Simulation and Optimization: A Model-Driven Approach

Rubén Ruiz-Torrubiano

Nov 10, 2025

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:

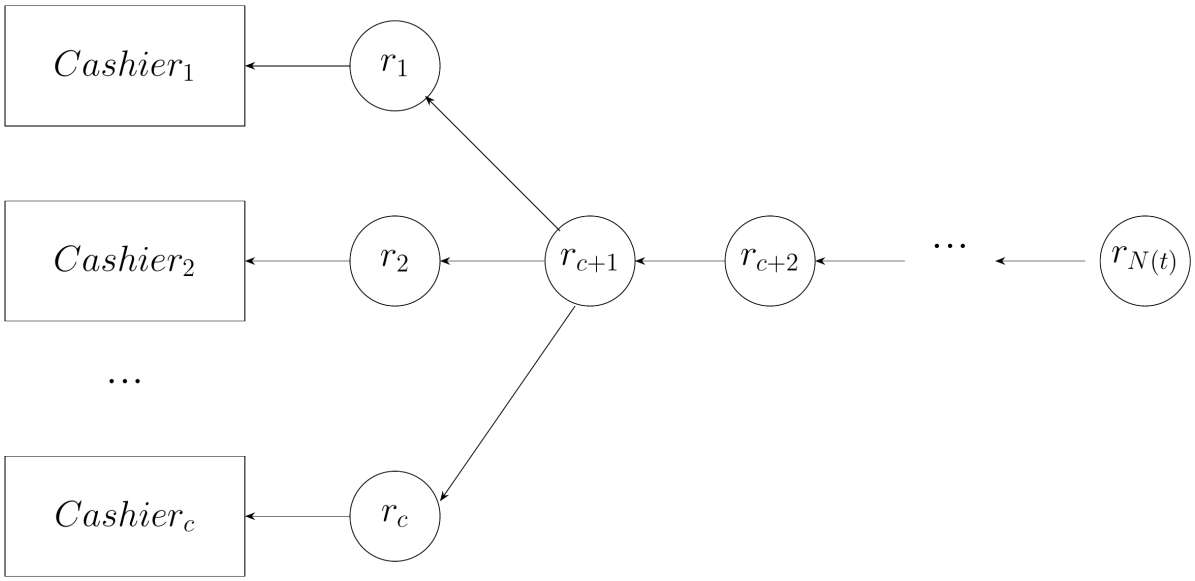

Figure 1.1: An illustration of the supermarket dynamics.

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.

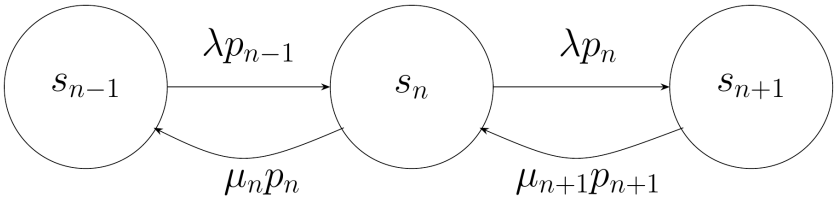

Figure 1.2: Transition dynamics between neighboring states

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.

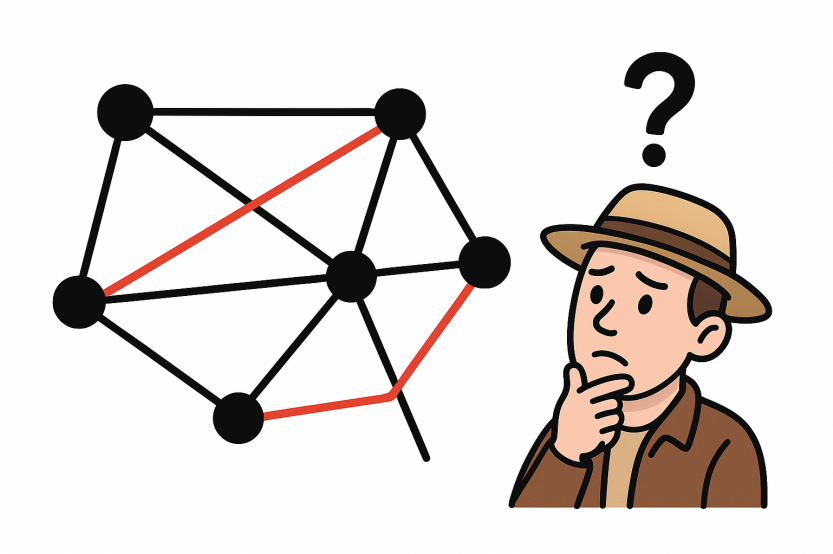

Figure 1.3: The traveling salesman has to find a good and practical solution

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 Chapter 5 concludes with considerations about agent-based modeling and simulation.

Part II is dedicated to optimization problems. In Chapter 6 we introduce the mathematical basics of optimization. Chapter 7 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 8. Finally, we review the importance of optimization methods for machine learning in Chapter 9.

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.

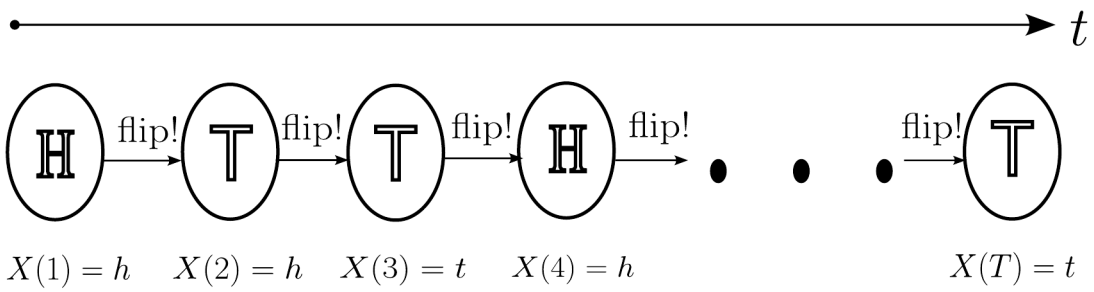

Figure 2.1: A typical Bernoulli process.

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)

plt.figure(figsize=(12, 6))
```

```
# 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()
```

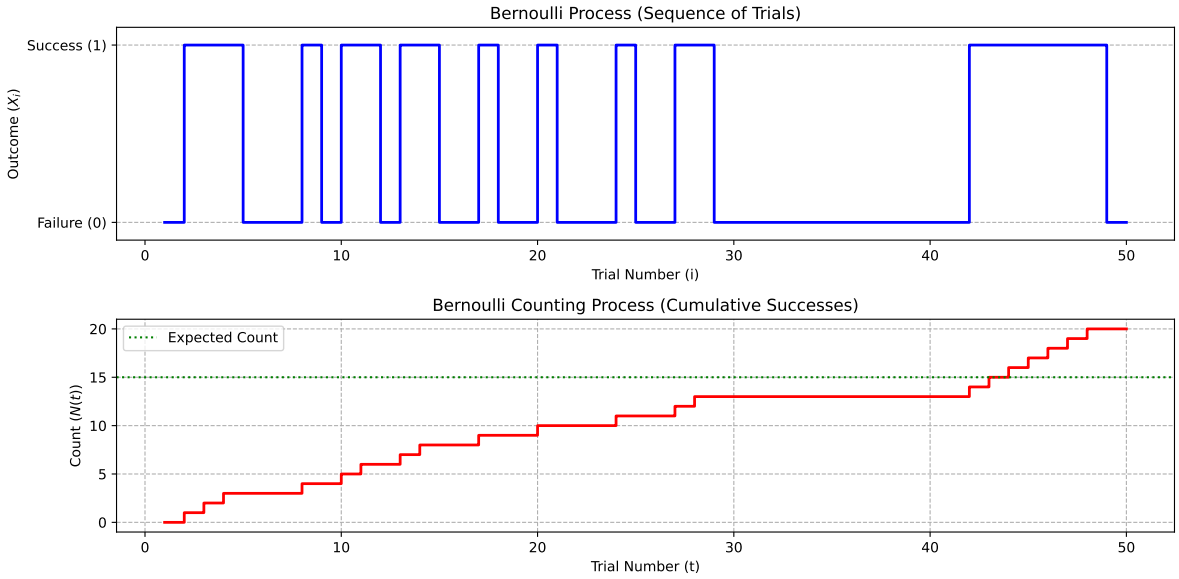

Figure 2.2: Plots of Bernoulli and counting processes

3 Monte Carlo

4 Discrete events

5 Agent-based simulation

Part II

PART II: OPTIMIZATION

6 Optimization basics

7 Exact methods

8 Metaheuristics

9 Optimization in Machine Learning

10 Summary

In summary, this book has no content whatsoever.

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.