. **Optimality Check**: From the current vertex, the algorithm looks along the edges connecting to neighboring vertices. Then it calculates the "reduced cost" (gradient) along each edge. If an edge leads "downhill" (improves the objective), the algorithm moves along that edge to the next vertex. If all edges leading out from the current vertex go "uphill" (worsen the objective), then—due to convexity—we know we are at the global minimum. We stop.
3. **Pivoting**: If an adjacent vertex offers a better objective function value, move to that vertex by performing a pivot operation.
4. **Iteration**: Repeat the optimality check and pivoting steps until the optimal solution is found.

Let's dive a bit deeper into the pivot operation. In a system with n variables and m constraints, a vertex is obtained by setting $n - m$ variables to zero (these are called the non-basic variables) and solving for the remaining m variables (the basic variables). The pivot operation involves selecting one non-basic variable to enter the basis (i.e., become a basic variable) and one basic variable to leave the basis (i.e., become a non-basic variable). This is done in such a way that the new vertex remains feasible and improves the objective function value.

6.2.1 A Step-by-Step Example

Consider the following example problem: Imagine a factory that produces two products, A and B. We denote by x_1 the number of units of product A and by x_2 the number of units of product B. The profit from each unit of product A is 3€, and from each unit of product B is 2€. There are in total 4 machine hours to produce, and it takes 1 hour to produce one unit of product A and 1 hour to produce one unit of product B. The factory has a limited amount of resources: product A requires 2 units of raw material per unit produced, and product B requires 1 unit of raw material per unit produced. The factory has a total of 6 units of raw material available.

We can formulate this problem as a linear program as follows:

$$\begin{aligned} \max_{x_1,x_2} \; & Z = 3x_1 + 2x_2 \\ \text{subject to } & 1x_1 + 1x_2 \leq 4 \quad \text{(Machine hours constraint)} \\ & 2x_1 + 1x_2 \leq 6 \quad \text{(Raw material constraint)} \\ & x_1, x_2 \geq 0 \end{aligned}$$

Step 1: Standarization

As a first step, we move the objective function to the left side to express it in standard form:

$$Z - 3x_1 - 2x_2 + 0s_1 + 0s_2 = 0 \quad \text{(Objective function)}$$

The Simplex method requires all constraints to be in the form of equalities. We can convert the inequalities into equalities by introducing slack variables. Let s_1 be the slack variable for the machine hours constraint and s_2 be the slack variable for the raw material constraint. The constraints become:

$$\begin{aligned} 1x_1 + 1x_2 + s_1 &= 4 \quad \text{(Machine hours constraint)} \\ 2x_1 + 1x_2 + s_2 &= 6 \quad \text{(Raw material constraint)} \\ s_1, s_2 &\geq 0 \end{aligned}$$

So we have $n = 4$ variables (x_1, x_2, s_1, s_2) and $m = 2$ constraints. This means that we need to set $n - m = 2$ variables to zero to find a vertex.

Step 2: Initial Basic Feasible Solution

We can start by setting the decision variables x_1 and x_2 to zero, which gives us the initial basic feasible solution (solving the system of equations for s_1 and s_2):

$$\begin{aligned} x_1 &= 0 \\ x_2 &= 0 \\ s_1 &= 4 \\ s_2 &= 6 \end{aligned}$$

We now represent the current state as a *tableau*, which is a compact way to keep track of the coefficients of the variables in the constraints and the objective function:

Basis	x_1	x_2	s_1	s_2	Solution
s_1	1	1	1	0	4
s_2	2	1	0	1	6
Z	-3	-2	0	0	0

Step 3: Optimality Check and Pivoting

Looking at the objective function row (the last row), we see that both x_1 and x_2 have negative coefficients (-3 and -2, respectively). This indicates that increasing either variable will improve the objective function value. We choose to increase x_1 since it has the most negative coefficient. **This is the variable that will enter the basis**.

Next, we need to determine which variable will leave the basis. This will be the variable that, when decreased, causes the solution to become infeasible. For this, we perform the minimum ratio test by dividing the solution values by the corresponding coefficients of x_1 in each constraint row: - For s_1: $4/1 = 4$ - For s_2: $6/2 = 3$ The minimum ratio is 3, so s_2 will leave the basis. We now perform a pivot operation to update the tableau. The main idea of the pivot operation is to make the coefficient of the entering variable (x_1) equal to 1 in the leaving variable's row (s_2) and 0 in all other rows by performing row additions and subtractions.

- We divide the entire row of s_2 by 2 to make the coefficient of x_1 equal to 1: $[1, 0.5, 0, 0.5, 3]$.
- Next, we update the row of s_1 by subtracting 1 times the new row of s_2 from it: $[0, 0.5, 1, -0.5, 1]$.
- Finally, we update the objective function row by adding 3 times the new row of s_2 to it: $[0, -0.5, 0, 1.5, 9]$.

The updated tableau looks like this:

Basis	x_1	x_2	s_1	s_2	Solution
s_1	0	0.5	1	-0.5	1
x_1	1	0.5	0	0.5	3
Z	0	-0.5	0	1.5	9

Now we have moved along the x_1 axis to a new vertex, which is (3,0) with an objective function value of 9. To find this new vertex, we need to solve the system of equations again using the new basis variables x_1 and s_1.

Continuing with the optimality check, we see that x_2 still has a negative coefficient (-0.5) in the objective function row. This means we can still improve the objective function by increasing x_2.

We repeat the pivoting process. The minimum ratio test gives us: - For s_1: $1/0.5 = 2$ - For x_1: $3/0.5 = 6$ The minimum ratio is 2, so s_1 will leave the basis. Performing the pivot operation, we get the new tableau:

Basis	x_1	x_2	s_1	s_2	Solution
x_2	0	1	2	-1	2
x_1	1	0	-1	1	2
Z	0	0	1	1	10

At this point, all coefficients in the objective function row are non-negative, indicating that we have reached the optimal solution. The optimal solution is $x_1 = 2$ and $x_2 = 2$, with a maximum profit of 10€.

6.2.2 Solving LPs using the Simplex Method in Python

The Simplex method is efficient in practice and can solve large-scale linear programs quickly, although its worst-case time complexity is indeed exponential. However, it remains one of the most widely used algorithms for solving LPs due to its practical performance and simplicity. Let's see how to solve the example using the Simplex method in Python implemented in the `scipy.optimize.linprog` function.

```
import numpy as np
from scipy.optimize import linprog

# Coefficients for the objective function (to be maximized)
c = [-3, -2]  # Negated for maximization
# Coefficients for the inequality constraints
A = [[1, 1],
     [2, 1]]
# Right-hand side of the inequality constraints
b = [4, 6]
# Solve the linear program using the Simplex method
res = linprog(c, A_ub=A, b_ub=b, method='highs')
# Print the results
print("Optimal value (max revenue):", -res.fun)  # Negate to get the original maximized value
print("Optimal solution (Product A, Product B):", res.x)
```

```
Optimal value (max revenue): 10.0
Optimal solution (Product A, Product B): [2. 2.]
```

6.3 Integer Linear Programming

In the previous example, the optimal solution happened to be integer-valued. However, this is not always the case. Linear programming allows decision variables to take on any real values, which can lead to solutions that are not practical in certain contexts. For example, you cannot produce a fraction of a product or hire half an employee. This leads us to the field of **Integer Linear Programming (ILP)**, where some or all decision variables are constrained to be integers.

Mathematicall, we "just" need to add integrality constraints to the LP formulation Equation 6.1:

$$\begin{aligned} \min_x \ & c^T x \\ \text{subject to } & Ax \leq b \\ & x \geq 0 \\ & x_i \in \mathbb{Z} \quad \text{for some or all } i \end{aligned}$$

This single constraint changes *everything*. It transforms the problem from a convex, polynomial-time task (P) into a **non-convex, NP-Hard** challenge. The nice, smooth edges of the polytope are replaced by a discrete grid of valid points, destroying the gradient information that the Simplex and other methods for LP rely on.

6.3.1 Definition and Variants

Depending on which variables are constrained to be integers, we have different variants of ILP:

1. **Pure Integer Linear Programming**: All decision variables are required to be integers. Example: the Knapsack Problem, where items can either be included or excluded from the knapsack (see Chapter 1).
2. **Mixed-Integer Linear Programming (MILP)**: Some decision variables are constrained to be integers, while others can take on continuous values. Example: a power plant optimization problem, where the number of power units to activate is an integer, but the amount of power generated can be continuous.
3. **Binary Integer Programming**: A special case of ILP where decision variables are restricted to binary values (0 or 1). Example: facility location problems, where a facility is either built (1) or not built (0). Additionally, this is the language of *logic* and *choice*. We can model logical propositions using linear constraints, for instance:
    - If we select option A, we cannot select option B: $x_A + x_B \leq 1$ where $x_A, x_B \in \{0, 1\}$.
    - If we build a warehouse, we can ship up to C units from it: $y \leq Cx$ where y is the number of units shipped and $x \in \{0, 1\}$ indicates if the warehouse is built.

In principle, one could round the solution of a linear program to the nearest integers to solve an ILP. However, this naive approach often leads to suboptimal or even infeasible solutions. For instance, consider the constraint $3x \leq 10$. The LP solution might yield $x = 3.33$, which rounds to $x = 3$ (which is feasible), however if the constraint would be $3x \geq 10$ instead, the rounded solution would be infeasible.

We could try to enumerate all possible integer solutions and select the best one, but this approach quickly becomes computationally infeasible as the number of variables increases, due to the combinatorial explosion of possible solutions. Because we cannot rely on gradients or rounding, we need a systematic way to search the discrete space without checking every single possibility. This leads us to the **Branch and Bound** algorithm.

6.3.2 The Branch and Bound Algorithm

Since we cannot round LP solutions, and we cannot brute-force check every integer combination (which would take billions of years for even moderate problems), we need a middle ground. We need a smart way to enumerate solutions that allows us to discard vast sections of the search space that are guaranteed to be suboptimal. The **Branch and Bound (B&B)** algorithm is a systematic method for solving ILPs by exploring the solution space in a tree-like structure. The main idea is to divide the problem into smaller subproblems (branching) and use bounds on the objective function to eliminate subproblems that cannot yield better solutions than the best one found so far (bounding).

The main idea for doing so is to work with the LP relaxation of the ILP. The LP relaxation is obtained by removing the integrality constraints from the ILP, allowing the decision variables to take on continuous values. This relaxed problem can be solved efficiently using LP methods like the Simplex method and provides a lower bound (for minimization problems) or an upper bound (for maximization problems) on the optimal ILP solution.

Imagine we are solving the LP relaxation of an ILP as a maximization problem, and we find $x_1 = 3.7$. We know that the optimal integer solution must be an integer ≤ 3 or ≥ 4. Thus, we can create two new subproblems (branches):

1. **Branch Left**: Add the constraint $x_1 \leq 3$ to the original ILP.
2. **Branch Right**: Add the constraint $x_1 \geq 4$ to the original ILP.

We have now effectively removed the fractional region $(3, 4)$ from the search space. We then solve the LP relaxation for each of these subproblems. If the solution to a subproblem is integer-valued, we compare it to the best-known integer solution and update our best solution if necessary. If the solution is fractional, we repeat the branching process on that subproblem. However, doing this blindly would lead to an explosion in the number of subproblems. This is where bounding comes into play. We use the *objective value* of the LP relaxation as a *bound*, as follows:

Step 1: Bounding: Let Z^* be the best-known integer solution found so far (called the *incumbent* solution). If the objective value of the LP relaxation of a subproblem is less than or equal to Z^* (for maximization problems), we can discard that subproblem from further consideration, as it cannot yield a better integer solution.

Step 2: Pruning: We stop exploring a branch when:

- The LP relaxation has no solution (infeasible).
- The LP relaxation is integer-valued (we have found a candidate solution). We compare it to Z^* and update if it's better. We stop because adding more constraints to this node will only make the objective worse.
- The objective value of the LP relaxation is worse than Z^* (we can prune this branch).

Step 3: Branching: If the LP relaxation yields a fractional solution, we select a variable with a fractional value and create two new subproblems by adding constraints to force that variable to take on integer values (as described above).

This process continues until all branches have been either explored or pruned. The best integer solution found during this process is the optimal solution to the original ILP.

6.3.3 Coding Example

We will now illustrate the Branch and Bound algorithm using Python's `PuLP` library, which provides a convenient interface for defining and solving ILPs. Let's consider a simple ILP example (Knapsack Problem): We have a knapsack which can hold 10 kg. There are four items available: A (value 15€, weight 4 kg), B (value 10€, weight 3 kg), C (value 9€, weight 2 kg) and D (value 5€, weight 1 kg). We want to maximize the total value of items in the knapsack without exceeding its weight capacity.

```
import pulp
from pulp import PULP_CBC_CMD

# 1. Initialize the Model (Maximization)
model = pulp.LpProblem("The_Knapsack_Problem", pulp.LpMaximize)

# 2. Define Decision Variables
# cat='Binary' tells the solver these are 0 or 1 integers.
# This triggers the Branch and Bound algorithm under the hood.
items = ['A', 'B', 'C', 'D']
weights = {'A': 4, 'B': 3, 'C': 2, 'D': 1}
values  = {'A': 15, 'B': 10, 'C': 9, 'D': 5}

x = pulp.LpVariable.dicts("Select", items, cat='Binary')
```

```python
# 3. Define Objective Function
model += pulp.lpSum([values[i] * x[i] for i in items]), "Total Value"

# 4. Define Constraint (Capacity <= 10)
model += pulp.lpSum([weights[i] * x[i] for i in items]) <= 10, "Capacity"

# 5. Solve
model.solve(PULP_CBC_CMD(msg=False))

# 6. Output Results
print(f"Status: {pulp.LpStatus[model.status]}")
print(f"Max Value: ${pulp.value(model.objective)}")
print("Selected Items:")
for i in items:
    if pulp.value(x[i]) == 1:
        print(f" - Item {i} (Value: {values[i]}, Weight: {weights[i]})")
```

```
Status: Optimal
Max Value: $39.0
Selected Items:
 - Item A (Value: 15, Weight: 4)
 - Item B (Value: 10, Weight: 3)
 - Item C (Value: 9, Weight: 2)
 - Item D (Value: 5, Weight: 1)
```

6.4 The Limits of Exact Methods

We have now seen two powerful algorithms: Simplex for Linear Programming and Branch & Bound for Integer Programming. In the examples provided (allocating server resources or packing a knapsack) these methods returned the provably optimal solution in milliseconds. It is tempting thus to conclude that optimization is a solved problem: If we can model it, we can solve it. Alas, this is an illusion: as we scale up the problem size, exact methods quickly become impractical. The Simplex method, while efficient for many practical problems, can exhibit exponential time complexity in the worst case. More critically, Integer Linear Programming is NP-Hard, meaning that no known polynomial-time algorithm exists to solve all instances of ILP optimally. As the number of variables and constraints increases, the time required to solve ILPs can grow exponentially, making them infeasible for large-scale problems.

In practice, the performance of B&B depends on the *quality of the bounds*: the tighter the bounds, the more branches can be pruned, and the faster the algorithm runs. However, for many

real-world problems, especially those with a large number of variables or complex constraints, even state-of-the-art ILP solvers can struggle to find optimal solutions within a reasonable timeframe. In the worst case, Branch and Bound degenerates into a brute force search.

In modern AI applications, such as optimizing the weights of a neural network (millions of variables) or routing a fleet of 10,000 delivery drones, exact methods simply cease to function. When exact methods fail, we must change our goal. We stop asking for the *best* solution and start asking for a *good* solution found fast.

This is the domain of **heuristic** and **metaheuristic** algorithms, which we will explore in the next chapter. Heuristics are meant to solve the following challenges of exact methods:

- **Latency:** A ride-sharing app like Uber needs to match a driver to a rider in milliseconds. It cannot run a 20-minute Branch and Bound job every time a user opens the app. It accepts a slightly suboptimal match (maybe the driver is 1 minute further away than the perfect choice) to ensure the system is responsive.
- **Data Uncertainty:** Why spend 100 hours computing the "perfect" supply chain schedule if the demand forecast is only 90% accurate? The noise in the data renders the precision of the exact solver meaningless.
- **Good enough Solutions:** In many cases, a solution that is 95% as good as the optimal one is perfectly acceptable, especially if it can be found in a fraction of the time. For example, in logistics, a delivery route that is slightly longer than the optimal route may still meet customer expectations while significantly reducing computation time.

In the next chapter, we will abandon the guarantee of optimality. We will explore metaheuristics: algorithms like Genetic Algorithms, Simulated Annealing, and Tabu Search. These methods do not use gradients, and they do not prove optimality. Instead, they use metaphors from nature (evolution, thermodynamics) to navigate the rugged, non-convex landscapes where Simplex and Branch & Bound do not perform well.

6.5 Summary

In this chapter, we explored exact methods for solving optimization problems, focusing on Linear Programming (LP) and Integer Linear Programming (ILP). We discussed the Simplex method, which efficiently solves LPs by navigating the vertices of the feasible region, and the Branch and Bound algorithm, which systematically explores the solution space of ILPs while pruning suboptimal branches. While these methods are powerful and can provide optimal solutions for many problems, they face significant challenges as problem size increases, particularly for ILPs, which are NP-Hard. This limitation motivates the need for heuristic and metaheuristic approaches in scenarios where quick, good-enough solutions are more practical than guaranteed optimality.

6.6 Exercises

1. **Formulating an LP**: A factory produces two products, X and Y. Each unit of product X requires 2 hours of labor and 3 units of raw material, while each unit of product Y requires 4 hours of labor and 2 units of raw material. The factory has a total of 100 hours of labor and 120 units of raw material available. The profit from each unit of product X is 30€, and from each unit of product Y is 20€. Formulate this as a linear programming problem to maximize profit.
2. **Solving an LP using Simplex**: Using the formulation from Exercise 1, implement the Simplex method in Python using `scipy.optimize.linprog` to find the optimal production quantities of products X and Y.
3. **Branch and Bound for ILP**: Consider the following integer linear programming problem: Maximize $Z = 10x_1 + 15x_2$ subject to the constraints $2x_1 + 3x_2 \leq 12$, $x_1 + x_2 \leq 5$, and $x_1, x_2 \geq 0$, with the additional constraint that x_1 and x_2 must be integers. Use the Branch and Bound method to solve this ILP manually, showing each step of the process.
4. **Implementing Branch and Bound**: Using the ILP from Exercise 3, implement the Branch and Bound algorithm in Python using the `PuLP` library to find the optimal integer solution.

7 Metaheuristics

7.1 Introduction

In the previous chapters, we operated in the realm of mathematical certainty. When the Simplex method terminates, it provides a certificate of optimality. When Branch and Bound finishes a search, we know with absolute confidence that no better integer solution exists. However, not all optimization problems can be solved exactly within a reasonable timeframe. Some problems are NP-hard, meaning that the time required to solve them grows exponentially with the size of the input. In such cases, exact methods become impractical for large instances.

To make progress, we must fundamentally change our goal. We shift from asking *What is the global minimum?* to asking *Can we find a solution that is good enough, quickly enough?*. Nobel laureate Herbert Simon coined the term *satisficing* (a portmanteau of satisfy and suffice) to describe this approach. In CS, this means trading optimality for efficiency. We accept a solution that might be 1% or 5% worse than the theoretical optimum if it allows us to find it in milliseconds rather than millennia.

In this context, we define a **heuristic** as an algorithm that produces a feasible solution to an optimization problem without guaranteeing optimality. Derived from the Greek *heuriskein* ("to find"), a heuristic is a problem-specific strategy designed to find a good solution quickly. Therefore, heuristics rely on domain knowledge and intuition. For instance, in the Traveling Salesman Problem (TSP), a simple heuristic is the nearest neighbor algorithm, which constructs a tour by repeatedly visiting the nearest unvisited city. While this method is fast and often yields a reasonable solution, it does not guarantee the shortest possible tour. It often gets trapped in poor local optima because it never makes a move that looks bad in the short term, even if it is necessary for long-term success.

A **metaheuristic** is a higher-level framework that guides the search process of heuristics, often inspired by natural processes. Metaheuristics are, in general, problem-independent. For instance, the same Genetic Algorithm logic used to design a jet engine nozzle can be used to tune the hyperparameters of a neural network. However, unlike simple heuristics, metaheuristics include mechanisms to escape local optima. They might accept a worse solution temporarily (Simulated Annealing) or maintain a population of diverse solutions (Genetic Algorithms) to avoid getting stuck.

The Exploration vs. Exploitation Dilemma

A central theme in metaheuristics is the balance between **exploration** and **exploitation**. Exploration involves searching new areas of the solution space to discover potentially better solutions, while exploitation focuses on refining known good solutions to find local improvements. An effective metaheuristic must strike a balance between these two strategies. Too much exploration can lead to inefficiency, while too much exploitation can cause premature convergence to suboptimal solutions.

A "pure" hill-climber is 100% exploitation. A random search is 100% exploration. The algorithms in this chapter (Simulated Annealing, Tabu Search, and Genetic Algorithms) are successful specifically because they dynamically manage the balance between these two extremes, starting with exploration and gradually shifting toward exploitation.

In the following, we will explore several popular metaheuristic algorithms, discussing their principles, implementations, and applications. We will divide our discussion into **trajectory-based** methods, which iteratively improve a single solution, and **population-based** methods, which evolve a set of solutions over time. In the first case, we will explore Simulated Annealing and Tabu Search as the most prominent examples. In the second case, we will study Genetic Algorithms.

7.2 Trajectory-Based Metaheuristics

The simplest way to explore a landscape is to send out a single hiker. In trajectory-based metaheuristics, we maintain a single candidate solution x and iteratively improve it by exploring its neighbors x'. The hiker evaluates nearby points and decides where to move next based on specific criteria. This approach is straightforward and often effective for many optimization problems. The sequence of solutions $x_0, x_1, x_2, ...$ generated by the algorithm forms a *trajectory* through the solution space.

The fundamental challenge in this type of methods is to avoid getting stuck in local optima. A standard "Hill Climber" (or Greedy Descent) algorithm accepts a neighbor only if it is better than the current solution. Consequently, once the hiker reaches a small peak, they get stuck, even if a massive mountain looms just across the valley. To cross that valley, the algorithm must be willing to temporarily accept a worse solution. To address this problem, trajectory-based metaheuristics incorporate strategies to escape local minima and continue exploring the solution space.

We will examine two dominant strategies for doing this: **Simulated Annealing** (which uses randomness) and **Tabu Search** (which uses memory).

7.2.1 Simulated Annealing

Simulated Annealing (SA) is inspired by the annealing process in metallurgy, where controlled cooling of a material allows atoms to settle into a low-energy configuration. In optimization,

SA mimics this process by allowing occasional uphill moves (i.e., accepting worse solutions) to escape local minima. The main idea is to perform the cooling process gradually, starting with a high "temperature" that permits more exploration and slowly lowering it to focus on exploitation.

In SA, our objective function is called "Energy", and the "Temperature" parameter controls the likelihood of accepting worse solutions. At high temperatures, the algorithm is more likely to accept uphill moves, promoting exploration. As the temperature decreases, the algorithm becomes more conservative, favoring downhill moves and refining the current solution.

The algorithm proceeds as follows:

1. Initialize the current solution x and set an initial temperature T.
2. Repeat until a stopping criterion is met (e.g., a maximum number of iterations or a minimum temperature):
    - Generate a neighbor solution x' by making a small change to x.
    - Calculate the change in energy $\Delta E = E(x') - E(x)$.
    - If $\Delta E < 0$ (i.e., x' is better), accept x' as the new current solution.
    - If $\Delta E \geq 0$, accept x' with a probability of $e^{-\Delta E/T}$.
    - Decrease the temperature T according to a cooling schedule.

The criterion $e^{-\Delta E/T}$ is called the **Metropolis criterion**. It ensures that as the temperature decreases, the probability of accepting worse solutions diminishes, allowing the algorithm to focus on refining the best solutions found. Usually, the cooling schedule is geometric, e.g., $T \leftarrow \alpha T$ with $\alpha \in (0, 1)$. A high value of α (e.g., 0.99) results in slow cooling, promoting exploration, while a low value (e.g., 0.8) leads to rapid cooling and quicker convergence.

Coding Example

We now implement a solution to the Traveling Salesman Problem (TSP) using Simulated Annealing.

```
import math
import random
import numpy as np

def total_distance(route, dist_matrix):
    """Calculates the total distance of the tour."""
    dist = 0
    for i in range(len(route) - 1):
        dist += dist_matrix[route[i]][route[i+1]]
    dist += dist_matrix[route[-1]][route[0]] # Return to start
    return dist
```

```
def simulated_annealing(dist_matrix, initial_temp=1000, cooling_rate=0.995):
    # 1. Initialization
    n_cities = len(dist_matrix)
    current_sol = list(range(n_cities))
    random.shuffle(current_sol)
    current_cost = total_distance(current_sol, dist_matrix)

    best_sol = list(current_sol)
    best_cost = current_cost

    temp = initial_temp

    # 2. The Annealing Loop
    while temp > 1:
        # Create Neighbor: Swap two random cities
        new_sol = list(current_sol)
        i, j = random.sample(range(n_cities), 2)
        new_sol[i], new_sol[j] = new_sol[j], new_sol[i]

        new_cost = total_distance(new_sol, dist_matrix)

        # Calculate Delta E
        delta_E = new_cost - current_cost

        # 3. Acceptance Criteria (Metropolis)
        # If better, accept. If worse, accept with probability exp(-delta/T)
        if delta_E < 0 or random.random() < math.exp(-delta_E / temp):
            current_sol = new_sol
            current_cost = new_cost

            # Update Global Best
            if current_cost < best_cost:
                best_sol = list(current_sol)
                best_cost = current_cost

        # 4. Cooling
        temp *= cooling_rate

    return best_sol, best_cost

# Example Usage
# Create a random distance matrix for 10 cities
```

```
dist_matrix = np.random.randint(10, 100, size=(10, 10))
np.fill_diagonal(dist_matrix, 0)

best_route, min_dist = simulated_annealing(dist_matrix)
print(f"Final Best Distance: {min_dist}")
```

```
Final Best Distance: 245
```

The code accepts a distance matrix representing the distances between cities and applies the Simulated Annealing algorithm to find a near-optimal tour. The algorithm starts with a random tour and iteratively improves it by exploring neighboring solutions, accepting worse solutions based on the Metropolis criterion, and gradually cooling down the temperature.

7.2.2 Tabu Search

While Simulated Annealing relies on rolling dice, Tabu Search (TS) relies on **memory**. It assumes that if we just visited a solution, we shouldn't go back to it immediately. To accomplish this, Tabu Search uses memory structures to guide the search.

The core idea of Tabu Search is to maintain a **tabu list**, which records recently visited solutions or moves. This list prevents the algorithm from revisiting these solutions for a certain number of iterations, thus encouraging exploration of new areas in the solution space. The tabu list can be implemented as a fixed-size queue that stores the last k moves or solutions.

The TS algorithm proceeds as follows:

1. Initialize the current solution x and the tabu list.
2. Repeat until a stopping criterion is met (e.g., a maximum number of iterations):
    - Generate a set of neighbor solutions $N(x)$.
    - Evaluate the neighbors and select the best one x' that is not in the tabu list (or satisfies an aspiration criterion). A neighbor can be accepted even if it's worse than the current solution, as long as it is not tabu.
    - Update the current solution to x'.
    - Update the tabu list by adding the move or solution to it, removing the oldest entry if necessary. It stays there for a specific duration (Tabu Tenure).

Coding Example

We will solve a generic problem: finding a binary string of length n that maximizes a value function, using Tabu Search. A "neighbor" is created by flipping exactly one bit.

```python
from collections import deque

def objective_function(x):
    # Example: Maximize the number of 1s (OneMax)
    # In reality, this would be a complex cost function
    return sum(x)

def get_neighbors(solution):
    neighbors = []
    for i in range(len(solution)):
        neighbor = list(solution)
        neighbor[i] = 1 - neighbor[i] # Flip bit i
        neighbors.append((neighbor, i)) # Store solution and the index flipped
    return neighbors

def tabu_search(n_bits, max_iter=100, tabu_tenure=5):
    # 1. Initialization
    current_sol = [random.randint(0, 1) for _ in range(n_bits)]
    best_sol = list(current_sol)
    best_score = objective_function(best_sol)

    # Tabu List: Stores indices of bits that were recently flipped
    tabu_list = deque(maxlen=tabu_tenure)

    for iteration in range(max_iter):
        neighbors = get_neighbors(current_sol)

        # Filter neighbors based on Tabu logic
        best_neighbor = None
        best_neighbor_score = -float('inf')
        move_index = -1

        for neighbor, idx in neighbors:
            score = objective_function(neighbor)

            is_tabu = idx in tabu_list
            is_aspiration = score > best_score # Aspiration Criteria

            # 2. Selection Logic: Allow if NOT Tabu OR meets Aspiration
            if (not is_tabu) or is_aspiration:
                if score > best_neighbor_score:
                    best_neighbor = neighbor
```

```
                    best_neighbor_score = score
                    move_index = idx

        # 3. Move and Update Memory
        if best_neighbor:
            current_sol = best_neighbor
            tabu_list.append(move_index) # Add the flipped bit to Tabu list

            # Update Global Best
            if best_neighbor_score > best_score:
                best_sol = list(best_neighbor)
                best_score = best_neighbor_score
                print(f"Iter {iteration}: New Best Score = {best_score}")

    return best_sol, best_score

# Example Usage
final_sol, final_score = tabu_search(n_bits=20, max_iter=50, tabu_tenure=3)
print(f"Final Solution: {final_sol}")
```

```
Iter 0: New Best Score = 11
Iter 1: New Best Score = 12
Iter 2: New Best Score = 13
Iter 3: New Best Score = 14
Iter 4: New Best Score = 15
Iter 5: New Best Score = 16
Iter 6: New Best Score = 17
Iter 7: New Best Score = 18
Iter 8: New Best Score = 19
Iter 9: New Best Score = 20
Final Solution: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The algorithm explores neighboring solutions by flipping bits, while maintaining a tabu list to avoid revisiting recently changed bits. The aspiration criterion allows the algorithm to accept tabu moves if they lead to a new global best solution.

In summary, SA is probabilistic and relies on randomness to escape local optima, while TS is deterministic and uses memory to guide the search. Both methods effectively balance exploration and exploitation, making them powerful tools for solving complex optimization problems. In the next section, we will explore population-based metaheuristics, which maintain a diverse set of solutions to explore the solution space more broadly.

7.3 Population-Based Metaheuristics

In trajectory-based methods (SA, TS), we send a single agent moving through the solution space. While effective, this approach has a limitation: the agent is lonely. It cannot learn from the successes or failures of other agents. Population-based metaheuristics address this limitation by maintaining a **population** of candidate solutions. Multiple agents explore the solution space simultaneously, sharing information and learning from each other. For instance, if Solution A has a good parameter for the first half of the problem, and Solution B has a good parameter for the second half, population methods attempt to combine them to create a superior Solution C. This collective behavior often leads to more robust exploration and better overall solutions. The most famous family of these algorithms is **Evolutionary Computation**, of which **Genetic Algorithms** (GAs) are the most well-known example.

7.3.1 Genetic Algorithms

Genetic Algorithms (GAs) are inspired by the principles of natural selection and genetics. They simulate the process of evolution by maintaining a population of candidate solutions (individuals) that evolve over generations. Each individual is typically represented as a chromosome, often encoded as a binary string, but other representations are also possible.

In any GA, we find the following key components:

- **Individuals:** Candidate solutions (e.g. a vector of weights). This would correspond to the *phenotype* in biology.
- **Chromosomes:** The genetic representation of individuals (e.g. binary strings). This corresponds to the *genotype* of the individual.
- **Gene:** A single unit of information in a chromosome (e.g. a bit in a binary string).
- **Fitness Function:** A function that evaluates how good an individual is with respect to the optimization objective.
- **Generation:** A single iteration of the algorithm, where the population is updated.

The GA algorithm proceeds in a loop where four biological-inspired operations are performed:

1. **Selection:** Individuals are selected from the current population to serve as parents for the next generation. Selection is typically based on fitness, with fitter individuals having a higher probability of being chosen.
2. **Crossover (Recombination):** Pairs of parents are combined to produce offspring. This is done by exchanging segments of their chromosomes, simulating genetic recombination.
3. **Mutation:** Random changes are introduced to the offspring's chromosomes to maintain genetic diversity and explore new areas of the solution space.
4. **Replacement:** The new generation of individuals replaces the old population, often keeping some of the best individuals (elitism) to ensure that good solutions are not lost.

In the following, we will dive deeper into each of these components and provide a coding example of a Genetic Algorithm applied to a simple optimization problem.

Selection Methods

Several selection methods exist, each with its own advantages and disadvantages:

- **Roulette Wheel Selection:** Individuals are selected with a probability proportional to their fitness. This method can be biased towards very fit individuals, potentially reducing diversity.
- **Tournament Selection:** A subset of individuals is randomly chosen, and the fittest among them is selected as a parent. This method maintains diversity while still favoring fitter individuals.
- **Rank Selection:** Individuals are ranked based on fitness, and selection probabilities are assigned based on rank rather than absolute fitness. This helps to prevent premature convergence.

The following example shows an implementation of tournament selection.

```
import numpy as np

def select_parents(population, fitnesses):
    """Tournament Selection: Pick 3, return the best."""
    parents = []
    for _ in range(len(population)):
        # Randomly sample 3 indices
        indices = np.random.randint(0, len(population), 3)
        # Find the one with the highest fitness
        best_idx = indices[np.argmax(fitnesses[indices])]
        parents.append(population[best_idx])
    return np.array(parents)
```

Crossover Methods

Crossover methods vary in how they combine the genetic material of the parents. Common methods include:

- **One-point Crossover:** A single crossover point is selected, and the offspring are created by exchanging the segments after this point.
- **Two-point Crossover:** Two crossover points are selected, and the segments between these points are exchanged.
- **Uniform Crossover:** Each gene is independently swapped with a certain probability.

The following example demonstrates one-point crossover.

```
def crossover(parent1, parent2):
    """Single-Point Crossover."""
    if np.random.rand() < 0.8: # 80% chance to mate
        point = np.random.randint(1, len(parent1))
        child1 = np.concatenate((parent1[:point], parent2[point:]))
        child2 = np.concatenate((parent2[:point], parent1[point:]))
        return child1, child2
    return parent1.copy(), parent2.copy()
```

Mutation Methods

Mutation introduces random changes to the offspring's chromosomes to maintain diversity. Common mutation methods include: - **Bit Flip Mutation:** For binary strings, randomly flip bits with a certain probability. - **Gaussian Mutation:** For real-valued representations, add a small random value drawn from a Gaussian distribution to the genes.

The following example shows bit flip mutation.

```
def mutate(individual):
    """Gaussian Mutation: Add small noise."""
    for i in range(len(individual)):
        if np.random.rand() < MUTATION_RATE:
            # Add random noise
            individual[i] += np.random.normal(0, 0.5)
    return individual
```

Putting It All Together

Now we can implement a simple Genetic Algorithm that combines these components to optimize a function. For demonstration purposes, we will implement a GA to minimize the Sphere Function defined as $f(x) = \sum_{i=1}^{n} x_i^2$, which has a global minimum at $x = 0$. While GAs were originally designed for binary strings, they are heavily used today for continuous optimization (e.g., tuning hyperparameters).

```
import numpy as np

# --- Configuration ---
POP_SIZE = 50        # Number of individuals
GENES = 10           # Number of variables (dimensions)
GENERATIONS = 100    # How long to run
MUTATION_RATE = 0.1  # Probability of mutation per gene
ELITISM_COUNT = 2    # Number of top solutions to keep
```

```python
def fitness_function(individual):
    """Objective: Minimize sum of squares (Sphere function)."""
    # We negate because GAs typically maximize fitness
    return -np.sum(individual**2)

def create_population(size, n_genes):
    """Initialize random real-valued population between -5.12 and 5.12"""
    return np.random.uniform(-5.12, 5.12, (size, n_genes))

# --- Main GA Loop ---
population = create_population(POP_SIZE, GENES)

for gen in range(GENERATIONS):
    # 1. Evaluate Fitness
    fitnesses = np.array([fitness_function(ind) for ind in population])

    # Track stats
    best_idx = np.argmax(fitnesses)
    if gen % 10 == 0:
        print(f"Gen {gen}: Best Fitness = {fitnesses[best_idx]:.5f}")

    # 2. Elitism: Keep the best
    # Sort indices by fitness (descending)
    sorted_indices = np.argsort(fitnesses)[::-1]
    elites = population[sorted_indices[:ELITISM_COUNT]]

    # 3. Selection
    parents = select_parents(population, fitnesses)

    # 4. Crossover & Mutation
    next_population = []
    # Add elites first
    next_population.extend(elites)

    # Fill the rest
    for i in range(0, POP_SIZE - ELITISM_COUNT, 2):
        p1, p2 = parents[i], parents[i+1]
        c1, c2 = crossover(p1, p2)
        next_population.append(mutate(c1))
        if len(next_population) < POP_SIZE:
            next_population.append(mutate(c2))
```

```
    population = np.array(next_population)

# Final Result
final_fitnesses = np.array([fitness_function(ind) for ind in population])
best_sol = population[np.argmax(final_fitnesses)]
print(f"Final Solution: {best_sol}")
```

```
Gen 0: Best Fitness = -40.83219
Gen 10: Best Fitness = -2.73949
Gen 20: Best Fitness = -0.43545
Gen 30: Best Fitness = -0.05074
Gen 40: Best Fitness = -0.03186
Gen 50: Best Fitness = -0.01999
Gen 60: Best Fitness = -0.00832
Gen 70: Best Fitness = -0.00814
Gen 80: Best Fitness = -0.00549
Gen 90: Best Fitness = -0.00461
Final Solution: [ 0.0007059   0.02276293 -0.0064019   0.01909445  0.0154404   0.01028623
  0.00133111 -0.01388762  0.04760251  0.00116085]
```

In this implementation, we initialize a population of random solutions and iteratively apply selection, crossover, and mutation to evolve the population towards better solutions. We also use elitism to ensure that the best solutions are preserved across generations. The algorithm optimizes the Sphere function, and we track the best fitness at regular intervals.

The power of GA lies in its ability to explore a large and complex solution space effectively, making it suitable for a wide range of optimization problems, from engineering design to machine learning hyperparameter tuning. In general, GAs are easily parallelizable, as the evaluation of fitness for different individuals can be done independently, making them well-suited for modern computational architectures. Additionally, they do not require gradients, making them applicable to non-differentiable or noisy optimization problems. Finally, the combination of crossover and mutation allows GAs to maintain diversity in the population, helping to avoid premature convergence to local optima.

7.4 Theoretical Considerations

We have now explored different metaheuristic algorithms, each with its own strengths and weaknesses. While these methods are powerful, it is important to understand their theoretical underpinnings and limitations. If we are building a routing system for a logistics company, should we default to a Genetic Algorithm because it sounds the most sophisticated? Or stick to Simulated Annealing because it is easier to code?

7.4.1 The No Free Lunch Theorems

While answering this question is complex, part of the answer lies in the **No Free Lunch Theorems** for optimization. These theorems state that no optimization algorithm is universally superior to others when averaged over all possible problems. In other words, an algorithm that performs well on one class of problems may perform poorly on another. Therefore, the choice of metaheuristic should be informed by the specific characteristics of the problem at hand.

We can derive the following implications from the No Free Lunch Theorems:

- There is no "super algorithm" that is best for all optimization problems. The effectiveness of an algorithm depends on the structure of the problem.
- Performance comes from specialization: An algorithm that is tailored to exploit the specific features of a problem will outperform a general-purpose algorithm on that problem.
- Domain knowledge is crucial: Understanding the problem domain can guide the design of heuristics and metaheuristics that are more effective for that particular problem.

Specifically, the job of the computer scientist is not just to pick an algorithm from a library, but to understand the *geometry* of their specific problem. For instance, if the problem has a "smooth" landscape, we can use a trajectory method like SA. However, if it has a "modular" structure instead (where combining parts of solution A and B makes sense), probably using a Genetic Algorithm would be the best choice.

7.4.2 Hyperparameters and Tuning

The second theoretical hurdle is the issue of configuration. Exact algorithms like Simplex have very few parameters. Metaheuristics, however, are full of "knobs" that must be tuned. For instance, in SA we have the initial temperature, cooling rate, and stopping criteria. In GAs, we have population size, mutation rate, crossover rate, selection method, and more. The performance of these algorithms can be highly sensitive to the choice of these hyperparameters.

So how do we choose e.g. the correct mutation rate? Ideally, we want the mutation rate that minimizes our loss function. But finding that rate is itself an optimization problem. We are effectively trying to "optimize the optimizer". This meta-optimization can be computationally expensive and may require specialized techniques.

For instance, one of the most common methods for hyperparameter tuning is **Grid Search**, where we define a discrete set of values for each hyperparameter and evaluate the performance of the algorithm for every combination. While this method is straightforward, it can be computationally expensive, especially when the number of hyperparameters and their possible values increases.

Additionally, there are more sophisticated methods like **Random Search**, which samples hyperparameter combinations randomly, and **Bayesian Optimization**, which builds a probabilistic

model of the objective function and uses it to select promising hyperparameter values. These methods can be more efficient than Grid Search, especially in high-dimensional hyperparameter spaces.

7.5 Summary

In this chapter, we explored the world of metaheuristics, which are powerful tools for solving complex optimization problems that are intractable for exact methods. We discussed the fundamental concepts of heuristics and metaheuristics, and how they differ from traditional optimization algorithms. We delved into trajectory-based methods like Simulated Annealing and Tabu Search, which use randomness and memory to escape local optima. We also examined population-based methods like Genetic Algorithms, which maintain a diverse set of solutions to explore the solution space more effectively.

Finally, we discussed theoretical considerations such as the No Free Lunch Theorems, which highlight the importance of problem-specific algorithm design, and the challenges of hyperparameter tuning in metaheuristics. Understanding these concepts is crucial for effectively applying metaheuristic algorithms to real-world optimization problems.

7.6 Exercises

1. Implement a Simulated Annealing algorithm to solve the Knapsack Problem. Compare its performance with a simple greedy heuristic.
2. Implement a Tabu Search algorithm for the Job Scheduling Problem. In the Job Scheduling Problem, we have a set of jobs, each with a processing time and a deadline. The goal is to schedule the jobs in a way that minimizes the total tardiness (the amount of time by which jobs miss their deadlines). Experiment with different tabu tenures and aspiration criteria to see how they affect the performance of the algorithm.
3. Implement a Genetic Algorithm to optimize the Rastrigin function, which is a non-convex function used as a performance test problem for optimization algorithms. The Rastrigin function is defined as:

$$f(x) = 10n + \sum_{i=1}^{n} [x_i^2 - 10\cos(2\pi x_i)]$$

   where n is the number of dimensions and x_i are the input variables. Experiment with different selection methods, crossover rates, and mutation rates to find the best configuration for optimizing this function.

4. Explore the No Free Lunch Theorems by implementing a simple optimization problem (e.g., maximizing a function) and comparing the performance of different metaheuristic algorithms (e.g., Simulated Annealing, Tabu Search, Genetic Algorithm) on this problem. Analyze how the structure of the problem affects the performance of each algorithm.

8 Optimization and Simulation in Machine Learning

8.1 Introduction

In the previous chapters, we treated optimization as a tool for decision-making, like finding the best shipping route or the optimal factory schedule. In these scenarios, the "model" was explicit. We knew the constraints, we knew the objective function, and we simply needed to find the variable values x that optimized the objective function. In Machine Learning (ML) and Artificial Intelligence (AI), the change the paradigm. We are no longer optimizing a specific decision, but rather *learning a function* that makes decisions.

From this perspective, ML is not magic, but simply optimization applied to function approximation. When we say a neural network "learns" to recognize a cat, we mean that an optimization algorithm has traversed a high-dimensional landscape and found a specific configuration of parameters (weights and biases) that minimizes the error between the network's output and the label "cat."

In general, we use a very specific notation when applying optimization to ML problems. We denote the input data as X and the output data (or labels) as y. The model we are trying to learn is a function $f(X;\theta)$, where θ represents the parameters of the model (e.g., weights in a neural network). The goal of training the model is to find the optimal parameters θ^* that minimize a loss function $\mathcal{L}(y, f(X;\theta))$, which quantifies the difference between the predicted outputs and the true labels.

$$\theta^* = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(y_i, f(x_i;\theta))$$

However, not all problems come with a static dataset. In Reinforcement Learning (RL) or game playing (like Go or Chess), the "data" is generated dynamically by the agent's interactions with the world. In this case, we often do not have an explicit formula for the objective function (e.g., "Win the game") and we cannot calculate the gradient of "winning" directly. Instead, we must rely on **simulation**: We run the system forward in time (simulate a game, simulate a robot arm moving), observe the outcome, and use that empirical data to estimate the gradient or value.

Thus, modern AI is the marriage of **optimization** (improving parameters based on data) and **simulation** (generating data based on parameters).

In this chapter, we will explore how these concepts manifest in three distinct phases of the AI lifecycle:

1. **Training:** The process of optimizing model parameters using data samples. In this case, we rely on Stochastic Gradient Descent (SGD) and its adaptive variants (like Adam). These methods use the "noise" inherent in data sampling to navigate complex terrains where traditional Newton-type methods fail.
2. **Planning:** The process of simulating future states of the world to make better decisions in the present. Here, we will explore techniques like Monte Carlo Tree Search (MCTS) that allow agents to evaluate potential future actions by simulating their outcomes.
3. **Interaction:** The process of learning from real-time feedback as the agent interacts with its environment. This involves methods like Q-learning and Policy Gradients, which use simulations of the agent's actions to improve its decision-making policy over time.

8.2 Optimization in Deep Learning

In Chapter 5, we established that convex problems are "easy" and non-convex problems are "hard." Deep Learning places us firmly in the "hard" category. A modern neural network may have billions of parameters, and the relationship between these parameters and the loss function is highly non-linear. Despite this, we routinely train these models to near-perfect accuracy. How is this possible? The answer lies in the specific geometry of high-dimensional spaces and the specific properties of the algorithms we use.

8.2.1 The Loss Landscape in DL

For decades, researchers and practitioners in ML feared local minima: small valleys where the algorithm might get stuck, far higher than the global minimum. However, intuition derived from 2D or 3D surfaces is misleading in high dimensions. In 2D, a local minimum is a point where all directions lead uphill. In high dimensions, however, the number of directions increases dramatically. As a result, the probability of encountering a true local minimum (where all directions are uphill) decreases significantly. Instead, we often find "saddle points" or "flat regions" where some directions lead uphill while others lead downhill.

Probability theory tells us this is incredibly rare. In high-dimensional landscapes, most points where the gradient vanishes ($\nabla \mathcal{L} = 0$) are not minima, but saddle points. These are points that curve up in some directions but down in others. The main implication is that algorithms that rely solely on Newton's method (using the Hessian) struggle here because the Hessian has

both positive and negative eigenvalues. Gradient-based methods, however, can usually "slide off" the saddle point along the downward-sloping dimensions.

8.2.2 Backpropagation and Stochastic Gradient Descent

In general, to minimize the loss $\mathcal{L}(\theta)$, we need the gradient $\nabla\mathcal{L}(\theta)$. For a deep network, deriving this analytically is impossible. Instead, we use **backpropagation**. Put simply, backpropagation is simply the recursive application of the familiar chain rule of calculus on a computational graph. It involves two passes through the network: a **forward pass** to compute the output and loss, and a **backward pass** to compute the gradients of the loss with respect to each parameter. Modern Deep Learning frameworks like PyTorch utilize Automatic Differentiation (AutoDiff): only the forward pass is needed, and the software automatically constructs the graph to compute the backward pass.

Standard Gradient Descent (Batch GD) computes the gradient over the entire dataset before taking a step.

$$\theta_{k+1} = \theta_k - \eta\frac{1}{n}\sum_{i=1}^{n}\nabla\mathcal{L}_i(\theta_k)$$

where $\mathcal{L}_i$ is the loss for the i-th data point, and η is the learning rate. However, for large datasets, this is computationally expensive. Instead, we use **Stochastic Gradient Descent (SGD)**, which approximates the gradient using a small batch of data points.

$$\theta_{k+1} = \theta_k - \eta\frac{1}{m}\sum_{i=1}^{m}\nabla\mathcal{L}_{j_i}(\theta_k)$$

where m is the batch size, and j_i are randomly selected indices from the dataset. This introduces noise into the gradient estimate, which can help the optimization process escape saddle points and explore the loss landscape more effectively.

The landscape of a neural network often features *ravines*: areas that are steep in one dimension but flat in another. Standard SGD tends to oscillate wildly across the steep slopes while making slow progress along the flat bottom. To fix this, we use algorithms that adapt the update step for each parameter individually.

Momentum

Momentum is a technique that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by maintaining a velocity vector that accumulates the gradients over time.

$$v_{k+1} = \beta v_k + (1-\beta)\nabla\mathcal{L}(\theta_k)$$
$$\theta_{k+1} = \theta_k - \eta v_{k+1}$$

Where β is a new hyperparameter that we call momentum and takes values between 0 and 1. In practice, this parameter is usually set to around 0.9.

Adam

The other idea is called *Adaptive Moment Estimation* (Adam). Adam maintains two moving averages: one for the gradients (first moment) and one for the squared gradients (second moment).

$$m_{k+1} = \beta_1 m_k + (1 - \beta_1)\nabla\mathcal{L}(\theta_k)$$
$$v_{k+1} = \beta_2 v_k + (1 - \beta_2)(\nabla\mathcal{L}(\theta_k))^2$$

$$\theta_{k+1} = \theta_k - \eta\frac{m_{k+1}}{\sqrt{v_{k+1}} + \epsilon}$$

Where β_1 and β_2 are hyperparameters that control the decay rates of these moving averages, typically set to 0.9 and 0.999, respectively. The small constant ϵ is added to prevent division by zero.

Let's take a look at a scratch implementation of Vanilla SGD and Adam in Python:

```
import numpy as np
import matplotlib.pyplot as plt

# 1. Define a "Ravine" Function: f(x,y) = 0.1x^2 + 2y^2
# It is much steeper in Y than in X.
def loss_function(x, y):
    return 0.1 * x**2 + 2 * y**2

def gradient(x, y):
    return np.array([0.2 * x, 4 * y])

# 2. Vanilla SGD Implementation
def run_sgd(start_pos, lr, steps):
    path = [start_pos]
    curr = np.array(start_pos)
    for _ in range(steps):
        grad = gradient(curr[0], curr[1])
        curr = curr - lr * grad
        path.append(curr)
    return np.array(path)
```

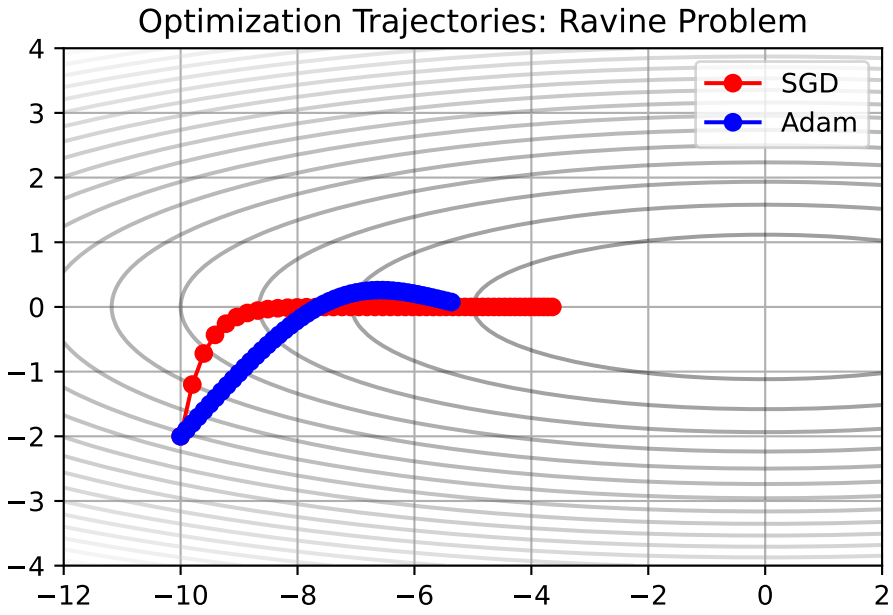
Now we implement Adam:

```
# 3. Adam Implementation (Simplified)
def run_adam(start_pos, lr, steps, beta1=0.9, beta2=0.999, epsilon=1e-8):
    path = [start_pos]
    curr = np.array(start_pos)
    m = np.zeros(2) # First moment (Momentum)
    v = np.zeros(2) # Second moment (Variance)

    for t in range(1, steps + 1):
        grad = gradient(curr[0], curr[1])

        # Update moments
        m = beta1 * m + (1 - beta1) * grad
        v = beta2 * v + (1 - beta2) * (grad**2)

        # Bias correction (important for early steps)
        m_hat = m / (1 - beta1**t)
        v_hat = v / (1 - beta2**t)

        # Update parameters
        curr = curr - lr * m_hat / (np.sqrt(v_hat) + epsilon)
        path.append(curr)
    return np.array(path)
```

In this implementation, we have added a *bias correction* step to the moment estimates. This is crucial, especially in the early stages of training, to ensure that the estimates are unbiased. Now we visualize the results:

```
start = [-10.0, -2.0] # Start far away
steps = 50
lr = 0.1

path_sgd = run_sgd(start, lr, steps)
path_adam = run_adam(start, lr, steps)

# Create contour map
X, Y = np.meshgrid(np.linspace(-12, 2, 100), np.linspace(-4, 4, 100))
Z = loss_function(X, Y)
plt.contour(X, Y, Z, levels=20, cmap='gray', alpha=0.4)

plt.plot(path_sgd[:,0], path_sgd[:,1], 'o-', label='SGD', color='red')
plt.plot(path_adam[:,0], path_adam[:,1], 'o-', label='Adam', color='blue')
```

```
plt.title("Optimization Trajectories: Ravine Problem")
plt.legend()
plt.grid(True)
plt.show()
```



In this graph, SGD oscillates vertically (along the steep Y-axis) while making very slow progress horizontally (along the flat X-axis). It struggles to find the right step size for both dimensions simultaneously. In contrast, Adam quickly adapts. It dampens the step size in Y (high gradient) and boosts the step size in X (low gradient), shooting directly toward the center minimum. This behavior is why Adam is the standard for training deep networks.

8.3 Monte Carlo Tree Search

In the previous section, we discussed optimizing a static function (the loss of a neural network). Now, we turn to planning: deciding what action to take in a complex, sequential environment, such as playing chess, routing a fleet of drones, or folding a protein. Historically, AI planning relied on Minimax search with heuristic evaluation functions. To know if a chess board state was "good," a human expert had to write a scoring function (e.g. $10\times$ if queens are on the board + $5\times$ number of rooks, etc). However, for complex situations like playing a game of Go or folding proteins, writing a good evaluation function gets too challenging.

This is where Monte Carlo simulation comes in. Instead of asking an expert "Is this state good?", we ask the computer to play it out. We simulate thousands of random games starting

from the current state. If we win 80% of those random games, the state is likely "good". If we win only 10%, it is likely "bad".

In the planning context, we call this a **rollout**:

1. Start at initial state S_t.
2. Select actions $a_t, a_{t+1}, ...$ until a terminal state (Win/Loss) is reached.
3. Record the result: $+1$ if win, 0 if loss.
4. Repeat N times.
5. Estimate the value of S_t as Total Wins/N.

However, classical Monte Carlo Simulation has a flaw: it wastes computational resources exploring obviously bad moves. If you hang your queen in chess, you don't need 1,000 simulations to know it's a bad idea; you need 1.

Monte Carlo Tree Search (MCTS) solves this by building a search tree *asymmetrically*. It uses the results of previous simulations to guide future simulations toward more promising parts of the tree, trying to balance exploitation (by focusing on moves with a high win rate) and exploration (by simulting moves that have few visits, just in case they turn out to be good).

MCTS runs a loop of four steps as many times as the computing budget allows:

1. **Selection:** We start at the root and traverse down the tree by selecting child nodes. At each step, we choose the child that maximizes the **Upper Confidence Bound (UCT)** formula (see Equation 8.1), stopping when a node that has not been fully expanded is reached.
2. **Expansion:** Add one or more child nodes to the tree (representing available moves from that state).
3. **Simulation:** From the new child node, play a random game to the end (rollout).
4. **Backpropagation:** Take the result of the simulation (Win/Loss) and walk back up the tree to the root, updating the statistics (wins and visits) for every node on the path.

The UCB formula is given by:

$$UCT_i = \frac{w_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}} \tag{8.1}$$

where w_i denotes the number of wins for child node i, n_i is the number of times that node i has been visited, N is the total number of visits to the parent node, and c is an exploration constant (typically $c = \sqrt{2}$).

We can interpret w_i/n_i as the average win rate, which encourages exploitation. The other term ($\sqrt{...}$) is an uncertainty "bonus": if a node is rarely visited (n_i is small), this term becomes

larger, which encourages exploration. As we proceed and visit more nodes, n_i becomes larger and the search concentrates more on the exploitation term.

MCTS allows an agent to plan in environments where it has no prior knowledge. It does not need to know strategy; it discovers strategy by simulating thousands of futures. E.g. in *AlphaZero*, this simulation data is used to train a Neural Network, which in turn guides the MCTS Selection phase, creating a cycle of self-improvement.

We now take a look at a sample generic implementation of MCTS. We start by defining a node in the search tree.

```
import math
import random
import copy

class MCTSNode:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent
        self.children = []
        self.wins = 0
        self.visits = 0
        self.untried_moves = state.get_legal_moves()

    def uct_select_child(self, c=1.41):
        # Sort children by UCT formula
        s = sorted(self.children, key=lambda child:
            (child.wins / child.visits) + c * math.sqrt(math.log(self.visits) / child.visits)
        return s[-1] # Return the best
```

An MCTS node tracks the number of wins, visits, the problem-dependent state, and the number of untried moves. Additionally, is provides a method for selectin the best child according to Equation 8.1.

The main MCTS algorithm with the four phases mentioned before can be implemented as follows:

```
def run_mcts(root_state, iterations=1000):
    root_node = MCTSNode(root_state)

    for _ in range(iterations):
        node = root_node
        state = copy.deepcopy(root_state)
```

```
        # Phase 1: Selection
        # Go down the tree until we hit a node with untried moves or a terminal state
        while node.untried_moves == [] and node.children != []:
            node = node.uct_select_child()
            state = state.make_move(node.move_from_parent) # Need to track moves in real impl

        # Phase 2: Expansion
        # If we can expand, add one child
        if node.untried_moves != []:
            m = random.choice(node.untried_moves)
            state = state.make_move(m)
            node = node.add_child(m, state) # Pseudo-code: create child logic

        # Phase 3: Simulation (Rollout)
        # Play randomly until end
        while state.check_winner() is None:
            state = state.make_move(random.choice(state.get_legal_moves()))

        # Phase 4: Backpropagation
        # Update stats up the tree
        winner = state.check_winner()
        while node is not None:
            node.visits += 1
            # If the player at this node won, increment wins
            # (Note: In real MCTS, we must be careful about whose turn it is)
            if winner == node.state.player_who_just_moved:
                node.wins += 1
            node = node.parent

    # Return the move with the most visits (most robust)
    return sorted(root_node.children, key=lambda c: c.visits)[-1].move_from_parent
```

The `make_move` and `check_winner` methods would need to be implemented for the specific problem at hand.

8.4 Reinforcement Learning

We now very briefly introduce Reinforcement Learning and frame it as an optimization method. In Supervised Learning, the optimization target is static (minimize the training error on a fixed dataset). In constrast, in **Reinforcement Learning (RL)**, the target is dynamic. We

are optimizing a sequence of decisions over time, where a decision made now determines the data we see later. The mathematical foundation of RL is based on the concept of a *Markov Decision Process (MDP)*, as we will see next.

8.4.1 Markov Decision Processes

The mathematical formalism of Markov Decision Processes underpins how optimization in RL is framed. We start with the following elements:

- **State Space** S**:** The set of all possible configurations (e.g. in chess, the set of all legal chess board states).
- **Action Space** A **:** The set of all possible actions that an agent can take (e.g. the set of all legal moves in a game).
- **Transition Probabilities** P**:** The rules for transitioning from one state to another. In general, this is represented by a probability distribution $P(s'|s, a)$ of states conditional on the previous state s and the action taken a.
- **Reward Function** R**:** The immediate reward $R(s, a)$ the agent collects on state s after performing action a.
- **Discount Factor** γ**:** A number between 0 and 1 that weighs immediate rewards against future rewards.

Our goal is to find a **policy** $\pi(a|s)$ that maps states to actions that maximizes the expected discounted return:

$$J(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

This can be framed as a constrained optimization problem: maximize J subject to the constraints of the environment dynamics.

8.4.2 Value-based Methods

One approach to solving this is to ignore the policy initially and focus on maximizing a notion of *value*. If we knew exactly how good every state was (the "Value Function"), the optimal policy would simply be "move to the state with the highest value". We can define the value of a state-action pair $Q(s, a)$ recursively as follows (**Bellman equation**):

$$Q(s, a) = R(s, a) + \gamma \max_{a'} Q(s', a')$$

which means that the value of an action is the *immediate reward* plus the *best possible future value*.

We can approximate the true $Q(s,a)$ using a deep neural network $Q(s,a;\theta)$, in what we call **Deep Q-Learning (DQN)**. We can train this network by minimizing the mean squared error (also called Bellman error, or Temporal Differences error):

$$\mathcal{L}(\theta) = E\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{target}) - Q(s,a;\theta)\right)^2\right] \tag{8.2}$$

In this function, we measure the difference between the prediction from the neural network $Q(s,a;\theta)$ and the current estimate of the target term:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta_{target})$$

In this target, r is the reward which was *actually experienced* by the agent. However, the second term is just the current best estimate on how much reward it is expected in the future. In summary, we are using the network's own prediction of the future to train its prediction of the present. This is sometimes referred to as **bootstrapping**.

This method is also referred to as an instance of **model-free** RL. In this case, we do not have a model of how the physical world works, but we can simulate several actions of the agent to obtain tuples $(s, a, r, s', \text{done})$ that we store in a so-called *replay buffer*. Once we have this buffer, we can feed these data on a training episode and use Equation 8.2 to update the weights θ of the neural network. Note that in this equation, we have two sets of weights θ_{target} and θ. The weights which are currently being optimized are θ, while θ_{target} is a frozen copy used for calculating the max term. This is done in order to prevent the training process from "chasing its own tail", as otherwise the network predictions would change with each gradient update.

8.4.3 Policy-Based Methods

Alternatively, we can try to optimize the policy $\pi_\theta(a|s)$ directly instead of the value function $Q(s,a)$. However, because the environment is usually a "black-box" or a discrete simulation, we cannot calculate the gradient $\nabla_\theta J(\theta)$ directly.

$$\nabla_\theta J(\theta) = \nabla_\theta E_{\tau\sim\pi}\left[R(\tau)\right] = \nabla_\theta \int P(\tau|\theta) R(\tau) d\tau$$

The main difficulty with this calculation is that we cannot differentiate this integral to obtain the gradient. However, we can use a property of the derivative of a logarithm (the *log-derivative trick*) to transform it into the integral of a derivative instead.

$$\frac{d}{dx} \ln f(x) = \frac{f'(x)}{f(x)} \Rightarrow f'(x) = f(x) \frac{d}{dx} \ln f(x)$$

$$\nabla_\theta P(\tau|\theta) = P(\tau|\theta) \cdot \nabla_\theta \ln P(\tau|\theta)$$

which converts the previous integral in:

$$\nabla_\theta J(\theta) = \int P(\tau|\theta) \nabla_\theta \ln P(\tau|\theta) \cdot R(\tau) d\tau = E_{\tau\sim\pi} \left[\nabla_\theta \ln P(\theta|\phi) \cdot R(\theta)\right]$$

Using the estimates that we have, we can approximate the gradient of the expected reward as follows:

$$\nabla_\theta J(\theta) \approx E\left[\sum_{t=0}^{T} \nabla_\theta \ln \pi_\theta(a_t|s_t) \cdot G_t\right]$$

where:

- G_t is the total return obtained from time t onwards.

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...$$

- $\nabla \ln \pi(a|s)$ tells us how to change θ to make action a more or less probable.

In other words, if G_t is positive (good outcome), we move θ to *increase* the probability of the action. Otherwise, if G_t is negative (bad outcome), we move θ to *decrease* that probability. This is known as the **REINFORCE** algorithm, and it's another instance of model-free RL.

8.4.4 Model-Based Reinforcement Learning

The methods mentioned above are model-free: they learn purely from experience without having any knowledge about how those experiences are formed. By contrast, in **model-based RL** our goal is to learn a model of the world, i.e. the transition function $P(s'|s, a)$. Once that the agent has learnt this model, we can use techniques like DQL more efficiently by, for instance, simulating training data and using real and synthetic data to train the agent.

8.5 Case Study: AlphaZero

In 2017, DeepMind introduced AlphaZero, a system that mastered the games of Chess, Shogi, and Go. Unlike its predecessor AlphaGo (which learned from human games) or Deep Blue (which relied on handcrafted heuristics), AlphaZero started from scratch. It knew only the rules of the game. As we will see next, AlphaZero as a case study unifies the themes outlined in this chapter: simulation, function approximation, and optimization.

At the heart of AlphaZero is a single Deep Residual Network (ResNet), parameterized by weights θ, which takes board data s as input. Unlike traditional engines that have separate modules for strategy and evaluation, AlphaZero uses a single network with two output "heads":

1. **The Policy Head:** Outputs a probability distribution $p(a|s)$ over all legal moves. This represents the "intuition" of the agent.
2. **The Value Head:** Outputs a single scalar $v(s) \in [-1, 1]$ as the expected outcome from state s (+1 for win, -1 for loss). This represents the "judgement" of the agent.

Now the critical insight of AlphaZero is how it uses MCTS. In traditional AI, search is used only at runtime to play the game. In AlphaZero, search is used during *training* to *generate* the data. AlphaZero uses a neural network to guide the MCTS selection process. For this, we need a modified UCT formula to include the network's prior probability $P(s, a)$:

$$UCT(s, a) = Q(s, a) + c \cdot P(s, a) \cdot \frac{\sqrt{N(s)}}{1 + N(s, a)} \tag{8.3}$$

where:

- $Q(s, a)$ is the average value of all simulations that passed through this specific state-action pair. It is calculated as the average of the Value Head outputs v of the leaf nodes reached by this branch. Its role is *exploitation* by representing the empirical evidence gathered so far: "Does this move actually lead to a win?".
- $P(s, a)$ is the probability assigned to action a by the Policy Head when it first evaluated state s. It represents the agent's intuition: "Does this move look like a good move?".
- $N(s)$ is the parent visit count, or total number of times the parent node (state s) has been visited.
- $N(s, a)$ is the child visit count, or number of times a specific action a has been selected from state s. Since this appears in the denominator, as we visit a specific move more often, this term grows, reducing the influence of the exploration bonus.
- c is an exploration constant, a hyperparameter that determines how much we rely on the prior $P(s, a)$ versus the empirical value $Q(s, a)$.

After running a large number of simulations for a single move, the MCTS produces a visit count distribution. We normalize these visit counts to create a new probability distribution π. This distribution represents a stronger policy than the raw network output $p_\theta(s)$, since the search process has "purified" the intuition (the prior).

8.5.1 The Training Process

The training process is a continuous loop of self-improvement.

1. **Self-Play (Simulation):** The agent plays games against itself. At every step, it runs MCTS to generate the improved policy π and selects a move. It plays until the game ends, producing a final outcome $z \in \{-1, +1\}$.
2. **Data Generation:** For every position in the game, we store a training example (s, π, z), where s represents the board state, π is what MCTS said was the best action and z represents the actual result (who won the game).
3. **Optimization:** We optimize the network parameters θ to minimize the error between its predictions and the MCTS data.

$$\mathcal{L}(\theta) = \underbrace{(z - v_\theta(s))^2}_{\text{Value loss}} - \underbrace{\pi^T \log p_\theta(s)}_{\text{Policy loss}} + \underbrace{c\|\theta\|^2}_{\text{Regularization}}$$

where the *value loss* is the MSE of the network predictions regarding the actual winner z and the *policy loss* is the cross-entropy of the network trying to mimic the MCTS search probabilities (π).

8.5.2 Simulation becomes Intuition

AlphaZero then learns the following way: Initially, the network is random and MCTS relies heavily on random rollouts. However, over time MCTS starts discovering winning strategies and the optimization steps forces the neural network to learn those strategies. As the neural network improves, its predictions (p and v) become more accurate, which improves MCTS (since p guides the search) to look deeper and find more subtle strategies.

This is the essence of optimization and simulation in ML: We use computationally expensive simulation (MCTS) to generate high-quality ground truth data, and then use optimization (SGD) to distill that complex simulation into a fast, efficient function approximator (the neural network). The result is an agent that possesses the "instinct" of a grandmaster.

8.6 Summary

In this chapter, we crossed the threshold from classical optimization—where the goal is to find the best value for a variable—to Machine Learning, where the goal is to find the best function to approximate reality.

We began by framing Deep Learning as a high-dimensional, non-convex optimization problem. We saw that the primary obstacles in training neural networks are not local minima, but saddle points and ill-conditioned curvature (ravines). We explored how Stochastic Gradient Descent (SGD) utilizes the noise of mini-batches to escape these traps, and how adaptive methods like Adam approximate the diagonal of the Hessian to normalize step sizes, allowing for efficient training on rugged landscapes.

Next, we moved from static functions to dynamic planning using Monte Carlo Tree Search (MCTS). We learned that when an objective function is too complex to define analytically (like the strategy of Go), we can estimate it through Simulation. By balancing exploration and exploitation via the UCT formula, MCTS builds an asymmetric search tree that focuses computational resources on the most promising futures.

Finally, we unified these concepts under Reinforcement Learning (RL). We framed sequential decision-making as an optimization problem over time, solving it either by approximating the value function (Deep Q-Learning) or by optimizing the policy directly (REINFORCE). We concluded with the case study of AlphaZero, which demonstrated that state-of-the-art AI is not magic, but a virtuous cycle: Simulation (MCTS) generates data to train a Function Approximator (Neural Network), which in turn guides the simulation to be more efficient.

8.7 Exercises

1. Consider the 2D "Rosenbrock-like" function, which features a narrow, curved valley:

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

   Implement the gradient $\nabla f(x, y)$ as a Python function and then Vanilla SGD with a fixed learning rate of $\eta = 0.001$. Implement the Adam optimizer with $\eta = 0.1$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Start both optimizers at $(-1.5, -1.0)$ and run for 200 steps. Visualize the trajectory of both optimizers on a contour plot and explain why SGD oscillates or gets stuck, while Adam follows the valley floor.

2. Consider a node S in a MCTS game tree. The node has been visited $N = 50$ times and has two child nodes A and B. The statistics for the child nodes are as follows:

   - Child A: Wins = 12, Visits = 20.
   - Child B: Wins = 18, Visits = 30.

   The exploration constant is $c = \sqrt{2}$. Calculate the UCT value for the child nodes A and B and select the next node according to the result. Suppose we visit the parent S an additional 50 times ($N = 100$), but we never visit child A again. Calculate the new UCT value for A.

3. Assume an agent in a Gridworld at state $s = (1, 1)$. When it takes the action "Right", moving to state $s' = (1, 2)$, it receives a reward $r = -1$. Let the discount factor be $\gamma = 0.9$ and the learning rate $\alpha = 0.1$. The Q-values for the next state are:

   - $Q(s', \text{Up}) = -2.0$.
   - $Q(s', \text{Right}) = -1.0$.
   - $Q(s', \text{Down}) = -4.0$.
   - $Q(s', \text{Left}) = -3.0$.

Calculate the temporal differences error and update the Q-value $Q((1,1), \text{Right})$ using the standard Q-Learning update rule.

4. In the context of AlphaZero, let (s, a) be state-move pair with a low probability $P(s, a)$. Imagine that this move is a forced checkmate sequence (win). Taking Equation 8.3 into account, explain how MCTS can eventually discover and select this move despite the network's initial bias against it. Which term in the formula drives this correction?

9 Summary

Simulation and optimization are not, as one might be tempted to think, mere satellite disciplines in the landscape of current AI. Instead, they are central actors by their own right. The whole of current ML can be written in the language of optimization, and simulation procedures are essential for solving complex problems and estimating quantities which can't be analytically obtained.

In **Part I** of the book, we focused on **simulation** methods. We learnt the fundamentals of simulation and focused on Monte Carlo simulation as one of the main pillars of simulation techniques used in modern AI. After that, we introduced Discrete Event Simulation (DES) and Queuing Theory to show how discrete simulations about specific systems are performed in practice.

In **Part II** we devoted the contents to **optimization** methods. We reviewed optimization basics and exact methods like Linear and Integer Programming. We transitioned to heuristics and metaheuristics as the go-to methods for finding efficiently good-enough solutions for problems of practical importance, including trajectory and population-based methods.

Finally, we put everything together to see how simulation and optimization methods are used in current ML, including central topics like Monte Carlo Tree Search and Reinforcement Learning. With this, we hope that the reader has gained a good grasp of the concepts involved that serve as the workhorse of modern AI, and they are able to transform this knowledge into the practice of developing advanced AI systems with relevant applications in practice.

References

Breugel, Boris van, Zhaozhi Qian, and Mihaela van der Schaar. 2023. "Synthetic Data, Real Errors: How (Not) to Publish and Use Synthetic Data." arXiv. https://doi.org/10.48550/arXiv.2305.09235.

Jordon, James, Lukasz Szpruch, Florimond Houssiau, Mirko Bottarelli, Giovanni Cherubin, Carsten Maple, Samuel N. Cohen, and Adrian Weller. 2022. "Synthetic Data – What, Why and How?" arXiv. https://doi.org/10.48550/arXiv.2205.03257.

Osais, Yahya Esmail. 2017. *Computer Simulation: A Foundational Approach Using Python*. New York: Chapman; Hall/CRC. https://doi.org/10.1201/9781315120294.