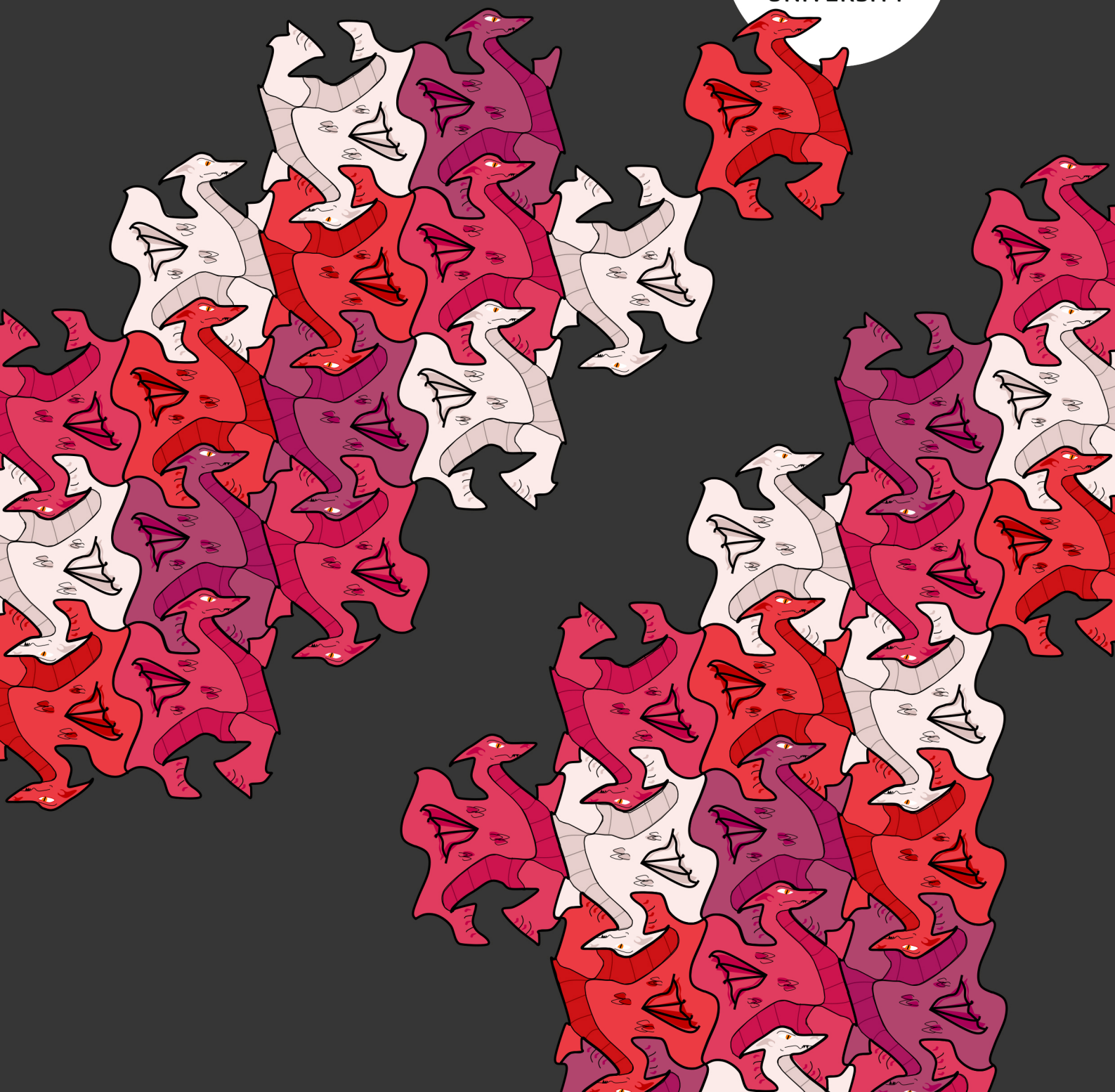
MATHEMATICS YEARBOOK 2024

School of Information Technology

Showcasing student work in undergraduate mathematics

DEAKIN UNIVERSITY



Mathematics Yearbook 2024

School of Information Technology, Deakin University

School of Information Technology
Deakin University
Burwood, VIC, Australia

ISBN: 978-0-7300-0193-5
ISBN: 978-0-7300-0502-5 (digital)
DOI: https://doi.org/10.26187/d8ep-xt85

Mathematics Yearbook 2024
Editors: L. Bandara, S. James, J. Ugon and G. Pineda Villavicencio

Cover image by Lochlan Hocking

Acknowledgement of Country
The editors and Deakin University acknowledge the Traditional Custodians of all the unceded lands, skies and waterways on which Deakin students, staff and communities come together. As we learn and teach through virtually and physically constructed places across time, we pay our deep respect to the Ancestors and Elders of Wadawurrung Country, Eastern Maar Country and Wurundjeri Country, where Deakin's physical campuses are located; as well as the Traditional Custodians of all the lands on which you may be learning and teaching, where education has taken place for many thousands of years.

Foreword

Mathematics has long been celebrated as the universal language – a tool through which we describe patterns, uncover truths, and solve problems across every domain of life. Yet beyond its logical structure and precision, mathematics is also a deeply human pursuit. It requires creativity, perseverance, and a willingness to embrace complexity, uncertainty, and even failure in the search for understanding.

This yearbook showcases the work of Deakin students who have embraced that pursuit. Whether these projects were completed as part of coursework or summer research, each one reflects the curiosity, skill, and intellectual courage of its author. You'll find a diversity of topics and approaches—from theoretical investigations to applied problem-solving. What makes this collection special is not only the quality of the work, but the diversity of approaches and motivations behind it. Some projects are driven by real-world applications, others by theoretical curiosity.

As you read through these pages, I invite you to appreciate not just the results, but the process—the questions asked, the methods explored, and the growth experienced. Mathematics is not a static body of knowledge, but a living discipline, constantly evolving through the minds of those who study it.

May this yearbook inspire you to see mathematics not only as useful, but as beautiful and may it remind us all that the pursuit of understanding is a journey worth taking.

Professor Trina Myers

Head of School, School of Information Technology

Acknowledgements

As editors and supervisors of student contributions in the *Mathematics Yearbook 2024*, we are very proud to be involved in bringing together the excellent work of our undergraduate mathematics students. In doing so, we would like to acknowledge the following.

Firstly, the teaching academics and support staff who provide the conditions under which our students can achieve their academic goals. Between changes in online learning and the rise of generative AI, universities have continued to face difficult challenges in serving the educational needs of our students, and so we acknowledge the great effort made in ensuring that the learning of students is the university's greatest priority.

We would also like to thank the School of IT and our school manager in particular, Sheena Saunders, for continuing to support the Mathematics Yearbook in providing funds for printing costs.

We would like to thank Trina Myers for generously giving her time to write the foreword and praising the contributions of our students.

This yearbook is the biggest we've had so far, and so a huge thank-you goes to Julie Higgins and the copyright team for the tireless work in providing copyright advice and helping us prepare the student contributions for publication.

Finally, we would like to thank the students who have agreed to have their work included in the yearbook, and who have made additional effort to edit and format their documents so that it is ready for publication. This year they also had to wait some additional time to see the end result, so we thank them for their patience too!

We look forward to seeing the future work of our yearbook students and we also look forward to next year's Yearbook when we do it all again!

Lashi Bandara, Simon James, Guillermo Pineda Villavicencio and Julien Ugon
Editors

Contents

1

The analysis of algorithms

Franco Diaz Licham

Abstract

Asymptotic notation is a fundamental concept in mathematics that is used in computational complexity theory to describe the behaviour of algorithms as their input size approaches infinity. Asymptotic notation provides a simplified but effective way to analyse and compare the efficiency of algorithms by focusing on their growth rates without concern of constant factors such as the type computer executing the program. In this report, the most common properties of asymptotic notations are studied. By understanding these properties, one can gain valuable insights into the performance of an algorithm and thus be able to make informed decisions when designing or analysing algorithms. Furthermore, the application of the analysis framework is used to study a simple example of a non-recursive algorithm.

1.1 Introduction

Computational Complexity

Complexity theory focuses on the study of computational problems according to their use of resources using mathematical models of computation to quantify the amount of resources needed to solve them, that is, their computational complexity [5]. Although time and space are the most studied complexity resources, any complexity measure is a computational resource such as bit complexity and communication complexity.

The evaluation of the complexity relies on the model of computation used in the problem which consists in defining the basic operations performed by the model per unit of time. Computational models such as the Turing machine are used in the study of complexity theory. A Turing machine is a simple and abstract mathematical computational device intended to help investigate the extent and limitations of what can be computed [10]. The machine manipulates symbols on a strip of infinite tape according to a table of rules performing a set of operations over these symbols before changing state and moving to the next symbol or halting computation [10]. Whilst there are many types of Turing machines used in complexity theory, a deterministic Turing machine is used in the analysis of algorithms [1].

The deterministic Turing machine is useful in modelling the central processing unit (CPU) that controls all data manipulation performed by a computer, with the machine using sequential memory to store data. The sequential memory is represented as a tape of infinite length on which the machine can perform read and write operations. The time required by a Turing machine is the total number of state transitions the machine makes before it stops and outputs an answer [1].

Despite the different types of models that can be used in complexity theory, complexity measures are not defined by which model is used, but are independent of such. Complexity measures are

defined by the Blum complexity axioms which provide a machine independent way to understand the complexity of a computation [4]. Several theorems about computational complexity have been proven for any measure of complexity resource satisfying these two axioms including the Gap and Speed-up theorems. These theorems can be proved using the Blum's axioms without any reference to any specific computational model. As a result, it applies to all measures of complexity including time and space within any context of study. In the analysis of algorithms this translates into the asymptotic notations later discussed that describes the time complexity of algorithms.

Time and Space Efficiency

The efficiency of an algorithm can be measured in terms of the time and space efficiencies and are measured as functions of the algorithm's input size n with respect to the Turing machine model M. *Time complexity*, or efficiency, indicates how fast an algorithm in question runs and is measured by counting the number of times the algorithm's basic operation is executed [15]. *Space complexity*, or efficiency, refers to the amount of memory units required by the algorithm in addition to the space needed for its input and output and is measured by counting the number of extra memory units used by the algorithm [15].

The Basic Operation

The *basic operation* is the most important operation of the algorithm and contributes the most to the total running time. For example, the basic operation for most sorting algorithms is the element comparison as they work by comparing elements in an array being sorted with each other. The established framework for the analysis of an algorithm's time complexity proposes measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n [8]. Let c_{op} be the execution time of an algorithm's basic operation on a particular computer. Let $C(n)$ be the number of times this operation needs to be executed for this algorithm. The running time $T(n)$ of a program implementing this algorithm on the computer can be estimated by Equation 1.1.

$$T(n) \approx c_{op}C(n) \tag{1.1}$$

However, a difference in running times between efficient and inefficient algorithms is not necessarily based on the hardware or software that a computer uses nor on small inputs, but by the size n approaching infinity [15]. That is, the framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency. This thus allows for the constant c_{op} to be ignored.

As the behaviour of different kinds of algorithms varies dramatically, in order to compare and rank the orders of growth of algorithms effectively, computer scientists use three asymptotic notations: Big-$\mathcal{O}$, Big-Ω and Big-Θ.

Asymptotic Notations

The Big-$\mathcal{O}$ notation is used to provide an idea of the worst possible runtime of an algorithm. A function $f(x)$ is said to be of order $g(x)$, denoted $\mathcal{O}(g(x))$, if:

$$\exists C, k \in \mathbb{R} : |f(x)| \leq Cg(x)\ \forall x \geq k \tag{1.2}$$

That is, a function $f(n)$ is said to be in $\mathcal{O}(g(n))$, if $f(n)$ is bounded above by some positive constant multiple of $g(n)$ for all large n [11]. The Big-Ω notation, is used to provide an idea of the best possible runtime of an algorithm. A function $f(x)$ is said to be of order $g(x)$, denoted $\Omega(g(x))$, if:

$$\exists C, k \in \mathbb{R} : |f(x)| \geq C|g(x)|\ \forall x \geq k \tag{1.3}$$

That is, a function $f(n)$ is said to be in $\Omega(g(n))$, if $f(n)$ is bounded below by some positive constant multiple of $g(n)$ for all large n [11]. The Big-Θ Notation is used to provide an idea of

the average possible runtime of an algorithm. A function $f(x)$ is said to be of order $g(x)$, denoted $\Theta(g(x))$, if:

$$\exists C_1, C_2, k \in \mathbb{R} : C_1 g(x) \leq |f(x)| \leq C_2 |g(x)| \; \forall x \geq k \tag{1.4}$$

That is, $f(x)$ is $\Theta(g(x))$ if $f(x)$ is $\mathcal{O}(g(x))$ and $f(x)$ is $\Omega(g(x))$. A function $f(n)$ is said to be in $\Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n [11].

However, there are many algorithms for which running time depends not only on an input size but also on the specifics of a particular input. For example, bubble sort algorithm is a native $\mathcal{O}(n^2)$ sorting algorithm, but when the list is sorted it takes $\mathcal{O}(n)$ [6]. The asymptotic notations previously discussed tell nothing about the type of input and thus, the idea of the worst-case, average-case, and best-case efficiencies need to be analysed.

Input Type Cases

The *worst-case efficiency* of an algorithm is its efficiency for the worst-case input of size n [6]. That is, it is an input of size n for which the algorithm runs the longest among all possible inputs of that size. When this is found, it guarantees that for any instance of size n, the running time will not exceed the running time on the worst-case inputs.

The *best-case efficiency* of an algorithm is its efficiency for the best-case input of size n [6]. That is, an input (or inputs) of size n for which the algorithm runs the fastest among all possible inputs of that size. *The average-case efficiency* is the algorithm's behaviour on a "typical" or "random" input [6]. Investigation of the average-case efficiency is considerably more difficult than investigation of the worst-case and best-case efficiencies and is outside the scope of this report.

1.2 Theoretical and Practical Explanations

Properties of asymptotic notations

Understanding the formal definitions and applications of the asymptotic notations is crucial in analysing the efficiency of algorithms. However, their properties also play a major role in such analysis. Especially, when deriving the basic operation of an algorithm. For example, an algorithm that involves a multi-step operation, the application of the additivity property can help to derive the function that has the most effect on the performance of the algorithm and thus determine the basic operation. Furthermore, Blum's axioms and related theorems are also shortly discussed as these provides the basis for understanding the time complexity asymptotic notations and thus their properties.

The most common properties used in the analysis of algorithms include scalability, transitivity, reflexivity, additivity, multiplicativity, and limits. Furthermore, proofs of the properties are based on the definition of the Big-$\mathcal{O}$ and can be extended similarly for the Big-Ω and the Big-Θ. [7] outline comprehensive proofs of such properties, and this section aims to explore and understand how such proofs are stated.

Blum's Axioms and Related Theorems

Blum's axioms provide a machine independent way to understand the complexity of a computation which leads to two main theorems about the complexity of computational functions. These include the Speed-up and Gap theorems. The two axioms are simply described by [12].

Definition 1.1. Let $\phi(i, x)$ be a function $\mathbb{N}^2 \to \mathbb{N}$, then ϕ is a complexity measure if it satisfies the following two axioms:

- $\phi(i, x)$ is defined if and only if the i-th Turing machine Φ_i halts on input x.

- Given i, x and y, the relation $\phi(i, x) \leq y$ is decidable.

The first axiom suggests that it only makes sense to talk about complexities that have a definite end whilst the second axiom, provides the Turing machine a computable procedure to execute $\phi(i, x)$ with a language i [12]. In order words, ϕ is a computational complexity measure which can be thought as the amount of some consumed resource by the Turing machine Φ_i in computing $\phi(i, x)$ using the program i.

In complexity theory, each computational problem can be solved by an infinite number of different programs each consuming a set of computational resources required by the different types of computations. As a result, finding an optimal program that has the lowest complexity is always desired [3]. The Speed-up and Gap theorems address this goal and propose interesting abstract rules that complexities abide by.

The speed-up theorem, in terms of time complexity, suggests that given any computable function $r(n)$, there is a language L such that for any Turing machine M using L, within a time bound $S(n)$, there exists another Turing machine M' that also uses L but within a time bound $S'(n)$ that satisfies $S'(n) \geq S(n)$ for all finite values of n. This means that for a language L there can be no fastest program computing $r(n)$ because a best program does not always exist for some computable functions [9].

The Gap Theorem is a major theorem about computational complexities stating the existence of gaps in the hierarchy of complexity classes. These complexity classes group together sets of computational problems that take a similar range of computational resources to solve, such as space and time [1]. In terms of time complexity, the Gap theorem states that given any computable function $r(n)$, there exists a time bound function $t(n)$ such that every language computable in time $r(t(n))$ is also computable in time $t(n)$. That is, no matter how much better one computer may be compared to another, there will be a $t(n)$ such that the set of functions computable in time $t(n)$ is the same for both machines [2].

Scalability

Theorem 1.1. *For the asymptotic notations $\mathcal{O}$, Ω and Θ, for all constants $c \in \mathbb{R}^+$*

$$\begin{gathered} \textit{If } f \in \mathcal{O}(g) \textit{ then } cf \in \mathcal{O}(g) \\ \textit{If } f \in \Omega(g) \textit{ then } cf \in \Omega(g) \\ \textit{If } f \in \Theta(g) \textit{ then } cf \in \Theta(g) \end{gathered}$$

Proof. By the definition of the Big-$\mathcal{O}$, there exists real positive numbers c and k, where the inequality is true $f(n) \leq cg(n)\ \forall n \geq k$ (Equation 1.2). Consider the function $af(n)$, where $a \in \mathbb{R}^+$, we can also find real positive numbers c and k, where the inequality is true

$$af(n) \leq cg(n)\ \forall n \geq k \tag{1.5}$$

As a is a real positive number, let $c = ac_1$ and by substituting into Equation 1.5, a value of k_1 can be found such that

$$\begin{aligned} af(n) &\leq cg(n) \\ af(n) &\leq ac_1 g(n) \\ f(n) &\leq c_1 g(n)\ \forall n \geq k_1 \end{aligned} \tag{1.6}$$

Equation 1.6 is essentially the definition of the Big-$\mathcal{O}$ (Equation 1.2), thus

$$af(n) \in \mathcal{O}(g(n))$$ ■

Interpretation The theorem dictates that constants do not alter the order of an algorithm. That is, that only the powers and functions are taken into account and all constans ignored.

Example Let $f(n) = 2n^2 + 5$ and $c = 2$. Determine the order of $cf(n)$:
It is known that $f(x) \in \mathcal{O}(x^2)$, thus multiplying $f(x)$ by c and applying the property

$$2f(n) = 10n^2 + 10n$$
$$10n^2 + 10n = \mathcal{O}(n^2)$$

Transitivity

Theorem 1.2. *For the asymptotic notations $\mathcal{O}$, Ω and Θ*

$$\begin{gathered}\textit{If } f \in \mathcal{O}(g) \textit{ and } g \in \mathcal{O}(h), \textit{ then } f \in \mathcal{O}(h) \\ \textit{If } f \in \Omega(g) \textit{ and } g \in \Omega(h), \textit{ then } f \in \Omega(h) \\ \textit{If } f \in \Theta(g) \textit{ and } g \in \Theta(h), \textit{ then } f \in \Theta(h)\end{gathered}$$

Proof. By the definition of the Big-$\mathcal{O}$, there exists real positive numbers c and k, where the inequality is true $f(n) \leq cg(n)\ \forall n \geq k$ (Equation 1.2). Then consider two functions $f(n)$ and $g(n)$ such that

$$f(n) \leq c_1 g(n)\ \forall n \geq k_1 \tag{1.7}$$
$$g(n) \leq c_2 h(n)\ \forall n \geq k_2 \tag{1.8}$$

Let $c_3, k_3 \in \mathbb{R}^+$ such that $c_3 = c_1 \times c_2$ and $k_3 \geq max\{k_1, k_2\}$, then by substituting Equation 1.8 into Equation 1.7

$$f(n) \leq c_1(c_2 h(n))$$
$$f(n) \leq c_1 c_2 h(n)$$
$$f(n) \leq c_3 h(n)\ \forall n \geq k_3 \tag{1.9}$$

Equation 1.9 is essentially the definition of the Big-$\mathcal{O}$ (Equation 1.2), thus

$$f(n) \in \mathcal{O}(h(n))$$ ■

Interpretation The theorem dictates that there is a relationship between the orders of two functions $f(n)$ and $h(n)$ if there is a middle function connecting the two. That is, if function $h(n)$ grows as quickly as $g(h)$, which grows as quickly as $f(n)$, then $h(n)$ grows as quickly as $f(n)$.

Example Let $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$. Determine whether $f(n) \in \mathcal{O}(h(n))$:
It can be seen that $n \leq n^3$ for some constant c and k. Therefore, $f(n) = \mathcal{O}(n^2)$. Furthermore, since $n^2 \leq n^3$ for some constant c and k, then $g(n) = \mathcal{O}(n^3)$. By the Transitivity property, since $n \leq n^3$ for some constant c and k, then $f(n) = \mathcal{O}(n^3)$.

Reflexivity

Theorem 1.3. *For the asymptotic notations $\mathcal{O}$, Ω and Θ*

$$f \in \mathcal{O}(f)$$
$$f \in \Omega(f)$$
$$f \in \Theta(f)$$

Proof. By the definition of the Big-$\mathcal{O}$, there exists real positive numbers c and k, where the inequality is true $f(n) \leq cg(n)\ \forall n \geq k$ (Equation 1.2). Since c is a real positive number, then let $c = 1$ and by substituting it can be observed that inequality is true

$$\begin{aligned} f(n) &\leq cf(n) \\ f(n) &\leq 1 \times f(n) \\ f(n) &\leq f(n)\ \forall n \geq k \end{aligned} \tag{1.10}$$

Since Equation 1.10 holds for any value of $n \geq k$ then by the Big-$\mathcal{O}$ notation

$$f(n) \in \mathcal{O}(f(n)) \qquad \blacksquare$$

Interpretation The theorem implies that since the maximum value of $f(n)$ will always be $f(n)$, then, the order of $f(n)$ will be determined by the maximum power or function. This is related to the scaling property.

Example Let $f(n) = 3n^3$, determine if $f(n) \in \mathcal{O}(f(n))$:
By the Big-$\mathcal{O}$ and choosing $c = 2$ and $k = 0$

$$\begin{aligned} 3n^3 &\leq 2 \times 3n^3 \\ 3n^3 &\leq 6n^3 \end{aligned} \tag{1.11}$$

Since Equation 1.11 is true, then $f(n) \in \mathcal{O}(n^3)$.

Additivity

Theorem 1.4. *For the asymptotic notations $\mathcal{O}$, Ω and Θ*

$$\begin{aligned} f + g &\in \mathcal{O}(max\{h, k\}) \\ f + g &\in \Omega(max\{h, k\}) \\ f + g &\in \Theta(max\{h, k\}) \end{aligned}$$

Proof. This proof relies on a simple property of real numbers [8]. Consider four real numbers a_1, b_1, a_2, and b_2. Then if $a_1 \leq b_1$ and $a_2 \leq b_2$, then

$$a_1 + a_2 \leq 2max\{b_1, b_2\} \tag{1.12}$$

By the definition of the Big-$\mathcal{O}$, there exists real positive numbers c and k, where the inequality is true $f(n) \leq cg(n)\ \forall n \geq k$ (Equation 1.2). Consider, functions $f(n)$ and $g(n)$ such that by the definition of the Big-$\mathcal{O}$

$$f(n) \leq c_1 h(n)\ \forall n \geq k_1 \tag{1.13}$$
$$g(n) \leq c_2 k(n)\ \forall n \geq k_2 \tag{1.14}$$

Let $c_3, k_3 \in \mathbb{R}^+$ such that $c_3 = max\{c_1, c_2\}$ and $k_3 \geq max\{k_1, k_2\}$, then adding Equation 1.13 and Equation 1.14, the following results

$$\begin{aligned} f(n) + g(n) &\leq c_1 h(n) + c_2 k(n) \\ &\leq c_3 h(n) + c_3 k(n) \\ &\leq c_3 (h(n) + k(n))\ \forall n \geq k_3 \end{aligned} \tag{1.15}$$

Considering that Equation 1.12, then Equation 1.15 above can be expressed as

$$f(n) + g(n) \leq 2c_3 max\{h(n) + k(n)\}\ \forall n \geq k_3 \tag{1.16}$$

Let $c = 2c_3$ and $k = k_3$, thus:

$$f(n) + g(n) \leq cmax\{h(n) + k(n)\} \; \forall n \geq k \tag{1.17}$$

Equation 1.17 is essentially the definition of the Big-O (Equation 1.2), thus

$$f(n) + g(n) \in \mathcal{O}(max\{h(n) + k(n)\})$$ ■

Interpretation The lemma implies that the algorithm's overall efficiency is determined by the least efficient part. That is, the part with the higher order of growth.

Example Let $f(n) = n$, $g(n) = n^3$ and $h(n) = f(n) + g(n)$. Find $\mathcal{O}(h(n))$:
By the order of growth of a polynomial $f(n) = \mathcal{O}(n)$ and $g(n) = \mathcal{O}(n^3)$. Since $f(n) < g(n)$, then $g(n) = max\{f(n) + g(n)\}$. As a result $h(n) \in \mathcal{O}(n^3)$.

Multiplicativity

Theorem 1.5. *For the asymptotic notations $\mathcal{O}$, Ω and Θ*

$$f \times g \in \mathcal{O}(h \times k)$$
$$f \times g \in \Omega(h \times k)$$
$$f \times g \in \Theta(h \times k)$$

Proof. By the definition of the Big-$\mathcal{O}$, there exists real positive numbers c and k, where the inequality is true $f(n) \leq cg(n) \; \forall n \geq k$ (Equation 1.2). Consider, functions $f(n)$ and $g(n)$ such that by the definition of the Big-$\mathcal{O}$

$$f(n) \leq c_1 h(n) \; \forall n \geq k_1 \tag{1.18}$$
$$g(n) \leq c_2 k(n) \; \forall n \geq k_2 \tag{1.19}$$

Let $c_3, k_3 \in \mathbb{R}^+$ such that $c_3 = c_1 \times c_2$ and $k_3 \geq max\{k_1, k_2\}$, then multiplying Equation 1.18 and Equation 1.19, the following results

$$\begin{aligned} f(n) \times g(n) &\leq c_1 h(n) \times c_2 k(n) \\ &\leq c_1 c_2 (h(n) k(n)) \\ &\leq c_3 (h(n) k(n)) \; \forall n \geq k_3 \end{aligned} \tag{1.20}$$

Equation 1.20 is essentially the definition of the Big-$\mathcal{O}$ (Equation 1.2), thus

$$f(n) \times g(n) \in \mathcal{O}(h(n) \times k(n))$$ ■

Interpretation The lemma implies that the order of the product of two functions is the same as if multiplying their orders individually. That is, the product of upper bounds of two functions results in an upper bound for the product of those two functions.

Example Let $f(n) = 2n + 1$, $g(n) = n^3 + n^2$ and $h = f(n) \times g(n)$. Find $\mathcal{O}(h(n))$:
By the order of growth of a polynomial $f(n) = \mathcal{O}(n)$ and $g(n) = \mathcal{O}(n^3)$. Then

$$\begin{aligned} \mathcal{O}(f(n) \times g(n)) &= \mathcal{O}(f(n)) \times \mathcal{O}(g(n)) \\ &= n \times n^3 \\ &= n^4 \end{aligned}$$

Limits

In order to define a limit it is important to introduce two concepts regarding the distance between two points. let a and b be points on a number line, then the distance between the two is given by $|a - b|$. Using this, it is also true that

- $|f(x) - L| < \epsilon$ is the distance between $f(x)$ and a point L being less than ϵ.
- $0 < |x - a| < \delta$ if $x \neq a$ then the distance between x and a is less than δ

Definition 1.2. Let $f(x)$ be defined for all $x \neq a$ over an open interval containing a. Let L be a real number such that

$$L = \lim_{x \to a} f(x) \tag{1.21}$$

if for every $\epsilon > 0$, there exists a $\delta > 0$, such that if $0 < |x - a| < \delta$, then $|f(x) - L| < \epsilon$.

The definition can be further understood graphically. Figure 1.1 below shows that as smaller values of ϵ are taken there are also small enough values of δ that can be found so that if a point x is chosen within δ and a, then the value of $f(x)$ is within ϵ of the limit L [13].

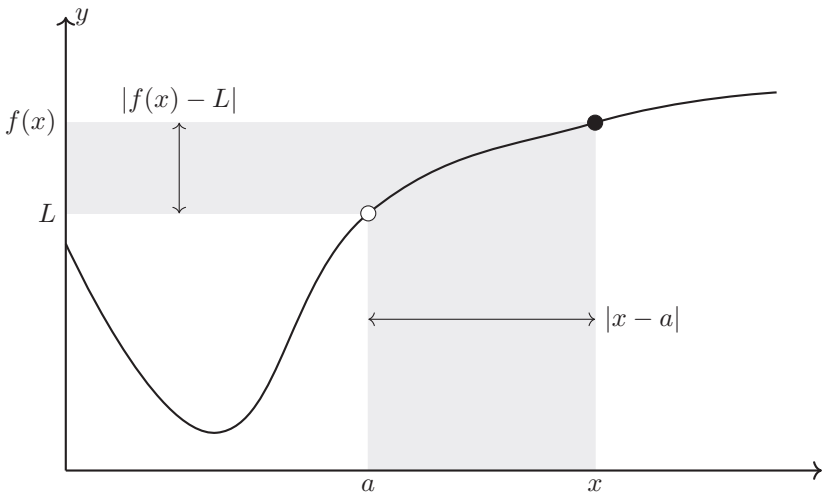


Figure 1.1: Graphical interpretation of the definition of a limit. Created using Tikz [14].

Limits provide a much more convenient method in analysing the order of growth of an algorithm. This is used when the formal proof that a function is in a certain order is not required. The method involves expanding Equation 1.21 to computing the limit of the ratio of two functions in question. One function being the one describing the algorithm's behaviour and the other is a proposed function of the order of the algorithm [8].

The relationship exists when the limit between the two function exists. That is, when the function $f(x)$ approaches a specific value as the input approaches a particular point. This value can be 0, a constant C or ∞ for all positive real numbers as shown in Theorem 1.6.

Theorem 1.6. *For the asymptotic notations $\mathcal{O}$, Ω and Θ, suppose that the limit L exists*

$$L = \lim_{x \to \infty} \frac{f(x)}{g(x)} \tag{1.22}$$

Then the following applies:

$$\begin{gathered} \textit{if } L = 0, \textit{ then } f \in \mathcal{O}(g) \textit{ and } f \notin \Omega(g) \\ \textit{if } L = C \textit{ then } f \in \mathcal{O}(g) \textit{ and } f \in \Omega(g) \textit{ thus } f \in \Theta(g) \\ \textit{if } L = \infty \textit{ then } f \in \Omega(g) \textit{ and } f \notin \mathcal{O}(g) \end{gathered}$$

Proof. Consider the Limit L below

$$L = \lim_{x \to \infty} \frac{f(x)}{g(x)}$$

If $L = 0$ then from the definition of a limit, it exists, thus there is some value of k such that:

$$\frac{f(n)}{g(n)} \leq 1 \; \forall n \geq k \tag{1.23}$$

Rearranging Equation 1.23 results in

$$f(n) \leq g(n) \; \forall n \geq k \tag{1.24}$$

Equation 1.24 is essentially the definition of the Big-O (Equation 1.2) where $c = 1$, thus

$$f(n) \in \mathcal{O}(g(n))$$

Conversely, if $c > 0$, it is not necessarily the case that $f(n) \geq cg(n) \; \forall n \geq k$. Since this cannot be guaranteed to be true, then it is true to say that

$$f(n) \notin \Omega(g(n))$$

■

Interpretation The theorem suggests different things about the relationship between the two functions being compared. 0 implies that $f(x)$ has a smaller order of growth than $g(x)$, C implies that $f(x)$ has the same order of growth as $g(x)$, ∞ implies that $f(x)$ has a larger order of growth than $g(x)$.

Example Show that $f(n) = \frac{1}{2}(n^2 - 1) \in \Theta(n^2)$:

$$L = \lim_{x \to \infty} \frac{f(n)}{g(n)} = \lim_{x \to \infty} \frac{\frac{1}{2}(n^2 - 1)}{n^2} = \frac{1}{2} \lim_{x \to \infty} \frac{n^2 - 1}{n^2} = \frac{1}{2} \lim_{x \to \infty} 1 - \frac{1}{n} = \frac{1}{2}$$

Since the limit is equal to a positive constant, the functions have the same order of growth. That is, $f(n) \in \Theta(n^2)$.

Analysis of non-recursive algorithms

Having developed the formal definitions for the Big-$\mathcal{O}$, Big-Ω and Big-Θ notations and extended on the most important properties of such, implies that all the tools necessary to study the general approach in analysing algorithms have been acquired. In this section, the analysis of a simple non-recursive algorithm will be discussed using the standard framework of analysis. This framework includes a general step-by-step approach developed by computer scientists to use when analysing algorithms [6]. Levitin summaries theses steps found in the literature to generally include the following [8]:

- Determine the variable(s) that control the size of the input, n;
- Identify the dominant operation in terms of time complexity – this will usually be the operation that is performed the most in running the algorithm;
- Find the relationship between the input size, n, and the number of times the operation identified in the previous step is performed.
- Formalise this relationship as a sum counting the number of times the operation is used;
- Find a closed-form formula or establish the order of growth.

Algorithm Example Problem

Consider the problem of finding the largest value of an element in an array of n numbers analysed by [8]. Algorithm 1 below is a common method to solving such problem. The step-by-step approach can be used to analyse the algorithm and thus understand how the algorithm behaves.

Input: An array $A[0 \dots n-1]$ of real numbers
Output: The value of the largest element in A
Set $maxval \leftarrow A[0]$ **for** $i \leftarrow 1$ `to` $n-1$ **do**
  if $A[i] > maxval$ **then**
    $maxval \leftarrow A[i]$
  end
end
return $maxval$

Algorithm 1: $MaxElement(A[0 \dots n-1])$

Firstly, the input size is determined by the number of elements in the array. That is, the value of n. Secondly, the basic operation can be determined by analysing the most important operations in the algorithm. In this example, it can be observed that there are two operations occurring in the body of the for loop. The comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Since the comparison operation is executed each iteration of the for loop and the assignment is not, then the basic operation of the algorithm is the comparison operation.

Thirdly, since the number of comparisons will be the same for all different types of arrays of size n, therefore, then the efficiency of the algorithm is only dependent on the size of the array. As a result, analysis of the worst, average and best cases in this example is not required. Fourthly, let the number of times the comparison operation is executed in the algorithm be denoted by $C(n)$, then a $C(n)$ can be derived as a function of the size n. The algorithm makes one comparison on each iteration, which is repeated for each value of the variable i within the limits of the value of 1 and $n-1$, inclusive.

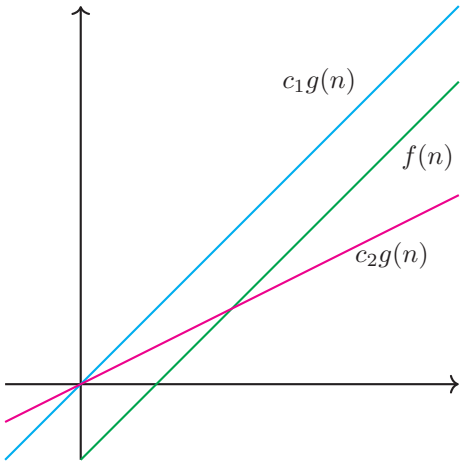


Figure 1.2: Graphical representation of $f(n) = n - 1$ with lower and upper bounds $g(n)$ with constants $c_1 = 1, c_2 = \frac{1}{2}$. Created using Tikz [14].

Therefore, $C(n)$ can be expressed as a sum Equation 1.25.

$$C(n) = \sum_{i=1}^{n-1} 1 \tag{1.25}$$

It can be observed that the closed form of the sum is derived by 1 repeating the algorithm $n-1$ times. Thus Equation 1.25 can be solved to Equation 1.26.

$$C(n) = n - 1 \tag{1.26}$$

Equation 1.26 is a linear function and thus displays a linear complexity dependent only in the input size n. Using the limit property previously discussed (Equation 1.22) and proposing that $C(n)$ has an order of growth of $g(n) = n$ then,

$$L = \lim_{n\to\infty} \frac{n-1}{n} = \lim_{n\to\infty} \frac{n}{n} - \frac{1}{n} = \lim_{n\to\infty} 1 - \frac{1}{n} = 1$$

Since the value of $L = 1$, this thus implies that $C(n) = \mathcal{O}(n)$. Furthermore, it can also be observed that this function is also in $\Omega(n)$ and as a result also in $\Theta(n)$. Figure 1.2 displays the behaviour of the efficiency of this algorithm with its lower and upper bounds.

1.3 Summary

In the realm of algorithm analysis, the properties of three asymptotic notations are indispensable tools that form part of the analysis framework for quantifying the efficiency of algorithms as input sizes become large [11]. By removing away constant factors such as hardware, these notations and their properties allows sole focus on the growth rates of algorithms as the ultimate measure of efficiency [15].

It is evident that this report has simply only scratched the surface of the complexity of studying algorithms. However, having presented the basic tools of the analysis framework means that one can be ready to further pursuit deeper knowledge in the subject area. This is highly required especially for someone in the pursuit of a career in software engineering. Having a solid foundation in the study of algorithms would certainly equip them to make informed decisions about algorithm selection, scalability and optimisation. Thus contributing to the development of more efficient and effective software solutions that are built on code that is clean, easily maintainable and extendable.

Context

This article was adapted from a report submitted for SIT192 - Discrete Mathematics.

About the Author

Franco is studying a Bachelor of Software Engineering and currently working as a Software Developer in the Education sector. He found a passion for this after finding a job in IT as a necessity during COVID-19. He is interested in cloud computing and web development and hopes to further develop his skills in these areas.

References

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2012.

[2] Andrea Asperti. A formal proof of Borodin-Trakhtenbrot's gap theorem. In *International Conference on Certified Programs and Proofs*, pages 163–177. Springer, 2013.

[3] Andrea Asperti. The speedup theorem in a primitive recursive framework. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 175–182, 2015.

[4] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):322–336, 1967.

[5] Daniel Pierre Bovet and Pierluigi Crescenzi. *Introduction to the theory of complexity*. Prentice Hall, illustrated edition, 1994.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms (3rd Edition)*. MIT Press, 2022.

[7] Michael J. Dinneen, Georgy Gimel'farb, and Mark C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages (2nd Edition)*. Pearson, 2016.

[8] Anany Levitin. *Introduction to the Design and Analysis of Algorithms (3rd Edition)*. Pearson, 2012.

[9] Chung-Chih Li. Speed-up theorems in type-2 computations using oracle turing machines. *Theory of Computing Systems*, 45:880–896, 2009.

[10] Edward E Ogheneovo. Turing machine and the conceptual problems of computational theory. *Research Inventy: Int'l Journal of Engineering and Science*, 4(4):53–60, 2014.

[11] Kenneth H. Rosen. *Discrete Mathematics and its Applications (8th Edition)*. McGraw-Hill Education, New York, 2019.

[12] Tiago Royer and Jerusa Marchi. Blum axioms and nondeterministic computation of functions. *Anais do XXXV Concurso de Trabalhos de Iniciação Científica da SBC*, pages 1–10, 2016.

[13] G. Strang and E. J. Herman. *Calculus Volume 1*. OpenStax, 2016. https://openstax.org/books/calculus-volume-1/pages/2-5-the-precise-definition-of-a-limit.

[14] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[15] Ryan T. White and Archana Tikayat Ra. *Practical Discrete Mathematics*. Packt Publishing, 2021.

2

Proofs of the centroid method for defuzzification

Tri Khuong Nguyen

Abstract

Defuzzification is an integral part that facilitates prevailing Artificial Intelligence systems to make decisions on their own. Given the importance, this paper presents an in-depth proof of the centroid method, a prominent mathematical approach for defuzzifying values. Particularly, in order to prove the formula of the centroid method, we introduce the two theorems related to weighted average of discrete elements, then instantiate their associations with the three main theorems of continuous weighted average. With the main theorems, we take a top-down approach, gather the pieces including limit, derivative, approximation, Riemann sum to eventually arrive at the correlation between the definite integral and the area. From there, we take a step further to uncover the intrinsic nature of the centroid method, which is strongly related to definite integral, area, and continuous weighted average. Finally, we exhibit its defuzzifying ability through a real-life example. Alternatively, we also look at a few applications of the centroid method in the real world, and how our two main areas of mathematics that we encounter in our journey, discrete and continuous, interact with each other.

2.1 Introduction

Today's world of technology is powered by Artificial Intelligence (AI), from self-driving cars to voice-activated assistants and smart home devices. AI systems often go beyond the ability of programmable machines, they are capable of making decisions, just like humans; because at the end of the day, *"AI"* is a word that was made for machines that can imitate humans' behaviour. As humans, we often interpret and produce information in a relative way, not absolute, meaning that things can be alternative to just true or false. While the language of computers, called binary code, is absolute, that of human and AI is relative. Thus, the transformation from traditional computers to AI systems is equivalent to the shift from binary logic to fuzzy logic.

Fuzzy logic, proposed by professor Lotfi Zadeh in 1965 [20], is a branch of logic that allows values to be any real number in the range from 0 to 1 inclusive rather than just strictly 0 and 1 as binary logic. It plays a pivotal role in fuzzy inference system, a type of AI system that produces a decision from a given input. This involves fuzzification as the first step and defuzzification as the last step [22]. These two subprocesses can be illustrated through an example. Suppose there is an air conditioner which functions as a fuzzy inference system that first measures the room's temperature, fuzzification can utilise these measurements to do some calculations, and through further computational steps, eventually, the system enters the defuzzification stage where it identifies an ideal temperature value to be set for human's comfort. While fuzzification often depends on human's preferences (i.e. each person prefers a unique temperature), defuzzification demonstrates pure mathematical approaches to defuzzify values.

Given various methods of defuzzification, this paper focuses on presenting the *centroid method*, rigorously proving the concepts and theorems related to the method, demonstrating that it can defuzzify values with a concrete example, exploring different applications of it, and connecting the two fields of mathematics that are fundamental to our proofs.

2.2 Preliminaries

Basic Definitions

There are some definitions in the fuzzy world that we need to comprehend before diving further into the underlying concepts.

Definition 2.1: Universe of Discourse

The universe of discourse $\tilde{U}$ contains all elements or objects in a particular discussion, regardless of whether they satisfy a predicate [14].

For example, if we are discussing about speed, $\tilde{U}$ contains all values of speed (we do not consider if speed is greater or less than a specific value).

Definition 2.2: Fuzzy Set

Fuzzy set is a collection of distinct elements where the degree of belongingness (theoretically it is called *membership value*, as explained in *Definition 2.3*) of every element is fuzzy, which can be any real number in the range from 0 to 1 [3].

Definition 2.3: Membership Value and Membership Function

Consider an element x and a fuzzy set $\tilde{A}$, we have $\mu_{\tilde{A}}(x)$ is the membership value of x with respect to $\tilde{A}$, which tells us the degree of truth of the proposition *"x is an element of $\tilde{A}$"* [23].

For instance, $\mu_{\tilde{A}}(x) = 0.5$ means x is 50% an element of $\tilde{A}$. When we take all possible values of x in $\tilde{U}$ into account, we have a membership function $\mu_{\tilde{A}}(x) = \begin{cases} g(x) \ \ if \ x \in A \\ h(x) \ \ if \ x \in B \\ \dots \end{cases}$ where $g(x)$, $h(x)$ are functions defined on the intervals A, B respectively [8].

Fuzzy Inference Process

We have had a glimpse of fuzzy inference system and what defuzzification does in the fuzzy inference process. However, as defuzzification uses the output of the preceding phases as its input, we need to take a step back and elaborate on these.

Definition 2.4: Fuzzy Inference Process

Fuzzy inference process takes measurable physical variables as input, then through fuzzy logic, produces a decision as an output. The process consists of 4 phases: fuzzification, implication, aggregation, and defuzzification [15].

We will go through these phases using an example for clearer understanding. Let the fuzzy inference system be a thermostat that controls the room's heating system based on the room's temperature (^{o}C) and humidity (%). The output should be an ideal heater power (W) that maintains a comfortable environment.

Fuzzification

Definition 2.5: Fuzzification

Fuzzification is the phase of taking input variables and using the membership functions with respect to fuzzy sets to assign the corresponding membership values for them [15].

With our example, the measured room's temperature and humidity are treated as the input. We have fuzzy sets $\widetilde{t\_low}$ (low temperatures), $\widetilde{t\_med}$ (medium temperatures), $\widetilde{t\_high}$ (high temperatures), $\widetilde{h\_low}$ (low humidities), $\widetilde{h\_med}$ (medium humidities), $\widetilde{h\_high}$ (high humidities) and their membership functions are defined as follows:

$$\mu_{\widetilde{t\_low}}(t) = \begin{cases} 1 & if\ t \leq 1 \\ -t+2 & if\ 1 < t \leq 2 \\ 0 & if\ t > 2 \end{cases}$$

$$\mu_{\widetilde{t\_med}}(t) = \begin{cases} 0 & if\ t \leq 1 \\ t-1 & if\ 1 < t \leq 2 \\ 1 & if\ 2 < t \leq 3 \\ -t+4 & if\ 3 < t \leq 4 \\ 0 & if\ t > 4 \end{cases}$$

$$\mu_{\widetilde{t\_high}}(t) = \begin{cases} 0 & if\ t \leq 3 \\ t-3 & if\ 3 < t \leq 4 \\ 1 & if\ t > 4 \end{cases}$$

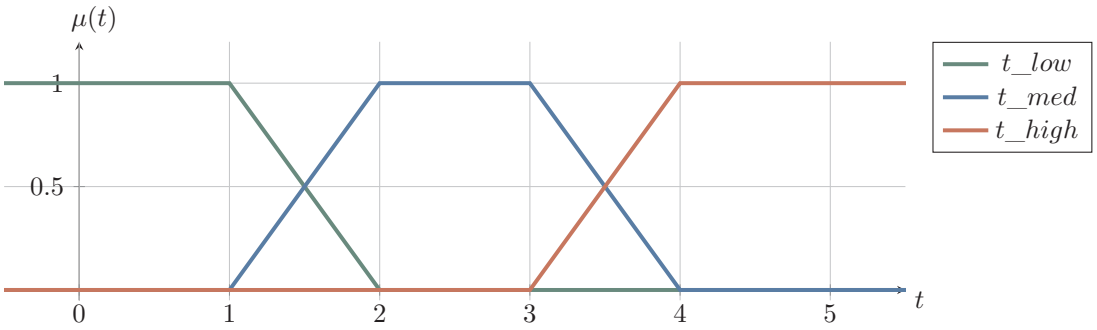


Figure 2.1: Fuzzy sets in the universe of temperatures

$$\mu_{\widetilde{h\_low}}(h) = \begin{cases} -0.25h + 1 & if\ 0 \leq h \leq 4 \\ 0 & if\ 4 < h \leq 10 \end{cases}$$

$$\mu_{\widetilde{h\_med}}(h) = \begin{cases} 0 & if\ 0 \leq h \leq 3 \\ 0.5h - 1.5 & if\ 3 < h \leq 5 \\ -0.5h + 3.5 & if\ 5 < h \leq 7 \\ 0 & if\ 7 < h \leq 10 \end{cases}$$

$$\mu_{\widetilde{h\_high}}(h) = \begin{cases} 0 & if\ 0 \leq h \leq 6 \\ 0.25h - 1.5 & if\ 6 < h \leq 10 \end{cases}$$

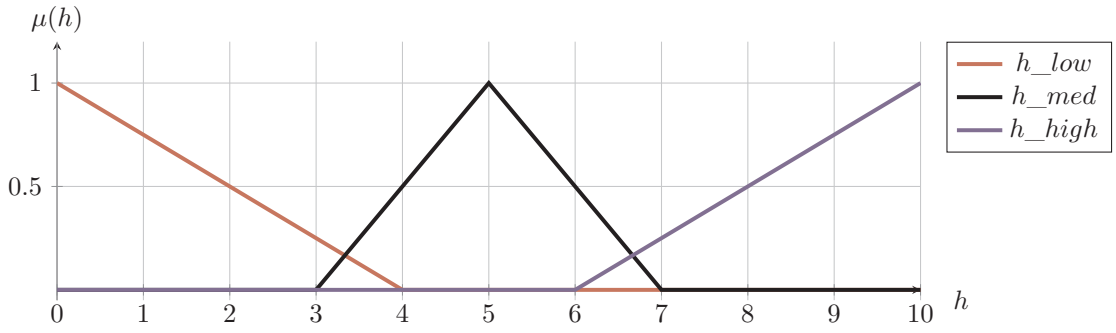


Figure 2.2: Fuzzy sets in the universe of humidities

Figure 2.1 illustrates the graphs of membership functions $\mu_{\widetilde{t\_low}}(t)$, $\mu_{\widetilde{t\_med}}(t)$, $\mu_{\widetilde{t\_high}}(t)$ of fuzzy sets $\widetilde{t\_low}$, $\widetilde{t\_med}$, $\widetilde{t\_high}$ respectively. These functions take t, an element in the universe of temperatures, as their input variable. Likewise, we have fuzzy sets in the universe of humidities as shown in *Figure 2.2*. In practice, the equations for these graphs are established by experts, so in our case, we empirically assume the shapes of these graphs and infer their equations.

Figure 2.1 and *Figure 2.2* were constructed in *desmos* (and redrawn with LaTeX for better-quality rendering), and the graphs would have stretched differently and more difficult to observe if we had drawn them in the exact unit on the x-axes. Thus, the ratio between the values on the x-axes as we see in the figures and their real values is $1 : 10$, that means, for instance, $t = 2.5$ and $h = 3$ correspond to real temperature value 25^oC and real humidity value 30% respectively.

Suppose our input variables are $t = 12$ (room's temperature is 12^oC) and $h = 40$ (room's humidity is 40%). For the universe of temperatures, projecting the points where $t = 12$ on the y-axis gives us $\mu_{\widetilde{t\_low}}(12) = 0.8$, $\mu_{\widetilde{t\_med}}(12) = 0.2$, and $\mu_{\widetilde{t\_high}}(12) = 0$ (*Figure 2.3*). For the universe of humidities, we have $\mu_{\widetilde{h\_low}}(40) = 0$, $\mu_{\widetilde{h\_med}}(40) = 0.5$, and $\mu_{\widetilde{h\_high}}(40) = 0$ (*Figure 2.4*).

We just fuzzified our input variables by assigning to them their fuzzy membership values.

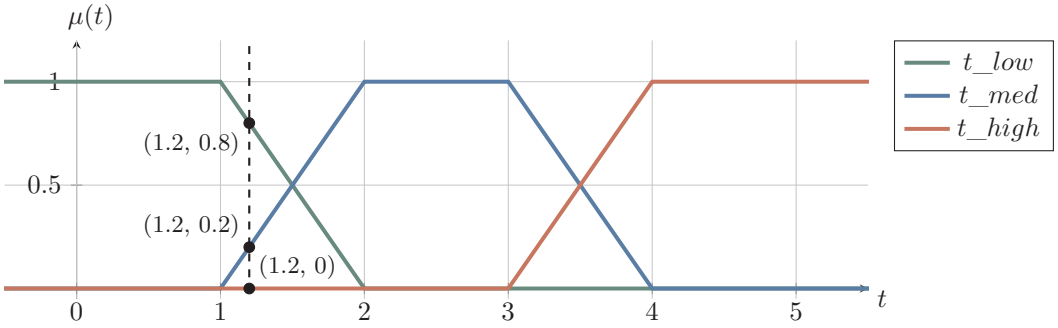


Figure 2.3: Membership values of t = 12

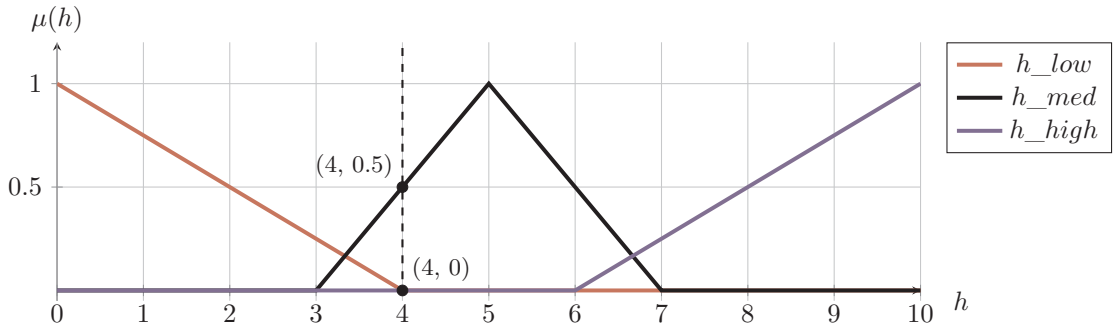


Figure 2.4: Membership values of h = 40

Implication

The next step is applying our fuzzified values for deduction. Here, we implement the logics that replicate human knowledge on the system as what we discussed about AI mimicking human's behaviour. We call these fuzzy rules or *"If-then"* rules since we often use these for our daily lives' reasonings.

Definition 2.6: Fuzzy Rule

A fuzzy rule is a conditional statement in the form "$If\ x\ is\ A\ then\ y\ is\ B$" [13] where x and y are elements of distinct universes of discourse; A and B are fuzzy sets defined in universes containing x and y respectively.

We first evaluate the antecedent (i.e. "$x\ is\ A$" in the *"If"* part), then assign the result to the consequent (i.e. "$y\ is\ B$" in the *"Then"* part) [13]. "$x\ is\ A$" and "$y\ is\ B$" are atomic propositions, but we can make them compound propositions by joining the atomic ones with fuzzy operators "AND", "OR", etc [13].

With our example, we want to define three fuzzy rules as in *Table 2.1*.

Table 2.1: Fuzzy rules

Rule 1	If *temperature* is *low* OR *humidity* is *high*, then *heater power* is *high*.
Rule 2	If *temperature* is *medium* AND *humidity* is *medium*, then *heater power* is *moderate*.
Rule 3	If *temperature* is *high*, then *heater power* is *low*.

With *Rule 1*, we extract the values from both atomic propositions of the antecedent. The proposition *"temperature* is $\widetilde{low}$*"* refers to the membership value of our measured temperature with respect to fuzzy set $\widetilde{t\_low}$, i.e. $\mu_{\widetilde{t\_low}}(12) = 0.8$. Likewise, *"humidity* is *high"* corresponds to $\mu_{\widetilde{h\_high}}(40) = 0$. These two propositions are connected by a fuzzy operator "OR", which performs a "max" operation that chooses the maximum value between two atomic propositions to give us the degree of truth for *Rule 1* ($Deg(R1)$) [16]. Basically, we proceed with the following calculation:

$$\begin{aligned} Deg\,(R1) &= max(\mu_{\widetilde{t\_low}}(12)\,,\ \mu_{\widetilde{h\_high}}(40)) \\ Deg\,(R1) &= max\,(0.8,\ 0) \\ Deg\,(R1) &= 0.8 \end{aligned}$$

With *Rule 2*, our evaluation for the values of atomic propositions is similar to *Rule 1*, except that the fuzzy operator "AND" performs the "min" operation. Thus, the degree of truth for *Rule 2* ($Deg(R2)$) is calculated as follows:

$$\begin{aligned} Deg\,(R2) &= min(\mu_{\widetilde{t\_med}}(12)\,,\ \mu_{\widetilde{h\_med}}(40)) \\ Deg\,(R2) &= min\,(0.2,\ 0.5) \\ Deg\,(R2) &= 0.2 \end{aligned}$$

Since the antecedent of *Rule 3* only contains one atomic proposition, the degree of truth for *Rule 3* ($Deg(R3)$) is simply the value of that proposition, which is:

$$\begin{aligned} Deg\,(R3) &= \mu_{\widetilde{t\_high}}(12) \\ Deg\,(R3) &= 0 \end{aligned}$$

Table 2.2 synthesizes these values.

Table 2.2: Degrees of truth of fuzzy rules

Rule	Degree of truth
Rule 1	0.8
Rule 2	0.2
Rule 3	0

For these degrees to be useful, we need to identify the membership functions representing fuzzy sets of the consequents, i.e., fuzzy sets in the universe of heater powers. We have fuzzy sets $\widetilde{P\_low}$ (low heater powers), $\widetilde{P\_mod}$ (moderate heater powers), $\widetilde{P\_high}$ (high heater powers), and their membership functions are defined as follows:

$$\mu_{\widetilde{P\_low}}(P) = \begin{cases} 1 & if\ 0 \leq P \leq 2 \\ -0.5P + 2 & if\ 2 < P \leq 4 \\ 0 & if\ 4 < P \leq 10 \end{cases}$$

$$\mu_{\widetilde{P\_mod}}(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.5P - 1.5 & if\ 3 < P \leq 5 \\ -0.5P + 3.5 & if\ 5 < P \leq 7 \\ 0 & if\ 7 < P \leq 10 \end{cases}$$

$$\mu_{\widetilde{P\_high}}(P) = \begin{cases} 0 & if\ 0 \le P \le 6 \\ 0.5P - 3 & if\ 6 < P \le 8 \\ 1 & if\ 8 < P \le 10 \end{cases}$$

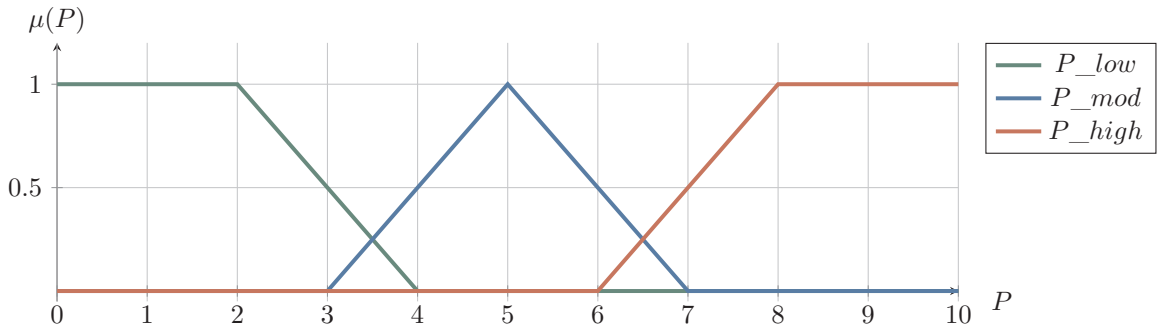


Figure 2.5: Fuzzy sets in the universe of heater powers

For better observation as the two previous graphs, we set the ratio between the values on the x-axis and their real values to be $1 : 100$, that means, $P = 10$ corresponds to the heater power $1000W$.

After calculating the degrees of all rules and constructing the consequents' functions, we begin the implication phase. Implication reshapes (truncates) each consequent's membership function based on the degree of truth of their rule [16].

Recall *Rule 1* where the proposition of the consequent is *"heater power* is *high"*, and the degree of truth for *Rule 1* is $Deg(R1) = 0.8$ as calculated. These tell us that the statement *"heater power* is *high"* is 80% true, and that means we should only consider the membership values of the function $\mu_{\widetilde{P\_high}}(P)$ up to 0.8. Hence, the curve of $\mu_{\widetilde{P\_high}}(P)$ is truncated at the membership value equals to 0.8 (*Figure 2.6*) [16]. After truncation, the function of $\mu_{\widetilde{P\_high}}(P)$ is:

$$\mu_{\widetilde{P\_high}}(P) = \begin{cases} 0 & if\ 0 \le P \le 6 \\ 0.5P - 3 & if\ 6 < P \le 7.6 \\ 0.8 & if\ 7.6 < P \le 10 \end{cases}$$

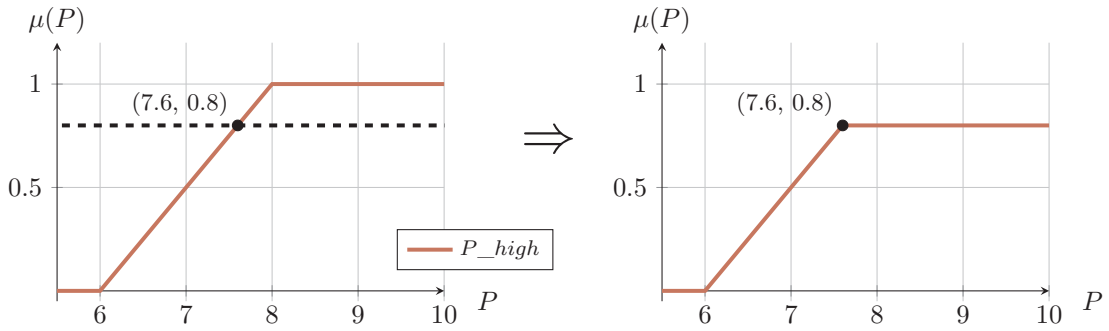


Figure 2.6: Truncation of $\mu_{\widetilde{P\_high}}(P)$

We truncate the $\mu_{\widetilde{P\_mod}}(P)$ curve at 0.2 as the consequent of *Rule 2* is *"heater power* is *moderate"* and $Deg(R2) = 0.2$ (*Figure 2.7*). The modified function of $\mu_{\widetilde{P\_mod}}(P)$ after truncating is given by:

$$\mu_{\widetilde{P\_mod}}(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.5P - 1.5 & if\ 3 < P \leq 3.4 \\ 0.2 & if\ 3.4 < P \leq 6.6 \\ -0.5P + 3.5 & if\ 6.6 < P \leq 7 \\ 0 & if\ 7 < P \leq 10 \end{cases}$$

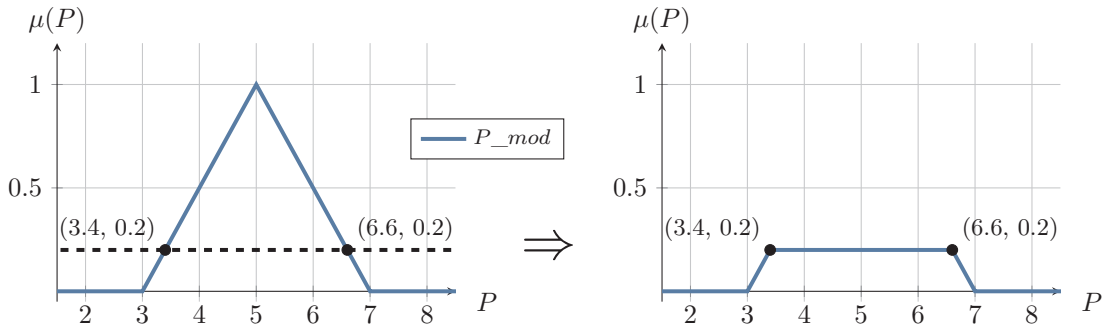


Figure 2.7: Truncation of $\mu_{\widetilde{P\_mod}}(P)$

With *Rule 3*, $Deg(R3) = 0$ implies that the statement *"heater power* is *low"* is 0% true, or completely false, therefore, the truncation of $\mu_{\widetilde{P\_low}}(P)$ is implemented at 0, means we do not take any membership value of $\mu_{\widetilde{P\_low}}(P)$ into account (*Figure 2.8*). The new function of $\mu_{\widetilde{P\_low}}(P)$ is the horizontal line $y = 0$ on the domain of heater powers:

$$\mu_{\widetilde{P\_low}}(P) = 0\ for\ 0 \leq P \leq 10$$

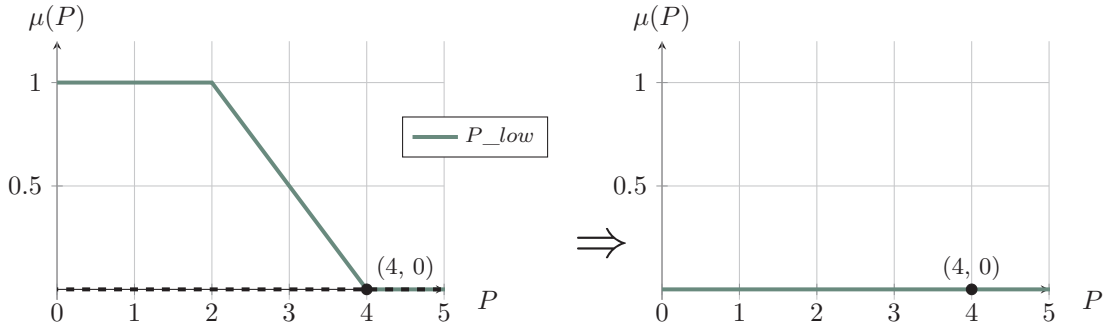


Figure 2.8: Truncation of $\mu_{\widetilde{P\_low}}(P)$

Aggregation

We have modified all three individual heater power types' membership functions based on their degrees of truth in accordance with the corresponding rules. Nevertheless, we want our system to

grasp all the rules we set out from the start because that will tell it to consider all cases before deciding which heater power value to set. This is achieved through aggregation.

Definition 2.7: Aggregation

Aggregation is the process that combines all rules' membership functions into a single function [17].

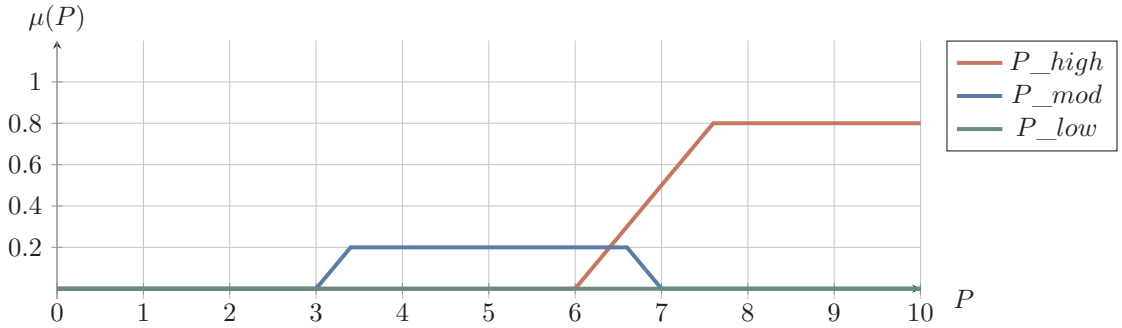


Figure 2.9: Truncated membership functions in universe of heater powers

Figure 2.9 shows all membership functions of heater powers that have been truncated. We can aggregate these functions using the "*max*" method where we take the highest membership value for every heater power on the x-axis [17]. This is illustrated by the purple curve in *Figure 2.10*.

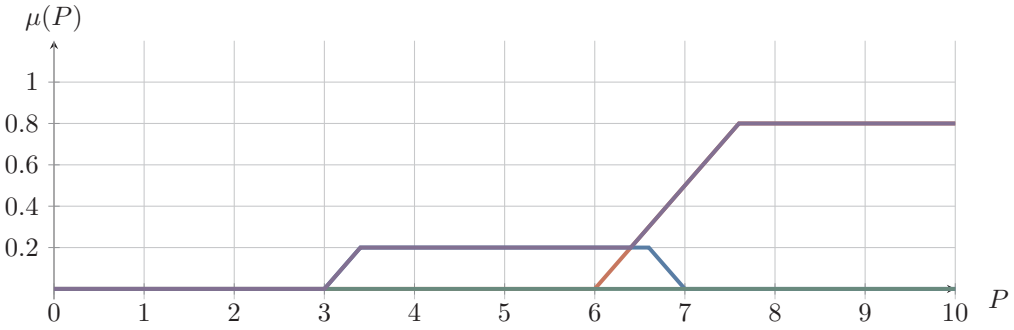


Figure 2.10: Max method for aggregation

This purple curve is also the result of aggregation. Our aggregated function $\widetilde{P\_agg}$ (*Figure 2.11*) is defined by the piecewise function:

$$\mu_{\widetilde{P\_agg}}(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.5P - 1.5 & if\ 3 < P \leq 3.4 \\ 0.2 & if\ 3.4 < P \leq 6.4 \\ 0.5P - 3 & if\ 6.4 < P \leq 7.6 \\ 0.8 & if\ 7.6 < P \leq 10 \end{cases}$$

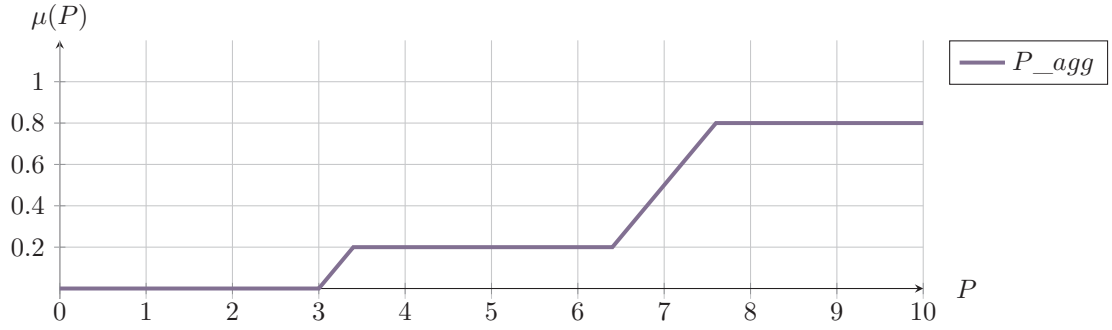


Figure 2.11: Aggregated membership function for heater powers

Defuzzification

We come to the final stage of our fuzzy inference process. Prior to this phase, we constructed the $\widetilde{P\_agg}$ function and sketched its graph, in which each point informs us a power value of the heater (projected value on the x-axis) and its degree of truth (projected value on the y-axis). Our goal is to determine an exact value (on the x-axis) for the heater to adjust its power to. Intuitively, by looking at the graph, we can predict that the ideal power value might be somewhere on the right-hand side. We do this by weighing the higher points more because we know that from the graph of degree of truth, the higher a point is, or the more likely that it is true, the nearer its x value will be to the result we are looking for. However, we are guessing and being uncertain whereas machines have a strong bond with mathematics, which entails a rigid procedure to follow, and the one we are referring to is called *defuzzification*.

Definition 2.8: Defuzzification [17]

Defuzzification is the process of converting the aggregated fuzzy set into a single crisp value, representing a definitive and practical outcome of the fuzzy inference system.

Established defuzzification methods are centroid, bisector, middle of maximum, largest of maximum, smallest of maximum, etc., but the most popular one is the centroid method which we will construct a formal proof for in the following section.

2.3 The Centroid Method

Definition 2.9: Centroid Method

Centroid, also known as Center of Gravity (COG) or Center of Area (COA), is a method that is used to compute the result of defuzzification by taking the weighted average of all possible elements in the aggregated fuzzy set, where each element is weighted by its corresponding membership value [7]. It is defined by the formula [25]:

$$P^* = \frac{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} \tag{2.1}$$

Where P^* is our defuzzified result and $\mu_{\widetilde{P\_agg}}(P)$ represents the aggregated membership function of $\widetilde{P\_agg}$. We account for all elements in the domain $[a, b]$, thus the definite integral from a to b.

While these definitions seem to be overwhelming, we will prove this method by constructing proofs for each theorem on the way.

Notice that in the definition of the centroid method, there are two terms *"weighted average"* and *"all possible elements"*. In the earlier phases, we worked with an example that we do not want to exclude any power value of the heater from $0W$ to $1000W$. A representation for the domain of $\widetilde{P\_agg}$ is $\{P \ : P \in \mathbb{R} \ \& \ P \in [0,\ 1000]$, hence the number of elements is infinite when we are referring to *"all possible elements"*. The problem with infinity is that it is uncountable, so it needs high-level mathematical concepts to address. Given that, we will start by looking at a simple case, the weighted average of discrete elements (finite and countable number of elements that can be separated).

2.4 The World of Discreteness

Average

Definition 2.10: Average

Average is one of the most basic concepts in mathematics, it is defined as the result after adding the values together and dividing by the number of added values [5].

Mathematically, the formula for average Avg is as follows:

$$Avg = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2.2}$$

We take the sum of all elements from x_1 to x_n, then divide by the number of elements n.

An insight of average can be seen through *Theorem 2.1*.

Theorem 2.1: Equal Contributions

In the calculation of average, every element x_i equally contributes a percentage of its value to the whole.

We can see this through *Example 2.1*.

Example 2.1: Equal Contributions

We have $x_1 = 1,\ x_2 = 2,\ x_3 = 3$ as our values. In this case, our average is:

$$Avg = \frac{x_1 + x_2 + x_3}{3} = \frac{1 + 2 + 3}{3}$$

We, however, are not interested in calculating this average, instead, we are going to separate the fraction:

$$Avg = \frac{1}{3} + \frac{2}{3} + \frac{3}{3} = 1 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} + 3 \cdot \frac{1}{3}$$

The numbers in blue are our initial elements x_1, x_2, x_3; the ones in green are the contributions of these elements.

We can easily see that all our elements have an equal contribution, that is $\frac{1}{3}$. Not only that, the sum of all contributions is also 1:

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = \frac{3}{3} = 1$$

Number 1 here corresponds to the percentage 100%.

We can form a general proof (*Proof 2.1*) for *Theorem 2.1* as:

Proof 2.1: Equal Contributions (*Theorem 2.1* Proof)

$$\begin{aligned} Avg &= \frac{\sum_{i=1}^{n} x_i}{n} \\ &= \frac{x_1 + x_2 + \cdots + x_n}{n} \\ &= \frac{x_1}{n} + \frac{x_2}{n} + \cdots + \frac{x_n}{n} \\ &= x_1 \cdot \frac{1}{n} + x_2 \cdot \frac{1}{n} + \cdots + x_n \cdot \frac{1}{n} \end{aligned}$$

Every element has an equal contribution of $\frac{1}{n}$, and we have n elements, so the total contribution is:

$$n \cdot \frac{1}{n} = \frac{n}{n} = 1$$

Thus, the percentage of the whole is 1 (or 100%), and *Theorem 2.1* is proved.

Weighted Average

Definition 2.11: Weighted Average

While all elements in average share the same contribution, weighted average is an extension of that where elements have varying degrees of contribution [12], still the property of summing all degrees to 1 is retained. The formula of weighted average W_Avg is given by [11]:

$$W\_Avg = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i} \tag{2.3}$$

where w_i is the weight of element x_i.

We will go through an example (*Example 2.2*) of weighted average.

Example 2.2: Varying Contributions

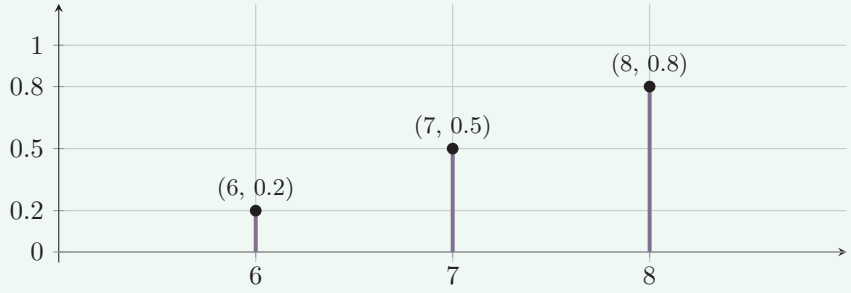


Figure 2.12: Weighted average example

In *Figure 2.12*, we have elements $x_1 = 6,\ x_2 = 7, x_3 = 8$ and their weights are $w_1 = 0.2,\ w_2 = 0.5,\ w_3 = 0.8$ respectively. Here the weight of an element is represented by the length of the line segment at the x coordinate of that element, or the y coordinate of the point on the top of that line segment. We apply the weighted average formula on these components, then break it down into multiple fractions like what we did with average:

$$W\_Avg = \frac{w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3}{w_1 + w_2 + w_3}$$

$$= \frac{0.2 \cdot 6 + 0.5 \cdot 7 + 0.8 \cdot 8}{0.2 + 0.5 + 0.8}$$

$$= \frac{0.2 \cdot 6}{0.2 + 0.5 + 0.8} + \frac{0.5 \cdot 7}{0.2 + 0.5 + 0.8} + \frac{0.8 \cdot 8}{0.2 + 0.5 + 0.8}$$

$$= 6 \cdot \frac{0.2}{0.2 + 0.5 + 0.8} + 7 \cdot \frac{0.5}{0.2 + 0.5 + 0.8} + 8 \cdot \frac{0.8}{0.2 + 0.5 + 0.8}$$

$$= 6 \cdot \frac{0.2}{1.5} + 7 \cdot \frac{0.5}{1.5} + 8 \cdot \frac{0.8}{1.5}$$

Clearly, the degrees of contribution of our elements are not the same as what we saw with average, but the sum of all degrees is still 1:

$$\frac{0.2}{1.5} + \frac{0.5}{1.5} + \frac{0.8}{1.5} = \frac{1.5}{1.5} = 1$$

So, we have *Theorem 2.2*:

Theorem 2.2: Varying Contributions

In weighted average, a degree of contribution of an element can be interpreted as a percentage which that element contributes to the whole (degrees can be equal or not equal for distinct elements), and adding all degrees will give us the percentage of the whole, represented by 1 *(or 100% as another way of interpreting the whole's percentage).*

We construct a general proof (*Proof 2.2*) for *Theorem 2.2* as follows:

Proof 2.2: Varying Contributions (*Theorem 2.2* Proof)

$$W\_Avg = \frac{\sum_{i=1}^{n} w_i \cdot x_i}{\sum_{i=1}^{n} w_i}$$

$$= \frac{w_1 \cdot x_1 + w_2 \cdot x_2 + \cdots + w_n \cdot x_n}{w_1 + w_2 + \cdots + w_n}$$

$$= \frac{w_1 \cdot x_1}{w_1 + w_2 + \cdots + w_n} + \frac{w_2 \cdot x_2}{w_1 + w_2 + \cdots + w_n} + \cdots + \frac{w_n \cdot x_n}{w_1 + w_2 + \cdots + w_n}$$

$$= x_1 \cdot \frac{w_1}{w_1 + w_2 + \cdots + w_n} + x_2 \cdot \frac{w_2}{w_1 + w_2 + \cdots + w_n} + \cdots + x_n \cdot \frac{w_n}{w_1 + w_2 + \cdots + w_n}$$

The degrees $\frac{w_i}{w_1+w_2+\cdots+w_n}$ of x_i and $\frac{w_j}{w_1+w_2+\cdots+w_n}$ of x_j $(1 \leq i, j \leq n$ and $i \neq j)$ can be different if

$w_i \neq w_j$, but they can still be equal if $w_i = w_j$. This holds for any arbitrary elements. If we consider each degree to be a percentage, all degrees will sum up to 1:

$$\frac{w_1}{w_1 + w_2 + \cdots + w_n} + \frac{w_2}{w_1 + w_2 + \cdots + w_n} + \cdots + \frac{w_n}{w_1 + w_2 + \cdots + w_n}$$

$$= \frac{w_1 + w_2 + \cdots + w_n}{w_1 + w_2 + \cdots + w_n} = 1 \ (satisfied)$$

2.5 The Shift from Discrete to Continuous

We have just discovered the *"weighted average"* term that we encountered in *Definition 2.11*, but that was just the case of discrete elements. Since the centroid method is essentially the weighted average of all possible elements and the number of elements is infinite, we now consider the weighted average of infinite elements. When we think of infinity, we imagine an extremely large number, so instead of having three discrete points as in *Figure 2.12*, we can have numerous discrete points like in *Figure 2.13*.

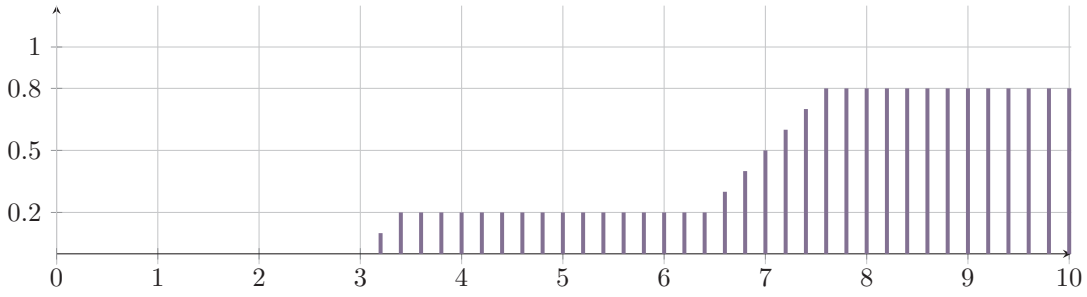


Figure 2.13: Weighted average of numerous discrete points

Same as *Figure 2.12*, the length of line segment at position x indicates the weight of element x (e.g. length of line segment at $x = 7$ is the weight of element 7). Viewing the lengths (weights) of all elements as a whole, we might notice something special, and that gradually becomes unfolded as the number of elements approaches infinity (*Figure 2.14*).

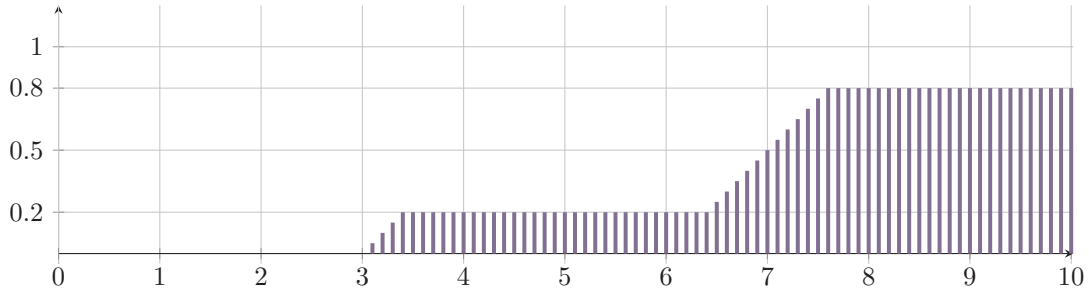


Figure 2.14: Weighted average of roughly infinite discrete points

Now, we can clearly see that a roughly infinite number of discrete line segments are filling up a continuous region, which we often refer to as an area.

The approximate area in *Figure 2.14* has a shape that is identical to the area below our $\widetilde{P\_agg}$ curve (*Figure 2.11*) and above the horizontal axis. This is a notable analogy that brings us the correlation between the discussed weighted average concept in the world of discreteness and our main centroid method in the world of continuum. If we interpret $\mu_{\widetilde{P\_agg}}(P)$ in *Eqn.* (11.1) as a function of weights, we start to see how the centroid method's formula (11.1) looks somewhat similar to the weighted average formula (7.21). From the discrete world with the sum ($\sum$) of separated weights (the denominator of (7.21)), as the number of elements goes to infinity, we enter the continuous world with the integral ($\int$) of a continuous weight function (the denominator of (11.1)), which is the same as the shift from infinite weights (*Figure 2.14*) to the area under the curve $\widetilde{P\_agg}$.

2.6 The World of Continuum

In this section, we will dig deeper into the shift that we talked about previously.

As we touched on the transformation, we mentioned that the integral of a function gives us the area under the curve of that function, and the centroid method is the weighted average in the world of continuum. These are typical concepts in calculus which we will mathematically prove in the following subsections. Specifically, our proofs will demonstrate the three main theorems:

Theorem 2.3: Area by Integration

The definite integral from a to b of a function $F'(P)$ is the area S bounded by its curve and the horizontal axis from a to b (with $a, b \in \mathbb{R}$):

$$S = \int_a^b F'(P)\ dP$$

Theorem 2.4: Calculation of Area

The area bounded by the curve of a function $F'(P)$ and the horizontal axis in a defined interval $[a, b]$ can be calculated by subtracting that function's integral $F(P)$ value at a from its integral value at b:

$$S = \int_a^b F'(P)\ dP = F(b) - F(a)$$

The prime symbol in $F'(P)$ denotes the derivative (a concept that will be covered below) of $F(P)$, but this does not mean that *Theorem 2.3* and *Theorem 2.4* can only be applied to derivatives, in fact, they are true for all functions. Take an arbitrary function $f(P)$ and its integral $g(P)$, we have the following expression upon applying the two mentioned theorems:

$$S = \int_a^b f(P)\ dP = g(b) - g(a)$$

Theorem 2.5: Continuous Weighted Average

In continuous weighted average, a degree of contribution of an element (in infinite number of elements) can be interpreted as a percentage which that element contributes to the whole (degrees can be equal or not equal for distinct elements), and the sum of all degrees is the percentage of the whole, represented by 1.

2.7 The Limit

We will be using *"the limit"* many times during our following proofs, so it is crucial to understand it.

Limit L is the value that a function $f(P)$ gets arbitrarily close to when its independent variable P approaches a value a. This is expressed through *Eqn.* (2.4):

$$\lim_{P \to a} f(P) = L \tag{2.4}$$

This is exemplified by *Example 2.3*:

Example 2.3: Example of Limit

$\lim_{n\to\infty} \frac{1}{n} = 0$ means the limit of function $\frac{1}{n}$ is 0 when n approaches infinity.

We have just defined the limit in a semantic way, but in mathematics, a formal and precise definition is always needed. The phrase "$f(P)$ gets arbitrarily close to L" can be converted into the language of mathematics as "the distance between $f(P)$ and L is smaller than an arbitrary ϵ" and expressed as $|f(P) - L| < \epsilon$. Likewise, "P approaches a" means the distance between P and a is smaller than a number δ, or $|P - a| < \delta$. We do not consider whether P touches a or not because with the example $\lim_{n\to\infty} \frac{1}{n} = 0$ that we have seen, n cannot be equal to infinity (since infinity is just a concept that represents a very large number, not a number itself) but the limit still exists. That is why the distance between P and a should be greater than 0 ($|P - a| > 0$).

A fully formalised definition of limit is:

Definition 2.12: Limit

$\lim_{P \to a} f(P) = L$ if for every number $\epsilon > 0$ there is some number $\delta > 0$ such that $|f(P) - L| < \epsilon$ whenever $0 < |P - a| < \delta$ [10].

The distances are represented by the absolute symbols in both $|f(P) - L| < \epsilon$ and $0 < |P - a| < \delta$, and we can solve absolute inequalities [21] to find the intervals of P and $f(P)$:

Table 2.3: Find intervals of P *and* $f(P)$

Steps	$\lvert f(P) - L \rvert < \epsilon$	$0 < \lvert P - a \rvert < \delta$
1	$-\epsilon < f(P) - L < \epsilon$	$\begin{cases} P - a < 0 \text{ or } P - a > 0 \\ -\delta < P - a < \delta \end{cases}$
2	$L - \epsilon < f(P) < L + \epsilon$	$\begin{cases} P < a \text{ or } P > a \\ a - \delta < P < a + \delta \end{cases}$
3	$L - \epsilon < f(P) < L + \epsilon$	$a - \delta < P < a \text{ or } a < P < a + \delta$

These intervals can be depicted through the below figure:

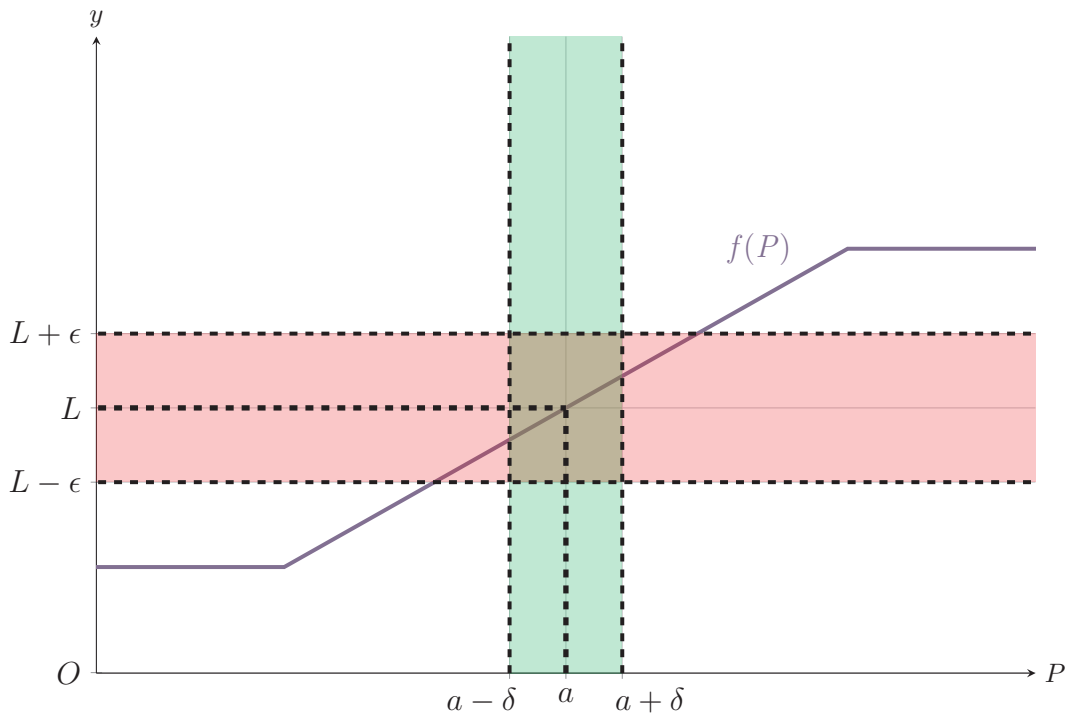


Figure 2.15: Epsilon-Delta definition of limit

In *Figure 2.15*, the limit L only exists if there is some part of the graph that falls into the intersection of the red region excluding $L+\epsilon$, $L-\epsilon$ ($L-\epsilon < f(P) < L+\epsilon$ satisfied) and the green region excluding a, $a+\delta$, $a-\delta$ ($a-\delta < P < a$ *or* $a < P < a+\delta$ satisfied) [10].

"For every $\epsilon > 0$*"* means the limit L still exists if we narrow the red region to be as small as possible, or choose an arbitrarily small ϵ (but has to be positive for the red region itself to exist) and the green region can still shrink small enough but still exists (*"there is some* $\delta > 0$*"*) to make every P in the green region (excluding $a-\delta$, $a+\delta$, a) to fully contain $f(P)$ in the red region (excluding $L+\epsilon$, $L-\epsilon$).

With c as a constant, we have the three types of limits that we will see in the following subsections:

$$\lim_{n\to\infty} \frac{c}{n} = 0 \tag{2.5}$$

$$\lim_{n\to 0} \frac{c}{n} = \infty \tag{2.6}$$

$$\lim_{n\to 0} (c+n) = c \tag{2.7}$$

The limit in *Eqn.* (2.6) does not exist since a limit L must be an exact value (e.g. a number), not infinity. However, we can still express it like in (2.6).

Now, we are ready for the main parts of our journey.

2.8 The Derivative

The first part of the proof is called *"the derivative"*. We will start the proof (*Proof 2.3*) for *Theorem 2.3* by building the definition of derivative through the below figures.

Proof 2.3: Area by Integration (*Theorem 2.3* Proof)

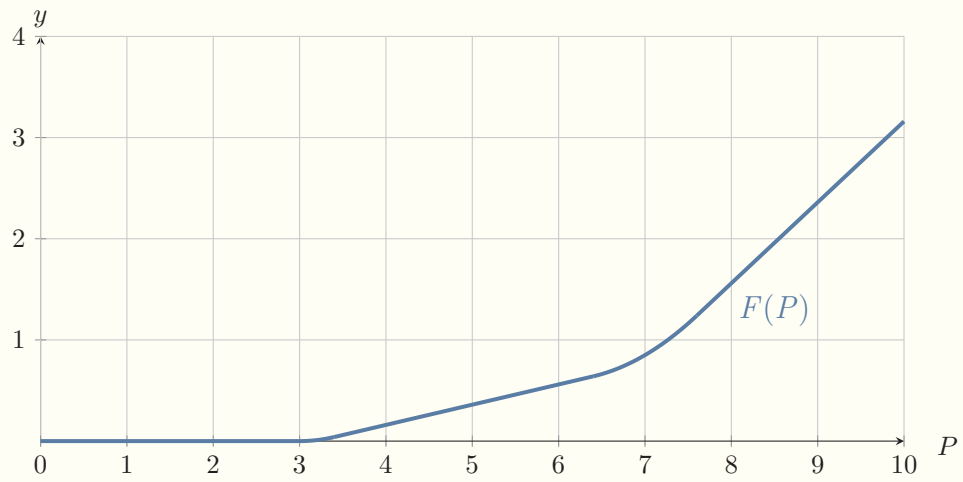


Figure 2.16: Integral of $\mu_{\widetilde{P\_agg}}(P)$

Figure 2.16 shows a function $F(P)$ that is an integral of the function $\mu_{\widetilde{P\_agg}}(P)$. At this rate, we do not need to know what an *"integral"* is.

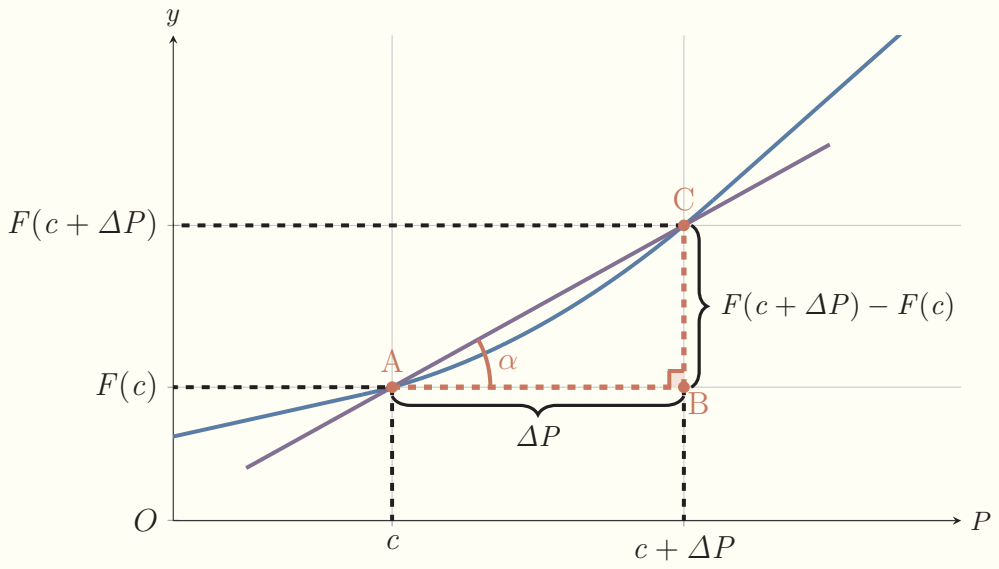


Figure 2.17: Secant line

In *Figure 2.17*, we choose two points A and C on $F(P)$ with their respective coordinates $(c, F(c))$ and $(c + \Delta P, F(c + \Delta P))$. We then draw a straight line passing the two chosen points. Those points are also the intersections of the curve $F(P)$ and the drawn line, called the secant line [26].

From point A, we draw a red horizontal dashed line. That line and the secant line has an angle α between them. This angle indicates how steep the secant line is with respect to the horizontal. The steepness of a line, however, is not measured by an angle but a concept called the slope. Let the slope of the secant line be m_{sec} [6]. To apply this concept into our figure, we need to draw a red vertical dashed line from the point C. Our two red dashed lines intersect at B, which creates a right angle (90^o angle) because they are perpendicular to each other (horizontal and vertical). In mathematics, we have a trigonometric function that represents the slope, the tangent function (tan). In the right triangle ABC (a triangle with one right angle), the slope of the hypotenuse AC (represents the secant line) is the tangent of angle α that is calculated by dividing opposite edge BC with the adjacent edge AB thus:

$$m_{sec} = tan\,(\alpha) \;= \frac{BC}{AB} \tag{2.8}$$

For AB, we see that:

$$\begin{aligned} AB &= (c + \Delta P) - c \\ AB &= \Delta P \end{aligned} \tag{2.9}$$

And for BC:

$$BC = F\,(c + \Delta P) - F\,(c) \tag{2.10}$$

So, from (2.8), (2.9), (2.10), the slope of the secant line is:

$$m_{sec} = \frac{F\left(c + \Delta P\right) - F\left(c\right)}{\Delta P} \tag{2.11}$$

In the next step, we want to observe what happens to $c + \Delta P$ when ΔP approaches 0. We can do that by applying the formula of the limit in *Eqn.* (2.7):

$$\lim_{\Delta P \to 0} c + \Delta P \ = c \tag{2.12}$$

We know $c + \Delta P$ is the x-coordinate of C and c is the x-coordinate of A, and from what we have in *Eqn.* (2.12), as $c + \Delta P$ gets smaller (because ΔP decreases), C slides to the left on the curve $F(P)$ until $c + \Delta P$ approaches c (because ΔP approaches 0) that the distance between two points C and A becomes infinitesimal and we can see roughly one point A only.

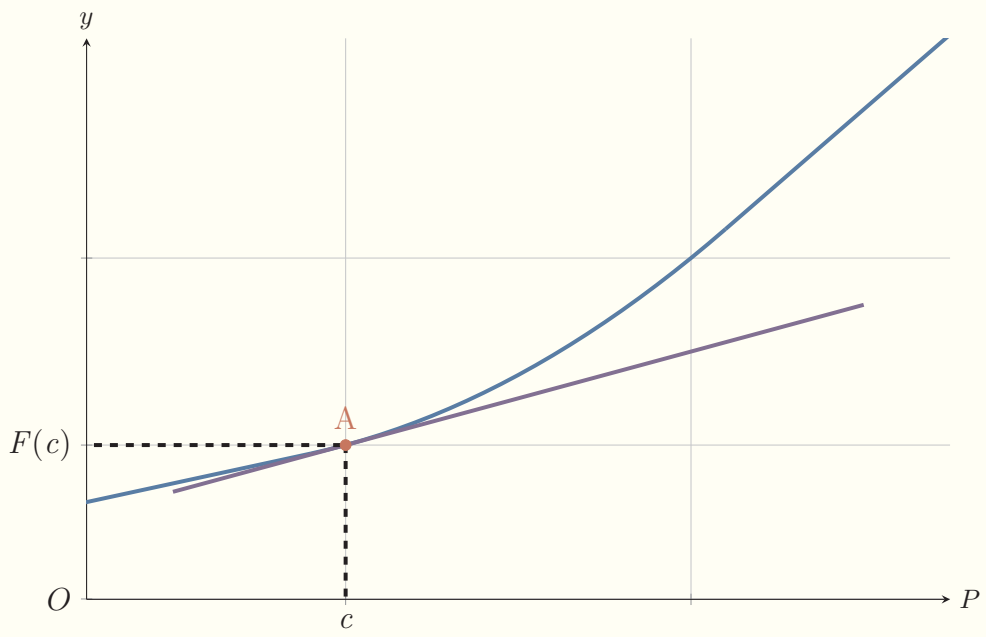


Figure 2.18: Tangent line

Now, we cannot see the secant line cutting the curve at two points explicitly anymore, instead, that line only touches the curve at a point, we call it the tangent line (*Figure 2.18*) [26]. From this idea and the concept of slope, the initial slope of the secant line (2.11) becomes the slope of the tangent line m_{tan} when ΔP approaches 0, expressed by the limit:

$$\begin{aligned} m_{tan} &= \lim_{\Delta P \to 0} m_{sec} \\ m_{tan} &= \lim_{\Delta P \to 0} \frac{F\left(c + \Delta P\right) - F\left(c\right)}{\Delta P} \end{aligned} \tag{2.13}$$

We have just assessed only the tangent line at one point c of the curve $F(P)$, but the idea applies to all points on the curve, thus we have *Definition 2.13*:

Definition 2.13: Derivative

$F'(P)$, the derivative of $F(P)$, is the slope of $F(P)$'s tangent line at P. It is expressed by the mathematical definition (only if the limit in the expression exists, as explained in *Section 2.7*):

$$F'(P) = \lim_{\Delta P \to 0} \frac{F(P + \Delta P) - F(P)}{\Delta P} \tag{2.14}$$

Note that when we say the tangent line touches the curve at one point, it is just an approximation of two points getting infinitesimally close to each other, because mathematically, if they merged into exactly one point, $\Delta P = 0$ in the denominator of (2.11) would make our tangent line undefined.

2.9 Approximation of Area

The graph of the function $F'(P)$ in *Eqn.* (2.14) is surprisingly our aggregated function $\mu_{\widetilde{P\_agg}}(P)$ (*Figure 2.11*). In the previous section, we saw the integral of $\mu_{\widetilde{P\_agg}}(P)$ is $F(P)$ (*Figure 2.16*), and now we know the derivative of $F(P)$ (function $F'(P)$) is $\mu_{\widetilde{P\_agg}}(P)$, so these two notions are the inverses of each other. More formally and mathematically, the definition of integral is given by *Definition 2.14* [9]:

Definition 2.14: Mathematical Definition of Integral

The mathematical definition of integral is given by the following expression (only when the limit in this expression exists, as explained in *Section 2.7*):

$$\int_a^b F'(P)\ dP = \lim_{n \to \infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P \tag{2.15}$$

Just like when we were stating *Theorem 2.3* and *Theorem 2.4*, $F'(P)$ is the derivative of $F(P)$, but we can absolutely generalise this derivative in *Eqn.* (2.15) to an arbitrary function $f(P)$:

$$\int_a^b f(P)\ dP = \lim_{n \to \infty} \sum_{i=1}^{n} f(P_i) \cdot \Delta P$$

Eqn. (2.15) seems familiar to the shift that we talked about in *Section 2.5* where the right-hand side is the limit after taking the sum of infinitesimal objects and the left-hand side is the continuous integral of $F'(P)$ ($\mu_{\widetilde{P\_agg}}(P)$) with respect to variable P that gives us the area under our aggregated curve. As much as we want to see why the *"integral"* of $F'(P)$ gives us the *"area"* underneath its graph, and *"integral"* still seems to be obscure to imagine, we can portray the *"area"* effortlessly, therefore it will be our initial approach.

The shape of our aggregated curve consists of only straight lines on different intervals, so we can divide it into many polygons (e.g. rectangles and triangles) for easier calculations of those areas (because we have their areas' formulas already) and sum them up to get the total area. Nonetheless, membership functions with "curly" curves, which we may encounter in other examples, do not

have any pre-defined formulas for area calculations. For this reason, it is best to use a general approach, approximation.

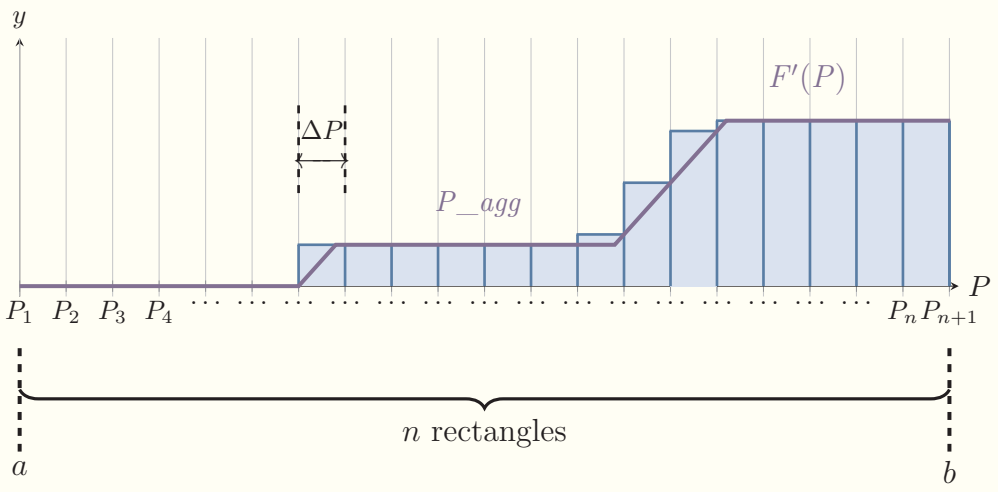


Figure 2.19: Approximation of area using rectangles

Using what we have, the area calculation formula of rectangle, to derive what we need, the area under the curve $F'(P)$, let it be S. Rectangle is a countable and discrete object, so if we want to map it into a continuous concept, we must use many rectangles. *Figure 2.19* depicts the use of rectangles to approximate S. The height of each rectangle should be equal to the height of the highest point in the interval from that rectangle's left side to right side (subinterval ΔP). There are subintervals where rectangles cover redundant areas that are not in S (above the curve), but it is because we are approximating. Later, we will see our approximation becomes more accurate. With a, b as the x-coordinates of the left-most point and right-most point of the graph respectively, the interval that contains the whole graph is $b - a$. All n contiguous rectangles share the same width ΔP, and span the whole interval of the graph, so the condition below must be satisfied:

$$\Delta P \cdot n = b - a \tag{2.16}$$

$P_1,\ P_2, \ldots,\ P_{n+1}$ are the x-coordinates indicating the vertical edges of rectangles. We start at x-coordinate $P_1 = a$, if we add 1 rectangle, its right side will land on P_2; if we add 2 rectangles, the right side of right-most rectangle will be at P_3. Following the chain, we have n rectangles, so the right side of the n^{th} rectangle is at P_{n+1}, which is also b ($P_{n+1} = b$).

We generalise the x-coordinates $P_1,\ P_2, \ldots,\ P_{n+1}$ to P_i with $1 \leq i \leq n+1$ and $i \in \mathbb{N}$. The distance between two consecutive vertical edges with x-coordinates P_i and P_{i+1} is ΔP, therefore we have a recurrence formula:

$$P_{i+1} = P_i + \Delta P \tag{2.17}$$

So far, our measurement of S is far from accurate as the redundant area is significantly large, but the real power of approximation emerges only when the number of rectangles n goes to an extremely huge value.

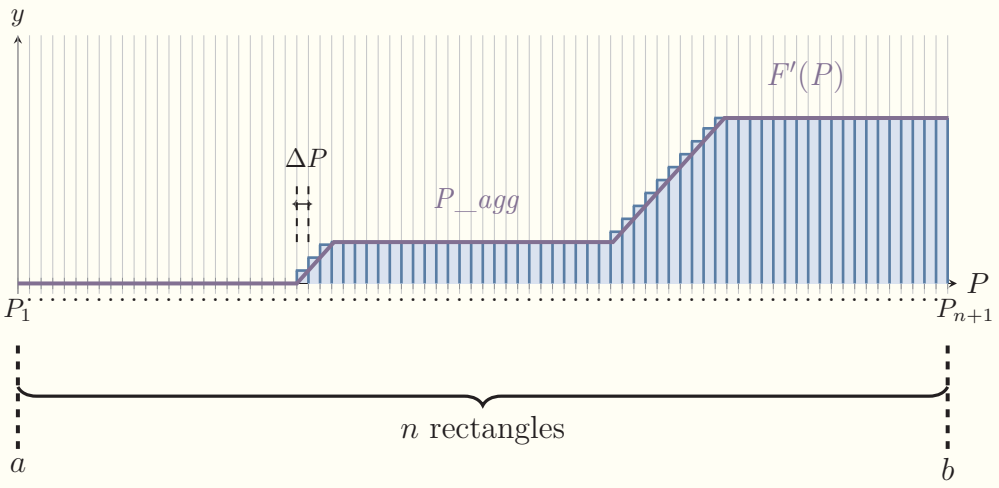


Figure 2.20: More accurate approximation

From *Eqn.* (2.16), since $b - a$ is the product of ΔP and n, and it is a constant (because a and b themselves are constants as the main interval is fixed), ΔP and n are inversely proportional; so when n increases, ΔP decreases, resulting in more rectangles with equally smaller width (*Figure 2.20*). With this, our approximation is more accurate as the redundant area is not that substantial compared to the one in *Figure 2.19*.

We want to maximise the accuracy of our approximation, and from what we have observed from *Figure 2.19* and *Figure 2.20*, this can be done by increasing n even more, until n approaches infinity. When n goes to infinity, we can assess the length of the subintervals ΔP by first, dividing both sides of (2.16) by n to isolate ΔP:

$$\Delta P = \frac{b-a}{n} \tag{2.18}$$

Then applying the limit (as in *Eqn.* (2.5) with $b - a$ as a constant):

$$\begin{aligned} \lim_{n\to\infty} \Delta P &= \lim_{n\to\infty} \frac{b-a}{n} \\ \lim_{n\to\infty} \Delta P &= 0 \end{aligned} \tag{2.19}$$

So when $n \to \infty$, we have $\Delta P \to 0$, which is illustrated through *Figure 2.21*.

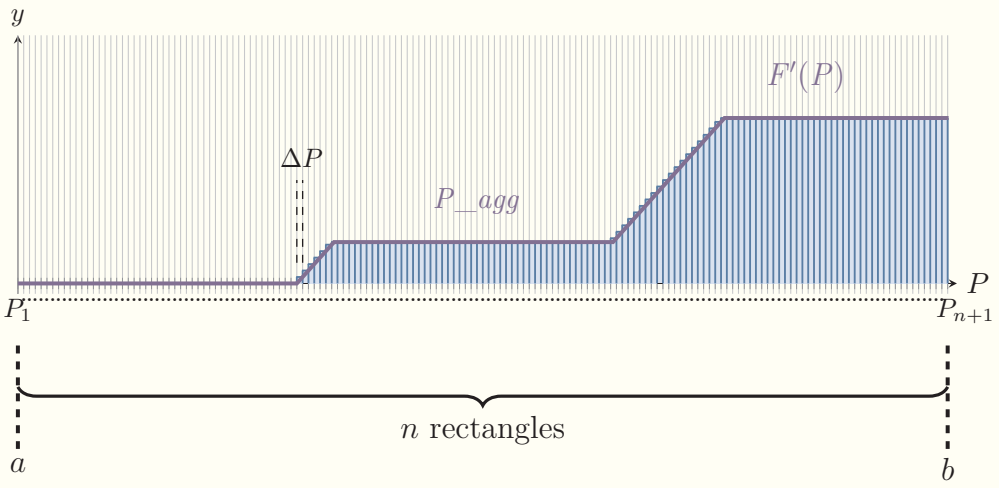


Figure 2.21: Approximation when $n \to \infty$ and $\Delta P \to 0$

Now, the redundant area has been reduced significantly which leads to the accuracy of our approximation enhanced substantially to an extent that all rectangles are almost perfectly contained under the curve.

We have seen ΔP became infinitesimal before, that was when the secant line became the tangent line, also called the derivative (*Section 2.8*). We can merge the concept in this section with the one in *Section 2.8* by mapping P_i to c (the x-coordinate of A, and that is because P_i and c are both x-coordinates), and ΔP together in both cases (because they both approach 0). Using the mapping, and *Eqn.* (2.13), we have the slope of the tangent line of $F(P)$ at P_i:

$$m_{tan} = \lim_{\Delta P \to 0} \frac{F(P_i + \Delta P) - F(P_i)}{\Delta P} \tag{2.20}$$

Not only that, the slope of the tangent line of $F(P)$ at P_i is also the derivative of $F(P)$ at P_i, or the value of $F'(P)$ at P_i, which is $F'(P_i)$, expressed by:

$$F'(P_i) = \lim_{\Delta P \to 0} \frac{F(P_i + \Delta P) - F(P_i)}{\Delta P} \tag{2.21}$$

Using *Eqn.* (2.17), we have *Eqn.* (2.21) equivalent to:

$$F'(P_i) = \lim_{\Delta P \to 0} \frac{F(P_{i+1}) - F(P_i)}{\Delta P} \tag{2.22}$$

With the formal definition of derivative, having the limit $\Delta P \to 0$ is mandatory for a general case, however, in our scenario, each subinterval ΔP is already near 0 (*Figure 2.21*), so we can omit the limit from *Eqn.* (2.22) to form a formula for our specific case only:

$$F'(P_i) = \frac{F(P_{i+1}) - F(P_i)}{\Delta P} \tag{2.23}$$

Note that P_i and P_{i+1} are the x-coordinates that represent every two nearest vertical edges, so in our context, the distance ΔP between P_{i+1} and P_i approaches 0 is the same as the distance between every two edges approaches 0 (distances between P_1 and P_2, P_2 and P_3, ..., P_n and P_{n+1} all approach 0).

2.10 Riemann Sum

We said the reason for choosing rectangles for approximation is that we have the formula to calculate the area of a rectangle already, which is the product of height and width. We already know the width of every rectangle is ΔP, so if we take the sum of all rectangles' areas, ΔP is a constant in every term of every rectangle's area formula. The height of each rectangle, however, is different, so we suppose each rectangle has the height of h_i that corresponds to its area S_i ($1 \leq i \leq n,\ i \in \mathbb{N}$). The area under the curve S is the sum of areas of all n rectangles, and is given by:

$$\begin{aligned} S &= S_1 + S_2 + \cdots + S_n \\ S &= h_1 \cdot \Delta P + h_2 \cdot \Delta P + \cdots + h_n \cdot \Delta P \\ S &= \sum_{i=1}^{n} h_i \cdot \Delta P \end{aligned} \tag{2.24}$$

Observing *Figure 2.21*, we see that the height of each rectangle h_i is roughly the value of $F'(P)$ at either its left vertical edge P_i or its right vertical edge P_{i+1}. Since P_i and P_{i+1} are infinitesimally close to each other, $F'(P_i) \approx F'(P_{i+1})$, but for simplicity, we choose $F'(P_i)$ to be the height h_i [18], therefore *Eqn.* (2.24) is equivalent to:

$$S = \sum_{i=1}^{n} F'(P_i) \cdot \Delta P \tag{2.25}$$

This is called the Riemann sum, a method for approximating the area under the curve using the areas of rectangles [18], which is typically what we have been doing.

Definition 2.15: Riemann Sum

Riemann sum adds all areas of rectangles to approximate the area under the curve S:

$$S = \sum_{i=1}^{n} F'(P_i) \cdot \Delta P$$

This approximation is inaccurate if the width of each rectangle ΔP is large (*Figure 2.19*), so we must impose a limit on *Eqn.* (2.25) when ΔP is near 0 ($\Delta P \to 0$). Not only that, but $\Delta P \to 0$ also results in $n \to \infty$ because from *Eqn.* (2.18), we have:

$$\Delta P = \frac{b-a}{n}$$

We multiply both sides by n:

$$\Delta P \cdot n = b - a$$

Then divide both sides by ΔP:

$$n = \frac{b-a}{\Delta P}$$

Next, we apply the limit on n as $\Delta P \to 0$ (using the formula (2.6)):

$$\begin{aligned} \lim_{\Delta P \to 0} n &= \lim_{\Delta P \to 0} \frac{b-a}{\Delta P} \\ \lim_{\Delta P \to 0} n &= \infty \end{aligned} \tag{2.26}$$

We can now put the limit on (2.25) as $n \to \infty$:

$$S = \lim_{n \to \infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P \tag{2.27}$$

This expression is just like the right side of *Eqn.* (2.15) that we saw at the beginning of *Section 2.9* where we defined the integral. From (2.15) and (2.27), we have:

$$S = \int_a^b F'(P)\ dP \tag{2.28}$$

Thus, up to this point, we have proved *Theorem 2.3*, that is, the definite integral from a to b of a function $F'(P)$ does indeed give us the area S bounded by its curve and the horizontal axis from a to b, as per *Eqn.* (2.28) with the area S on the left side and the definite integral on the right side.

However, we have yet to know why we need the definite integral on the right-hand side of (2.28) and what exact calculation is performed with it. Applying the limit of a sum every time we need to calculate the area (*Eqn.* (2.27)) is cumbersome, so the definite integral (2.28) offers us a more effective way to achieve the same outcome.

Demonstrating the effectiveness of definite integral (transformation from (2.27) to (2.28)) as well as finding its exact formula is the same as proving *Theorem 2.4*. To do this, we can establish the connection between the term on the right side of (2.27) with *Eqn.* (2.23). We multiply both sides of (2.23) with ΔP:

Proof 2.4: Calculation of Area (*Theorem 2.4* Proof)

$$F'(P_i) \cdot \Delta P = \frac{F(P_{i+1}) - F(P_i)}{\Delta P} \cdot \Delta P$$

Which is then simplified to:

$$F'(P_i) \cdot \Delta P = F(P_{i+1}) - F(P_i) \tag{2.29}$$

With *Eqn.* (2.23) manipulated into *Eqn.* (2.29), the analogy is explicit between (2.27) and (2.29) (or (2.23)). In (2.27), we take the limit of the sum of the left side of (2.29), which is the same as taking the limit of the sum of the right side of (2.29):

$$\lim_{n\to\infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P = \lim_{n\to\infty} \sum_{i=1}^{n} F(P_{i+1}) - F(P_i) \tag{2.30}$$

We can break the sum on the right side of *Eqn.* (2.30) into the terms as following:

$$\begin{aligned} \lim_{n\to\infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P = \lim_{n\to\infty} \Bigg[& \Big(F(P_2) - F(P_1) \Big) + \Big(F(P_3) - F(P_2) \Big) \\ & + \cdots + \Big(\ldots - \ldots \Big) + \ldots \\ & + \Big(F(P_n) - F(P_{n-1}) \Big) + \Big(F(P_{n+1}) - F(P_n) \Big) \Bigg] \end{aligned} \tag{2.31}$$

An interesting pattern can be seen in *Eqn.* (2.31), that is every left subterm of a term is cancelled out by the right subterm of the following term. Given that, $F(P_{n+1})$ is the left subterm of the last term so there is no right subterm to cancel it out. Likewise, $-F(P_1)$ is the right subterm of the first term so it does not cancel out the non-existent left subterm of the preceding term. As a result, from (2.31), we are left with:

$$\begin{aligned} \lim_{n\to\infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P &= \lim_{n\to\infty} \Big[\Big(-F(P_1) \Big) + \Big(F(P_{n+1}) \Big) \Big] \\ \lim_{n\to\infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P &= \lim_{n\to\infty} \Big[F(P_{n+1}) - F(P_1) \Big] \end{aligned} \tag{2.32}$$

In the interval $[a, b]$, the left-most edge $P_1 = a$, and the right-most edge $P_{n+1} = b$ as said in *Section 2.9*, thus *Eqn.* (2.32) is equivalent to:

$$\lim_{n\to\infty} \sum_{i=1}^{n} F'(P_i) \cdot \Delta P = \lim_{n\to\infty} \Big[F(b) - F(a) \Big] \tag{2.33}$$

Since a and b are constants, $F(a)$ and $F(b)$ are also constants, which leads to $F(b) - F(a)$ being a constant that does not depend on n, so we can omit the limit to simplify *Eqn.* (2.33) into:

$$\lim_{n\to\infty}\sum_{i=1}^{n} F'(P_i)\cdot\Delta P \;= F(b) - F(a) \tag{2.34}$$

Through a sequence of algebraic manipulations, a bulky limit of a sum in (2.27) is truncated to just a subtraction given by the right-hand side of *Eqn.* (2.34), which is also the exact calculation that we perform to get the result of a definite integral.

We combine equations (2.27), (2.28), and (2.34) to get:

$$S = \int_a^b F'(P)\; dP = F(b) - F(a) \tag{2.35}$$

Eqn. (2.35) is our proof (*Proof 2.4*) for *Theorem 2.4*, that is the area between the curve of a function $F'(P)$ and the horizontal axis in a defined interval $[a, b]$ can be calculated by subtracting that function's integral $F(P)$ value at a from its integral value at b. This formula is called the *"Fundamental Theorem of Calculus"*.

2.11 Continuous Weighted Average

To use the formula of the centroid method, we need the numerator and the denominator of (11.1). With our proof for *Theorem 2.3* in the previous section, and looking at *Figure 2.22*, the definite integral in the denominator $\int_a^b \mu_{\widetilde{P\_agg}}(P)\cdot dP$ is the area shaded in purple and the definite integral in the numerator $\int_a^b \mu_{\widetilde{P\_agg}}(P)\cdot P\cdot dP$ is the area shaded in red.

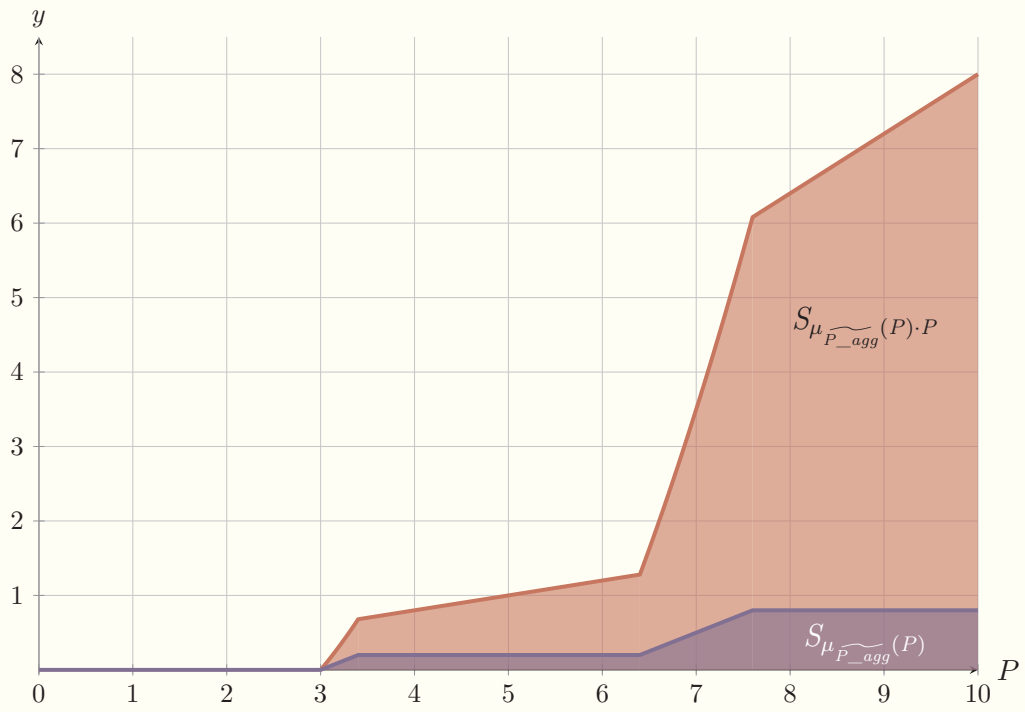


Figure 2.22: Areas of the numerator and denominator

Circle back to *Section 2.5* where we witnessed the discrete weights filling up an area (*Figure 2.14*). That area is the one under the curve of a continuous weight function (purple region in *Figure 2.22*). That is the association between the denominator of the weighted average formula ($\sum_{i=1}^{n} w_i$) and the denominator of the centroid formula ($\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP$). For the numerator of the discrete weighted average, if we multiply each weight by its corresponding x_i ($\sum_{i=1}^{n} w_i \cdot x_i$), we also have to multiply the weights function $f(P) = \mu_{\widetilde{P\_agg}}(P)$ with the elements function $f(P) = P$ for the numerator of the centroid formula, that is $\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP$. Because $f(P) = P$ is a function itself, multiplying P with the weights function $\mu_{\widetilde{P\_agg}}(P)$ is sufficient to account for every possible value of P ($P \in \mathbb{R}$) in the interval $[a, b]$ instead of adding products like in the discrete weighted average. We can interpret the centroid formula as the continuous weighted average of all possible values in the domain, consequently, it satisfies *Theorem 2.5*. This can be proved using the similar approach as we did with the discrete weighted average. We have our initial centroid formula (11.1) as follows:

Proof 2.5: Continuous Weighted Average (*Theorem 2.5* Proof)

$$P^* = \frac{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}$$

Since the definite integrals in the numerator and denominator are the areas under their corresponding curves from a to b, we can split the whole area into numerous small areas then add them up. This is similar to splitting the interval $[a, b]$ into infinitesimal subintervals $[a, c], [c, d], \ldots, [e, b]$ (with $a < c < d < \cdots < e < b$), so the numerator becomes:

$$\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP + \int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP + \cdots + \int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP \quad (2.36)$$

The subintervals are infinitesimal that the x-coordinates P do not change substantially, so P in each subinterval can be considered as a constant in that subinterval:

$$\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot P_{ac} \cdot dP + \int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot P_{cd} \cdot dP + \cdots + \int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot P_{eb} \cdot dP \quad (2.37)$$

$P_{ac}, P_{cd}, \ldots, P_{eb}$ are constants in subintervals $[a, c], [c, d], \ldots, [e, b]$ respectively.

Considering the first term of *Eqn.* (2.37), but without the constant P_{ac}, this is the area under the curve $\mu_{\widetilde{P\_agg}}(P)$ from a to c:

$$\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP \quad (2.38)$$

Now we multiply the whole function $\mu_{\widetilde{P\_agg}}(P)$ by the constant P_{ac} again, so at every x-coordinate the value of the graph increases by P_{ac} times, this makes the whole graph stretches by P_{ac} times, hence the area under its curve from a to c is also multiplied by P_{ac}. The following equation expresses our description, with the left side as the area of the initial function after being increased

by P_{ac} times (like what we have done initially), and the right side as the initial area multiplied by P_{ac} (what we have inferred):

$$\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot P_{ac} \cdot dP = P_{ac} \cdot \int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP \tag{2.39}$$

What we just did can be summarised as *"taking the constant out of an integral"*. We can do that for other terms in (2.37) to get:

$$P_{ac} \cdot \int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP + P_{cd} \cdot \int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot dP + \cdots + P_{eb} \cdot \int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot dP \tag{2.40}$$

Subsequently, we divide each term of the numerator (2.40) by the denominator of (11.1):

$$\frac{P_{ac} \cdot \int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + \frac{P_{cd} \cdot \int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + \cdots + \frac{P_{eb} \cdot \int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} \tag{2.41}$$

With further manipulations, (2.41) is equal to:

$$P_{ac} \cdot \frac{\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + P_{cd} \cdot \frac{\int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + \cdots + P_{eb} \cdot \frac{\int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} \tag{2.42}$$

The elements P_{ac}, P_{cd}, P_{eb} have their respective degrees of contribution:

$$\frac{\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}, \quad \frac{\int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}, \quad \frac{\int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}$$

Also, each degree is a percentage, so the sum of all degrees should be 1:

$$\frac{\int_a^c \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + \frac{\int_c^d \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} + \cdots + \frac{\int_e^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} = 1 \tag{2.43}$$

With the common denominator, we can add all the terms of all numerators, which is simply the sum of infinitesimal areas in subintervals $[a, c]$, $[c, d]$, $\ldots$, $[e, b]$ that made up the whole area from a to b:

$$\frac{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP} = 1 \; (satisfied)$$

Thus, *Theorem 2.5* is proved.

All the proofs in our journey (*Section 2.6*) have brought us to a confirmation that the centroid method determines the most optimal value P^* from all values $P \in \mathbb{R}$ in the domain of aggregated membership function by considering the degree of truth of every value in percentage that contributes to 100%.

2.12 Applying the Centroid Method for Defuzzification Result

After we have proved all needed concepts, we can apply the centroid formula into calculating the result of defuzzification, the final and indispensable part of a complete fuzzy inference process.

Using the centroid formula (11.1):

$$P^* = \frac{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot P \cdot dP}{\int_a^b \mu_{\widetilde{P\_agg}}(P) \cdot dP}$$

For simplicity, we rename our functions as follows:

$\mu_{\widetilde{P\_agg}}(P)$ is changed to $D'(P)$ (denominator)

$\mu_{\widetilde{P\_agg}}(P) \cdot P$ is changed to $N'(P)$ (numerator)

$$P^* = \frac{\int_a^b N'(P) \cdot dP}{\int_a^b D'(P) \cdot dP} \tag{2.44}$$

In $D'(P)$ and $N'(P)$, the prime symbol indicates that these functions are treated as derivatives.

The definite integrals in both the numerator and denominator can be interpreted as per our proved *Eqn.* (2.35) in *Section 2.10*:

$$\int_a^b F'(P)\ dP = F(b) - F(a)$$

Starting with the denominator, $D'(P)$ is in the form $F'(P)$ (derivative), which needs to be integrated into $F(P)$. $D'(P)$ is the aggregated membership function from *Section 2.2*:

$$D'(P) = \mu_{\widetilde{P\_agg}}(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.5P - 1.5 & if\ 3 < P \leq 3.4 \\ 0.2 & if\ 3.4 < P \leq 6.4 \\ 0.5P - 3 & if\ 6.4 < P \leq 7.6 \\ 0.8 & if\ 7.6 < P \leq 10 \end{cases}$$

Using the integral formulas from [1], we establish $D(P)$ as the integrated function of $D'(P)$:

$$D(P) = \begin{cases} 0 + C_1 & if\ 0 \leq P \leq 3 \\ 0.25P^2 - 1.5P + C_2 & if\ 3 < P \leq 3.4 \\ 0.2P + C_3 & if\ 3.4 < P \leq 6.4 \\ 0.25P^2 - 3P + C_4 & if\ 6.4 < P \leq 7.6 \\ 0.8P + C_5 & if\ 7.6 < P \leq 10 \end{cases}$$

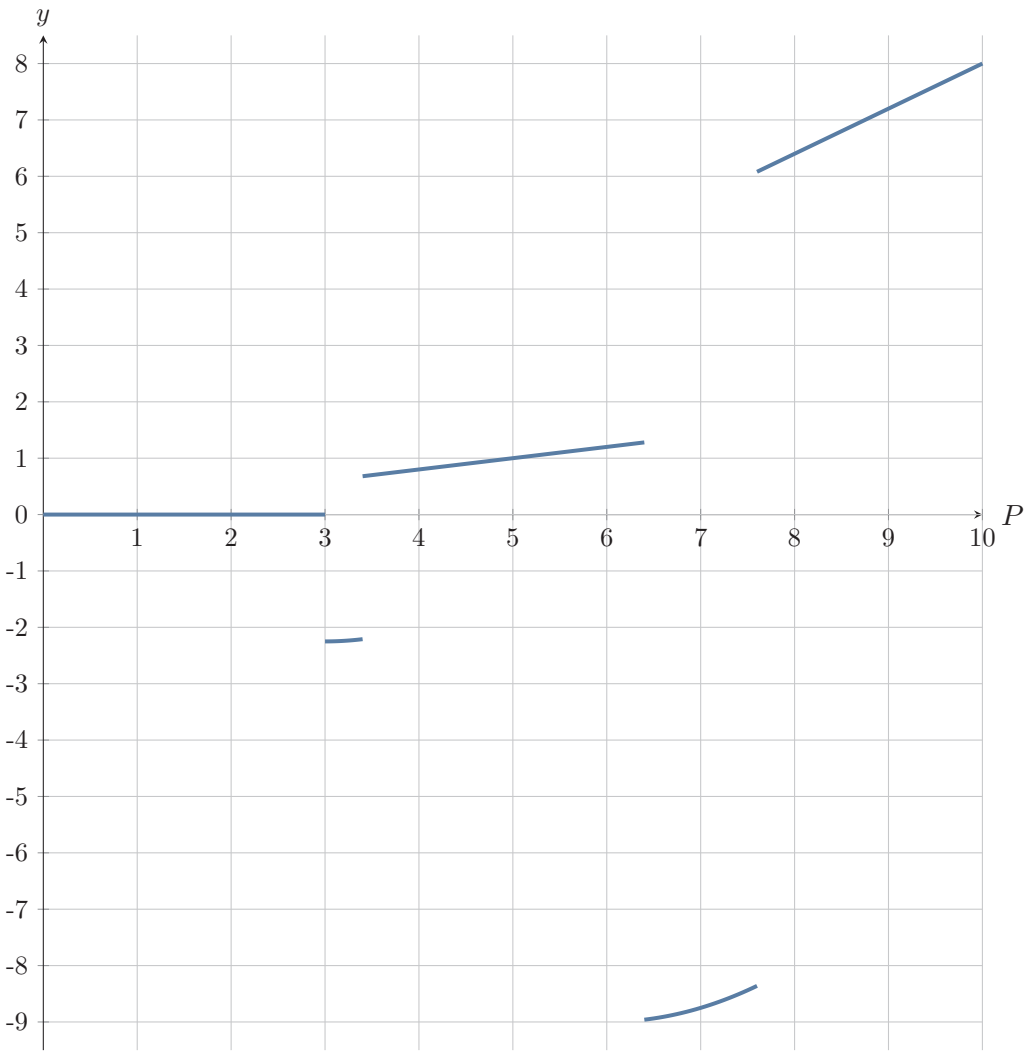


Figure 2.23: Graph of undetermined constants

Our graph looks like *Figure 2.23* if the constants $C_1, C_2,$ *etc.* are set to 0. We need to identify the constants to make our function continuous:

$$D(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.25P^2 - 1.5P + 2.25 & if\ 3 < P \leq 3.4 \\ 0.2P - 0.64 & if\ 3.4 < P \leq 6.4 \\ 0.25P^2 - 3P + 9.6 & if\ 6.4 < P \leq 7.6 \\ 0.8P - 4.84 & if\ 7.6 < P \leq 10 \end{cases}$$

Remarkably, $D(P)$ is also the function $F(P)$ in *Figure 2.16*. We said that it is the integral of $\mu_{\widetilde{P\_agg}}(P)$, which has just been confirmed with our recent calculation.

We move on to the numerator:

$$N'(P) = \mu_{\widetilde{P\_agg}}(P) \cdot P = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ 0.5P^2 - 1.5P & if\ 3 < P \leq 3.4 \\ 0.2P & if\ 3.4 < P \leq 6.4 \\ 0.5P^2 - 3P & if\ 6.4 < P \leq 7.6 \\ 0.8P & if\ 7.6 < P \leq 10 \end{cases}$$

Integrate $N'(P)$ into $N(P)$ using the same approach for the denominator:

$$N(P) = \begin{cases} 0 & if\ 0 \leq P \leq 3 \\ \frac{1}{6}P^3 - 0.75P^2 + 2.25 & if\ 3 < P \leq 3.4 \\ 0.1P^2 - 1.0253 & if\ 3.4 < P \leq 6.4 \\ \frac{1}{6}P^3 - 1.5P^2 + 20.82 & if\ 6.4 < P \leq 7.6 \\ 0.4P^2 - 15.7613 & if\ 7.6 < P \leq 10 \end{cases}$$

Now that we have the integrated functions $D(P)$ and $N(P)$, we can substitute the values $a = 0$ and $b = 10$ in and calculate P^*:

$$P^* = \frac{\int_0^{10} N'(P) \cdot dP}{\int_0^{10} D'(P) \cdot dP}$$

$$P^* = \frac{N(10) - N(0)}{D(10) - D(0)} \tag{2.45}$$

$$P^* = \frac{24.2387 - 0}{3.16 - 0}$$

We can sketch $N(P)$ and $D(P)$ on *desmos* to verify our calculations (*Figure 2.24*). The blue curve is $D(P)$ (integral of the purple curve $D'(P)$) and the green curve is $N(P)$ (integral of the red curve $N'(P)$).

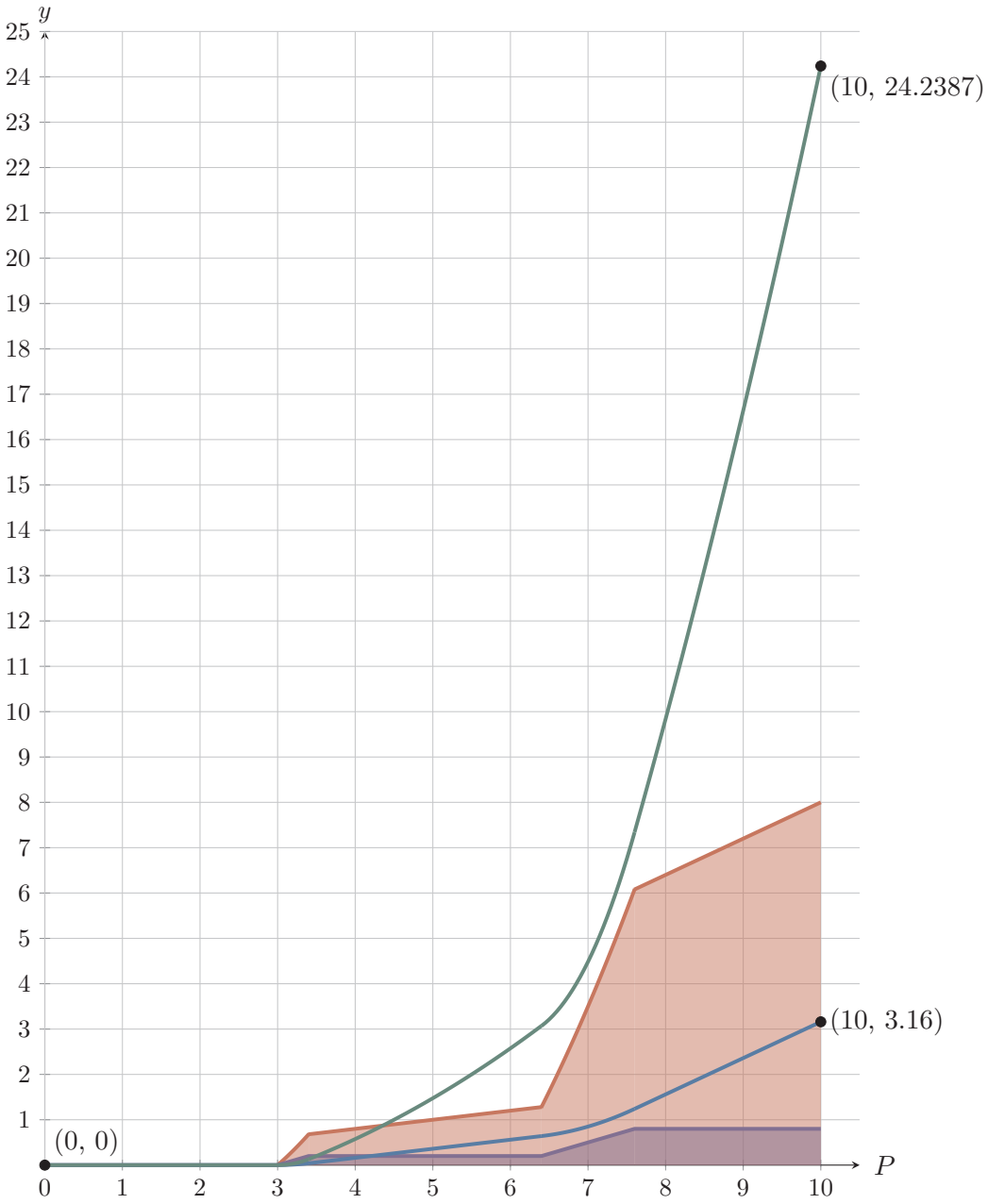


Figure 2.24: Results of definite integrals

From either (2.45) or *Figure 2.24*, our result is:

$$P^* \approx 7.67047$$

Since we have been using the ratio $1:100$ (as mentioned in *Section 2.2*), the output should be $P^* \approx 767.047W$.

Hence, the fuzzy inference process has determined that for temperature 12^oC and humidity 40%, the most optimal heater power is $767.047W$.

2.13 Other Applications

The centroid method has an abundance of uses besides AI systems like the thermostat that automatically adjusts the heater power in our worked-through example.

The weighted average (centroid in the discrete world) appears in many education providers' grading systems in which Deakin University is a prime example. Deakin uses Weighted Average Mark (WAM) to assess students' academic performance [2]. A student achieved a mark of 80 in a unit with 2 credit points and a mark of 73 in a one-credit-point unit has a provisional WAM of roughly 77.7 $\left(\frac{80\cdot 2+73\cdot 1}{3}\right)$. This essentially says that units with more credit points are more important and thus should be considered with larger degree of importance.

Centroid in the continuous world, as mentioned, can also be called the center of gravity, and this is closely related to physics and engineering. For instance, when designing a car, engineers must ensure that the car's center of gravity is near the center of the car [4] to maintain stability and safety since it is the point that makes the force of gravity pulls evenly on every part of the car.

2.14 Relations between Discrete Mathematics and Continuous Mathematics

Mathematics can be divided into *"discrete mathematics"* and *"continuous mathematics"*. Discrete mathematics, as the name suggests, is the study of separable objects such as Boolean algebra, counting, graph theory, and many more, while continuous mathematics is about continuity with limits, real numbers, approximations, etc [19].

Given the boundary between the two fields, we established some intriguing connections for them during our journey. We saw how discrete weighted average filled up a continuous notion, the area. In the continuous world, we encountered the limit that we used to later prove that an infinite number of discrete rectangles can form a continuous area (*Theorem 2.3* and *Theorem 2.4*). In the final proof for the centroid method (*Theorem 2.5*), we took a reverse approach where we mapped the continuous world to the discrete world by separating a continuous interval into infinitesimal discrete subintervals to observe and rigorously prove why the centroid method is the continuous version of weighted average that takes all real values into account. It is fascinating to see that even though *"discrete"* and *"continuous"* seem to be the contrast of each other, they are surprisingly interconnected, with each capable of underpinning the other.

2.15 Conclusion

To recap, we have worked through a fuzzy inference process using a concrete example from the beginning. When we got to the last stage, defuzzification, we exhibited our main concept for this paper, the centroid method. We began the proof for the method by building our intuition for the discrete weighted average in which we demonstrated two important theorems, which are *"Theorem 2.1: Every element equally contributes a percentage of its value to the whole in the average calculation"* and *"Theorem 2.2: In the calculation of weighted average, a degree of contribution of an element is a percentage which that element contributes to the whole (degrees can be equal or not equal for distinct elements), and adding all degrees will give us the percentage of the whole, represented by* 1*"*. Thereafter, we entered the world of continuum. From here, we discovered what a limit is, how it is used alongside the secant line and the tangent line to form the definition of derivative. Subsequently, we approximated the area under the curve utilising the area formula of rectangles, and from that, by combining the Riemann sum with the support of the limit and the proved definition of derivative, we confirmed our proofs for *Theorem 2.3* (*The definite integral from a to b of a function is the area bounded by its curve and the horizontal axis from a to b*) and *Theorem 2.4* (*The area bounded by the curve of a function and the horizontal axis in a defined interval* $[a, b]$ *can be calculated by subtracting that function's integral value at*

a from its integral value at b). We completed our mathematics discussion by demonstrating *Theorem 2.5* (*In continuous weighted average, a degree of contribution of an element (in infinite number of elements) can be interpreted as a percentage which that element contributes to the whole (degrees can be equal or not equal for distinct elements), and the sum of all degrees is the percentage of the whole, represented by* 1) after linking from the discrete weighted average to the continuous weighted average. We thereby made a final confirmation on the intrinsicality of the centroid method that it identifies the most ideal value from all real values in the domain of aggregated membership function by considering the degree of truth of every value in percentage that contributes to the whole. Moreover, we demonstrated that the centroid method can actually defuzzify values with the calculation of the output of fuzzy inference system from the example prior to the proofs. We also looked at some other applications of the centroid method, and eventually, enclosed our journey by reflecting on the beautiful intertwinement of discrete and continuous mathematics.

Context

This article was adapted from the final HD report submitted in SIT192 - Discrete Mathematics.

About the Author

I am Tri Khuong Nguyen, a first-year undergraduate student who is currently pursuing a Bachelor's degree in Artificial Intelligence.

With an immense passion and dedication for mathematics since high school, it was an opportunity for me in SIT192 to delve deeper into a concept that I encountered but did not have much time for in year 12, Area by Integration. Besides the proofs that may seem to be too theoretical, I also accounted for the practical aspect with a realistic scenario utilising AI (which is what I have always loved). Gathering all of these motivations, this report was born, exploring Fuzzy logic, a crucial part of AI systems' human-like reasonings, extending binary logic in discrete mathematics to the continuous world with the Centroid formula and the Area by Integral.

Apart from AI, computer science, maths, and physics, I would love to be immersed in music, to play football and video games with my pals after long hours of study.

Acknowledgements

I would like to give a big thank you to Lashi Bandara for not only assisting me in modifying and finalising the report but also sharing a lot of experience in the field of teaching and researching, which perhaps will be the path that I aim for in the upcoming years. He also reached out to me with an invitation to feature my report in this Mathematics Yearbook, a quite remarkable feat for me personally as a first-year undergraduate.

References

All figures created using Tikz [24].

[1] Integration formulas. *GeeksforGeeks*, 2022. https://www.geeksforgeeks.org/integration-formulas/.

[2] Weighted average mark. *Deakin University*, 2023.

[3] Fuzzy logic | set 2 (classical fuzzy sets). *GeeksforGeeks*, 2023. https://www.geeksforgeeks.org/fuzzy-logic-set-2-classical-fuzzy-sets/.

[4] Real life applications of center of gravity. *GeeksforGeeks*, 2024. https://www.geeksforgeeks.org/real-life-applications-of-center-of-gravity/.

[5] Average (definition). *Cambridge Dictionary, Cambridge University Press and Assessment*, 2025.

[6] Slope formula. *GeeksforGeeks*, 2025. https://www.geeksforgeeks.org/slope-formula/.

[7] D. Baker and W. Haynes. Centroids. In *Engineering Statics - Open and Interactive*. 2025. https://engineeringstatics.org/Chapter_07-centroids.html.

[8] M. Reha Civanlar and H. Joel Trussell. Constructing membership functions using statistical data. *Fuzzy Sets and Systems*, 18(1):1–13, 1986.

[9] P. Dawkins. Definition of the definite integral (Online notes). *University of Texas, Department of Mathematics*, 2024.

[10] P. Dawkins. The definition of the limit (Online notes). *Paul's Online Math Notes*, 2024.

[11] J. Frost. Weighted average: Formula and calculation examples (Online article). *Statistics by Jim*, 2023.

[12] A. Ganti. Weighted average: Definition and how it is calculated and used (Online article). *Investipediea*, 2024.

[13] Z.X. Guo and W.K. Wong. Fundamentals of artificial intelligence techniques for apparel management applications. In W.K. Wong, Z.X. Guo, and S.Y.S. Leung, editors, *Optimizing Decision Making in the Apparel Supply Chain Using Artificial Intelligence (AI)*, Woodhead Publishing Series in Textiles, pages 13–40. Woodhead Publishing, 2013.

[14] J. Nescolarde-Selva J. Uso-Domenech and H. Gash. Universe of discourse and existence. *Mathematics*, 6(11):272, 2018.

[15] H. Lamaazi and N. Benamar. OF-EC: A novel energy consumption aware objective function for RPL based on fuzzy logic. *Journal of Network and Computer Applications*, 117:42–58, 2018.

[16] MathWorks. Foundations of fuzzy logic. *MATLAB Documentation*, 2025.

[17] MathWorks. Fuzzy inference process, 2025.

[18] L. A. Oberbroeckling. Numerical integration. In *Programming Mathematics Using MATLAB*, pages 183–191. Academic Press, 2021.

[19] S. Oliveira and D. J. Stewart. Discrete and continuous. In *Building Proofs*, pages 63–111. World Scientific, 2015.

[20] Tekla S. Perry. Lotfi Zadeh and the birth of fuzzy logic. *IEEE Spectrum*, 32(6):26–29, 1995.

[21] J. Redden. Solving absolute value equations and inequalities. In *Advanced Algebra*. 2012. https://2012books.lardbucket.org/books/advanced-algebra/.

[22] J. Ren, Y. Man, R. Lin, and Y. Liu. Multicriteria decision making for the selection of the best renewable energy scenario based on fuzzy inference system. In Jingzheng Ren, editor, *Renewable-Energy-Driven Future*, pages 491–507. Academic Press, 2021.

[23] M. Smithson and J. Verkuilen. *Fuzzy Set Theory*. SAGE Publications, 2006.

[24] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[25] B. Uzun, I. Ozsahin, V. O. Agbor, and D. Uzun Ozsahin. *Theoretical aspects of multi-criteria decision-making (MCDM) methods*. Academic Press, 2021.

[26] Eric W. Weisstein. Secant line. *MathWorld–A Wolfram Web Resource*, 2025.

3

On the Turing machine and the undecidability of the halting problem

Ari Robin

Abstract

In this paper, we explore the concept of Turing machines, introduced by Alan Turing in 1936. Turing machines are fundamental models of computation that help us to understand what is and is not possible to compute. We will look at how Turing addressed the limitations of previous computational models, along with the origins and developments leading to the Turing machine concept. To demonstrate its functionality, we will design a Turing machine to recognize palindromes, highlighting its components and how it works. We will then explore the halting problem, its implications for demonstrating the limits of computation, and why it is still relevant today. Finally, we will explore Turing completeness, practical real-world applications of Turing machines, and the Church-Turing thesis. By understanding Turing machines, we gain insight into what can be computed today and set the stage for further exploration into computational theory and its modern real-world applications.

3.1 Introduction to Turing Machines

In 1936, before computers became a reality, Alan Turing, a British mathematician and logician, devised a way to represent how a machine could compute any mathematical statement that a human could also calculate, given enough time and memory (see [7] for an introduction to Turing machines as a computational model). This model of computation is foundational to computer science as it provides a way for us to understand what can and cannot be computed, underlying all modern computing systems.

In short, Turing Machines are an abstract machine that can solve problems by simulating any algorithm, given enough time. Turing Machines are analogous to how physical computers function today. They consist of an 'infinitely long tape' sectioned into 'cells', with each cell containing a 'symbol', like a number, letter or a symbol indicating the cell is blank. A 'tape head' is able to move along each cell, back and forth, one cell at a time, while reading or writing each cells symbol. As it does this, the machine transitions to different 'states', with the logic behind which states are transitioned to being predefined by rules known as the 'transition function'. This determines what the tape head does based on the symbol it reads, and which direction it moves in along the tape. Although this may seem trivial, given enough tape and time, this model can compute anything that 'is computable'. There are some problems that no computer can solve, and the exploration of Turing Machines gives us insight into how some of these problems are known as 'unsolvable' or 'undecidable'.

We're going to explore how that is possible, and how Alan Turing's insight into computation enabled the computers that we use today.

Entscheidungsproblem

In 1928, David Hilbert posed the *Entscheidungsproblem*, asking if there was an algorithmic way to determine if any mathematical proposition was true or false [3]. Kurt Gödel responded with a partial answer to this question in 1931 in the form of his Incompleteness Theorems, proving that no complete system could satisfy this requirement, as his Incompleteness Theorems state that there are always some truths within a logical system that cannot be proved within the system [5]. This led Alan Turing to conclusively address Hilbert's problem by introducing the conceptual model of a Turing machine, which, among many other things referred to in this module, demonstrated that there does not exist a universal computational method to solve all possible mathematical problems, showing the impossibility of solving Hilbert's *Entscheidungsproblem* [7].

3.2 Theoretical and Practical Explanations

We now explore the theoretical computational models of computation before Turing Machines, and the necessity of the creation of the Turing Machine model, before looking deeper into the actual workings of a Turing Machine and its components.

The Need for a More Powerful Computational Model

Before Alan Turing devised the Turing machine concept, there were other computational models that had been conceived, but these models lacked the capability to handle many different forms of computation due to having limited computational power [4]. These previous computational models, such as the finite automata and pushdown automata models, contained many restrictions on the computation they could perform due to the limits on their structural capabilities.

Finite Automata and Pushdown Automata

Finite automata could not count beyond a fixed limit due to containing a certain number of finite states. The model was conceived to recognize patterns within the input of strings, but was based on a finite set of states and transitions, with the key restriction being that their input could only be processed from left to right without the ability to go backward (see [8] for an overview of automata).

Pushdown automata built on the concept of finite automata by incorporating a stack. Unlike finite automata, pushdown automata could use their stack to store and retrieve data, meaning they had a form of memory. Even though this was an improvement, their memory was still a last-in-first-out model (LIFO), meaning that randomly accessing memory was not possible [11]. Seeing as pushdown automata could only manipulate the top element on their stack, more complex conditional statements for processing data were not possible.

Lack of Computational Power

Importantly, neither of these models could modify their input during processing and only processed input in one direction without the ability to go backwards. Although there are more, these two key restrictions meant that these models were limited to a very strict number of tasks they could perform.

In contrast, Turing machines provide a model that can modify their input and contain the ability of bidirectional movement, as the machine's *tape head* could move *left or right*, being able to traverse its input forward and backward [4]. This meant that the machine had randomly accessible memory by being able to write to its infinitely long tape wherever it chose and could have as many elements in its set of transitions as it required.

A More Complex Requirement Example by Recognizing Palindromes

A classic test of this more complex ability that other models were not suited for before Turing machines was recognizing a palindrome, being a sequence of characters or numbers that reads the same forward and backward [8]. The finite automata model couldn't do this, as it had no memory to be able to store and compare the characters of the string it was inspecting for being a palindrome (its input), and could only process the input in one direction, meaning that it could not go back and check if the back half of the input was the same as the front half. The pushdown automata couldn't either as, although it could store the symbols of the input in its memory, the LIFO structure of the data meant symbols couldn't be compared and accessed randomly, leading to inspecting and determining an input to be a palindrome impossible.

With its infinitely long tape and ability to move the tape head in both directions, the Turing machine model was able to inspect and compare both ends of the input string by comparing and storing these comparisons to be true or false, enabling the completion of the task the other models found impossible [7].

Predicting Halting

As another example, Turing machines could solve the halting problem for both finite automata and pushdown automata. We will have a deeper look at the Halting Problem shortly. When a computational model finishes processing its input (and achieves a result at the end of its computation) the machine stops processing data. This is known as *halting* and will be explored later in this module. Machines may also never halt and instead run forever, indicating an impossibility or paradox (such as an infinite loop) that a computational model can never solve, even when given infinite memory (or tape in the context of Turing machines) and time [5]. The Turing machine's ability to determine if these earlier computational models would halt or continue running indefinitely is an ability that the finite and pushdown automata models could not do themselves, further proving Turing machines to be the superior computational model.

Using Turing Machines to Solve Problems

Just like a modern-day computer, a Turing machine can be *programmed* to perform computation that is useful to us. Unlike the previous computational models mentioned, a Turing machine can compute and perform any mathematical task that a human could compute, given enough time and memory ([7]). Today, our computers are programmed to solve problems for our needs. In the same way, we can define the functionality of a Turing machine to solve a problem or perform computations.

Designing a Specific Machine

Let's design a Turing machine that can recognize a palindrome, and inspect each component of the machine as we go. As a Turing machine is capable of computing anything that a human can compute, it needs several components to be defined within the model (see [9] for specific Turing Machine components).

Components We'll Need

To start, we have an infinitely long amount of tape with individual cells that serves as the machine's memory. The symbols on the tape are manipulated by the tape head, that reads and writes symbols on each cell (usually 0s, 1s, and Bs representing blank cells). These symbols can be different than just these symbols though and are defined in our machine's tape alphabet, including an explicit definition of what symbol will represent a blank cell (which is usually 'B'). This particular alphabet for our palindrome will be the lowercase English alphabet a–z, as well as our blank symbol B [11]. While doing this, we should also define our input alphabet, which will

be the same as our tape alphabet in our example but excluding the symbol representing blank. The tape where our machine starts is known as the *initial tape*, and the input string for our machine will be on this tape with each of the symbols (or letters/digits) from the string placed in each cell of the tape from left to right. For our machine, the input tape will contain the string that is going to be determined to be palindromic, or not.

Next, we need to define the behaviour of our machine to achieve the functionality of determining an input to be palindromic or not. This will inform the machine on how it should operate given an input, the current state of the tape, and the position of the tape head. To do this, we need to define the initial state of the machine before it starts processing the input, all the different possible states of the machine, and the set of transition functions (that can also be referred to as the program of the machine). Included in these states, we should also define the state the machine will be in when indicating the computation has completed by terminating successfully to indicate the machine has recognized a palindrome. This successfully terminating state will be known as the machine's final or accepting state. This accepting state will be when the machine has checked that each front letter of the current tape state matches the one at the back. These transition functions will tell our Turing machine what symbol to write, which state to transition into, and which direction it should move the tape head. The tape head can only move one symbol to the left, or the right, or stay in place, for each transition function.

Defining Functionality

While defining our transition functions and states, we need to specifically define what state the machine should transition to if it recognizes the input as a palindrome or determines it is not palindromic. We can define a set of states (that is a subset of our existing set of all the different possible states of our machine) that will be our final states. The machine will halt at this state, indicating it has finished its computation.

So, in summary, our Turing machine will need to contain:

1. Set of all different states the machine can be in.
2. The tape alphabet, a total set of symbols read/writable within our machine (including the blank symbol).
3. The definition of the symbol representing a blank cell.
4. The input alphabet, a set of symbols that can be given as input (same as tape alphabet, without blank symbol).
5. Set of transition functions.
6. Initial state definition of the machine, also contained within the set of all different states.
7. Set of final states to halt on, indicating the computation has finished, also contained within set of total states.

We can define and demonstrate the functionality of our Turing machine before creating tape diagrams to demonstrate its functionality.

Defining Our Machine's Operation

This machine will read the first symbol on the tape and mark it as a B, before moving to the end of the input string and checking that the final symbol of the input matches the first. It will then mark this as a B, before returning to the start of the tape. It will then move to the first character on the tape that is not a B (as the remaining part of the input not overwritten by a B) and will repeat the checking process. If at any point the first and last letters of the remaining input do

not match, the machine does not have any transition functions left to continue operation. Seeing as the only way to transition to the accepting state q_2 is after reading two 'Bs', which is only possible after every letter is checked and marked as matching with a 'B', the machine will halt in a non-final (or accepting) state indicating that the word is not palindromic. If the final state is q_2, the word is palindromic (see [7] for information into mathematical definitions of Turing Machines).

In order to remember which letter the machine needs to check for, each symbol within the input alphabet will correspond to a different state within the set of all states. This way, if the computer reads the letter 'a' as the first letter it will transition into q_a. When in state q_a, the machine will know to look for the letter 'a' in our example once reaching the end of the tape. If the letter 'a' is found at the end of the tape while in state q_a, execution will continue. If the letter 'a' is not found, the machine will halt, and seeing as the tape is not blank (and all letters have not been marked with a 'B' indicating they have been checked), the machine has decided the input was not palindromic.

For every symbol in the input alphabet (Σ), there will be a corresponding state. If the letter 'b' is in the input alphabet, the state q_b will exist within Σ.

States and Transitions in Use

Before we can construct our palindrome-checking machine and provide an example, we need to be able to define the states and transition functions of the machine. We also need to formally define the mathematical representation of a Turing machine so we can understand how to properly construct one. We will then show examples of these states and transitions and what they are used for when constructing our new machine.

Set of States

The set of states in a Turing machine hold the various conditions the machine can hold at any step of its operation, and using transitions, which state the machine is in determines how the machine behaves when it encounters each symbol on the tape. A Turing machine always starts its operations at initial state of q_0 (or s_0). We can use these characteristics to create Turing machines conceptually to complete a task or requirement, as we'll see below.

Set of Transition Tuples

Sets of transitions tuples use states, symbols, and the direction of the tape head to dictate how a Turing machine will operate given a current state, and are usually represented in five-tuples in this format:

$$(q, s, q', d, s')$$

1. q is the current state.
2. s is the symbol being read at the current position of the head.
3. q' is the next state the machine should transition into after the current state q and current symbol s are both matching this tuple.
4. d is the direction that the head will move after completing the first four steps (either left, right or none/stop).
5. s' is the symbol that overwrites s, as the current symbol being read at the current position of the head.

Each transition is a small set of instructions that, when paired with the entire set of transition tuples, provides the ability to define and understand the functionality of a Turing machine given an input, and its current state.

3.3 Formal Mathematical Definition of Turing Machine

Matching the order of our example above for the components of our Turing machine, we can define these components more formally by a seven-tuple, as:

$$M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$$

1. Q is a finite set of states the machine can be in.
2. Γ is the set of finite symbols as the tape alphabet, including the blank symbol b.
3. b is the blank symbol that appears on all blank cells of the tape, and appears on every cell that isn't inputted.
4. Σ is the input alphabet, which is the same as Γ with the blank cell symbol b excluded.
5. δ is the transition function that maps a state and tape symbol to a new state, with a symbol to write, and direction to move tape head.
6. q_0 is the initial state that the machine starts in when it begins to process the input, which is an element in Q.
7. F is the set of final or accepting states, also a subset of Q.

By defining a Turing machine this way, we create a framework to analyze and work with the operations (and limitations) that the machine, and therefore computational systems in general, which gives us consistency within theoretical computation.

Using States and Transitions to Create Our Palindrome-Checking Machine Specifications

Now that we understand the formal notation of a Turing machine, let's list our components for our particular machine as we listed above.

1. **Set of all states**

   $Q = \{q_0, q_{a1}, q_{a2}, q_{b1}, q_{b2}, q_{c1}, q_{c2}, \ldots, q_1, q_2, \ldots\}$

   Contains a state for every symbol on the input tape, as well as q_0–q_2.

2. **Tape alphabet**

   $\Gamma = \{a, b, c, \ldots, y, z, B\}$

   Lowercase alphabet, and the B blank symbol.

3. **Blank symbol definition**

   B

4. **Input alphabet**

   $\Sigma = \{a, b, c, \ldots, y, z\}$

5. **Set of transition functions (δ)**

$(q_0, !, B, R, q_{!1})$	From leftmost cell (start of tape) recognize input alphabet letter ! and mark with a B (blank)
$(q_{!1}, Y, Y, R, q_{!1})$	Move right on all input alphabet letters Y until head reaches B (the end of the entire input of tape)
$(q_{!1}, B, B, L, q_{!2})$	Move back once to the left
$(q_{!2}, !, B, L, q_1)$	Check if this character (the end character) matches the initially marked character and mark B if so
(q_1, Y, Y, L, q_1)	Move all the way back to the left (back to start)
(q_1, B, B, R, q_0)	Move to the right and transition back to q_0 to start the process again on next input char
(q_0, B, B, N, q_2)	If symbol being read is blank all symbols have been checked, word is palindromic, go to accept state
$(q_{!2}, B, B, R, q_0)$	If in state $q_{!2}$ and reading a B, change to q_0 to prepare for potential transition to accepting state

6. **Initial state**

   q_0

7. **Final (or accepting) state**

   $F = \{q_2\}$

   State which indicates input is palindromic; if machine halts in any other state, input not palindromic.

Although we could explicitly define the final states as accept and reject, indicating if the input is palindromic or not, it's also common to just define the state that confirms true in this context, which q_2 will represent. If our machine stops processing input and halts on any state that isn't q_2, the machine has determined that the input was not palindromic. Now that we've defined the machine's functionality, let's use a tape diagram to demonstrate our machine's functionality and its steps of operation.

3.4 Tape and Tape Head Diagrams

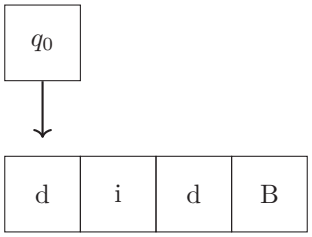
We can use diagrams to represent every step of a Turing machine's operations, from its initial tape (its input) to its final (or accepting) state, where the machine no longer has any other transition functions that match the current state and symbol being read. This is when the Turing machine halts, completing its computation.

Tape Diagrams

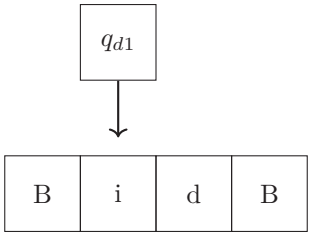
Drawing a tape diagram represents each cell on the infinitely long tape, inserting blank cells represented by 'B"s wherever there is no input. Here, an integer (that we'll call n) is represented in unary representation, which is simply using n many '1"s.

Demonstrating Our New Machine's Functionality with Tape Diagrams

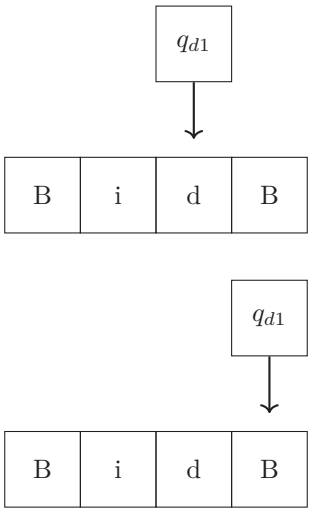
In our example, we're going to get our new machine to process the palindromic input of 'did'. Although we could have as many (or as few) blank cells as we like, we'll use four total tape cells. The transition tuples for this Turing Machine are listed above, so we're going to follow our machine's operation and the state of the tape at each step of its computation. Given the input and the set of transitions, the Turing Machine would act as follows:



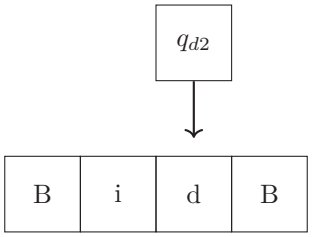
Here, the machine starts with the input in an initial tape, where the initial state q_0 starts at the leftmost cell. The transition $(q_0, !, B, R, q_{!1})$ applies here, as the machine is currently at state q_0 and reading a given letter of the input alphabet ! (d in this case). The machine's state will change to q_{d1} and write a B on the current cell, indicating it's marking this as the letter we're checking:



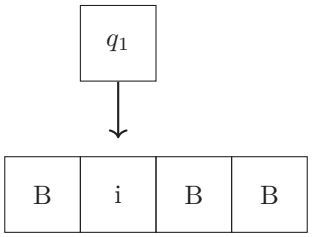
The transition $(q_{!1}, Y, Y, R, q_{!1})$ (where $q_{!1}$ is actually q_{d1}) applies for the next two steps, where Y is any letter (including the letter the machine is checking for) in the input alphabet. When reading any letter in the input alphabet, the machine is continuing right and remaining in state q_{d1}:



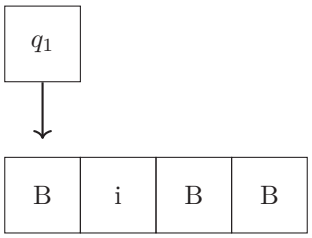
The transition $(q_{!1}, B, B, L, q_{!2})$ now applies as the machine is reading a B symbol. The head moves back to the left one cell and transitions to state q_{d2}:



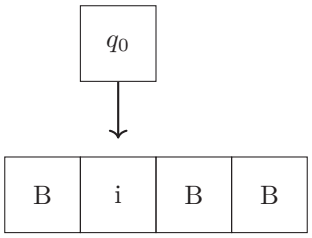
Since the symbol being read is equal to ! in our first transition that was applied (the character matches the first character that we marked with a B), the transition $(q_{!2}, X, B, L, q_1)$ applies. The machine overwrites the symbol with a B, moves to the left, and changes to state q_1. At this point, if the end symbol didn't match the starting symbol, the machine would halt, and as the state it would halt in wouldn't be q_2, the machine has determined the word is not a palindrome:



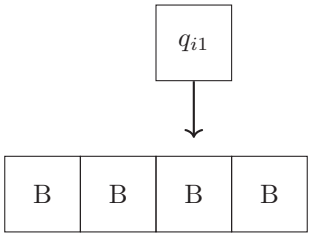
Next, the (q_1, Y, Y, L, q_1) applies, where the tape head continues to move left, provided that the machine is reading any letter within the input alphabet, all the way back to the symbol the machine just checked for and replaced with a B:



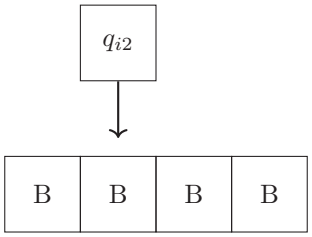
Now that our previous transition no longer applies and the machine instead reads a B, the transition (q_1, B, B, R, q_0) applies, where the tape head moves to the right while keeping the symbol as a B, and the machine transitions back to state q_0:



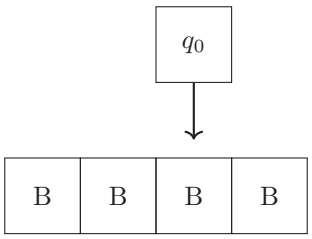
From this point, we start our cycle again with the first transition of $(q_0, X, B, R, q_{!1})$ applying (where q_{i1} is now q_{i1} in this case), where the machine is checking for an i, represented as ! in our transition tuple, at the end of the remaining input string that hasn't been replaced with a B. The machine replaces the symbol with a B and moves to the right, transitioning to state q_{i1}:



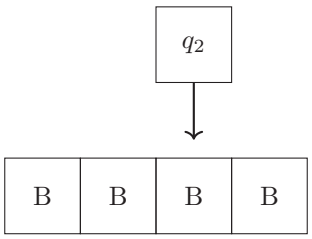
For any input, this process would continue indefinitely until either the symbol being checked for didn't exist at the end of the remaining input and the machine halts, or until the tape arrived at its current state in our example, where $(q!1, B, B, L, q!2)$ applies.



We've now reached our two final transitions where our machine will halt in its accepting state of q_4, indicating it has determined the input to be palindromic. The transition $(q!2, B, B, R, q_0)$ applies, changing to state q_0 and moving to the right.



Our final transition of (q_0, B, B, N, q_2) applies, where the symbol being read is still a B. The tape head doesn't move (N representing none for not moving the tape head), and the machine transitions into its final (accepting) state of q_2.



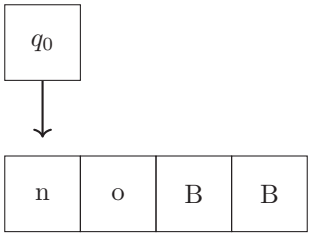
This concludes our machine's computation, and since it has halted at q_2, it has determined the input to be palindromic. This was determined by replacing the first and last letters of the input string with B, provided they matched, then starting again from the leftmost cell that wasn't a B and repeating the process.

If at any point the end symbol didn't match the starting symbol the machine was checking for, the machine would've halted, and seeing as it wouldn't have halted in q_2, the machine had determined that the input was not palindromic.

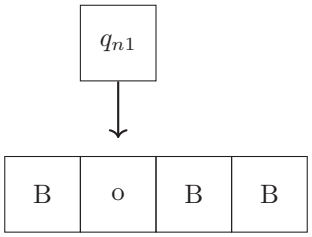
This example has shown the power of using the different components of a Turing machine to achieve the functionality that we want and, using just these required components, can create any possible program or computation that is possible for a human to perform given enough time and memory.

Processing a Non-Palindrome

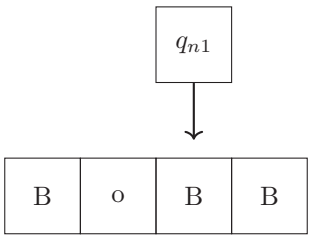
What happens if we give our machine an input that isn't a palindrome?



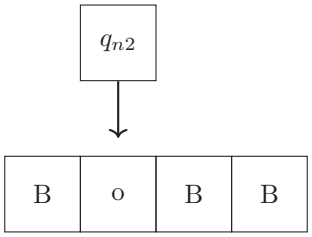
Here, the machine starts with the input in an initial tape, where the initial state q_0 starts at the leftmost cell. The transition $(q_0, !, B, R, q_{!1})$ applies here, as the machine is currently at state q_0 and reading a given letter of the input alphabet ! (n in this case). The machine's state will change to q_{n1} and write a B on the current cell, indicating it's marking this as the letter we're checking:



The transition $(q_{!1}, Y, Y, R, q_{!1})$ (where $q_{!1}$ is actually q_{n1}) applies for the next step, where Y is any letter (including the letter the machine is checking for) in the input alphabet. When reading any letter in the input alphabet, the machine is continuing right and remaining in state q_{n1}:



The transition $(q_{!1}, B, B, L, q_{!2})$ now applies as the machine is reading a B symbol. The head moves back to the left one cell and transitions to state q_{n2}:



Instead of the transition $(q_{!2}, !, B, L, q_1)$ applying (where $q_{!2}$ is actually q_{n2} in this case), our machine does not have any transition tuples that apply and halts. As it didn't halt in the accepting state of q_2, the machine decides the input isn't palindromic.

The Output of Our Machine

As we've seen, our palindrome-recognizing machine will successfully recognize a palindrome if it halts in the final/accepting state of q_2. If our machine doesn't halt in the accepting state and instead halts in a different state (which will always be q_1), the machine determined that the input wasn't palindromic. Our machine, therefore, will always halt for every input string.

The way that we should look at the behavior of our Turing Machine is to say that it verifies the proposition that "this input is palindromic," and the output of the machine is a Boolean value of true or false. The proposition (and the machine's output) is true if and only if the machine halts while in state q_2. If the machine halts in a rejecting state, the proposition (and output of the machine) is false.

The machine has decided on the proposition's truth, hence the English version of the *Entscheidungsproblem*; the 'decision problem'.

As we've seen above, our specific machine will always halt regardless of its input. But what happens if, given a certain input (such as a paradox), a machine never halts, and therefore never determines an output?

3.5 The Halting Problem

Now that we understand the Turing machine, its components, and its behaviour, we can dive deeper into the Halting Problem.

The Entscheidungsproblem and Invention of the Turing Machine

In the early 20th century, David Hilbert, along with other mathematicians, created the concept of a problem that would end up being foundational in theoretical computer science. It was noted as an important unsolved problem in the realm of mathematics. Hilbert's problem was solved if one could determine whether there is a definitive method (also known as an algorithm) that when applied to any mathematical statement, would definitively determine whether the statement is true or false. This became known as the *Entscheidungsproblem*, German for 'Decision Problem' [5]. The proposition of this problem led Alan Turing to invent the Turing machine, a theoretical computational model and abstract device to simulate the logic of a computer algorithm. In doing this, Turing had created a way that future computer scientists understand the limits of what mechanical computation can achieve and what it can't.

When Alan Turing introduced his Turing machines in 1936 through his paper, *"On Computable Numbers, with an Application to the Entscheidungsproblem"*, he stated that while his abstract computational model could compute any algorithm, it fell short of solving Hilbert's 'Decision Problem'. Turing introduced the Turing machine concept to explore the limits of what a machine

could compute, and his ongoing work with this model revealed that certain problems are impossible to solve, and therefore demonstrating that the *Entscheidungsproblem* is in fact unsolvable. The most famous example of such 'undecidable' problems is the Halting Problem, which further demonstrates the impossibility of resolving the *Entscheidungsproblem*.

When a Turing Machine Halts

As we've seen, if a Turing machine halts it means that the machine has completed its computation. As we saw in our example of determining palindromes, our machine would halt once it detected there wasn't a match between the first and last letters in that step, or once all letters had been checked and the word was determined to be palindromic. Depending on the state the machine halted in determines what result the machine reached. When halting, the machine was in the accepting state of q_2 if the word was palindromic. If it wasn't in q_2 and halted, the word wasn't palindromic. In short, the Turing machine stopped because it reached a final state that was defined in the set of states, and within one or more of the transition functions. But what happens when a Turing machine never encounters its final state?

When a Turing Machine Runs Forever

What happens when we give a Turing machine a problem that is unsolvable or "undecidable", such as a paradox? What if we give a machine that can understand language the input "This statement is false" and asked it to determine if that statement was true or false? The machine would run indefinitely, as an answer was impossible to determine. The machine will continue to assess the statement being true, causing the statement to be false, and vice versa, completing this cycle indefinitely. If a Turing machine can't determine an answer due to an indeterminable input, the machine will run forever.

In another example, let's imagine a machine that can either run forever or halt, given its input. The objective of this machine is to find a specific symbol, 1, on a tape that contains a sequence of 0s and 1s. The machine starts at the beginning of the tape and moves to the right, examining each symbol:

- If it finds a 1, this means the machine has successfully met its objective. It then transitions to a halt state, stopping its operation because it achieved what it was set to do.
- If the machine reaches the end of the tape without finding a 1, it enters a different state where it has no further instructions to halt. It continues moving right indefinitely or stays in place, effectively running forever because it never met its objective to find a 1.

This illustrates how a Turing machine can have two very different behaviors based on its input and the rules (transitions) it follows.

Some Turing machines are designed to halt when they successfully complete a task, like finding a particular symbol on a tape as in this example. Others may not find what they are designed to look for (given their designed transition functions) and enter a state where they continue to run indefinitely. Running forever (in this example) means the machine never reaches a conclusion or a halting point because the conditions for stopping are not met as the appropriate transition function to do so wasn't provided.

Behavior and Input

As we'll see, Turing machines take an input (its initial tape) and a set of transitions before processing the input through the set of transitions. When the processing is complete, and there are no more possible transitions that can be used given the tape's current state (or if the machine has been specifically instructed to halt within a given transition for the current tape) the Turing

machine will halt, indicating that it has completed its computation, and the tape state at this point is the result of the computation.

Relation to the Entscheidungsproblem

The Halting Problem is a version of the *Entscheidungsproblem* that asks if there is a Turing machine that could conceivably be constructed that can determine if a different Turing machine will halt (conclude its computation) or if that machine will continue to run infinitely and never arrive at an answer. In other words, the Halting Problem asks: "Is there a computer A that can always determine if another computer B will arrive at a conclusion or continue to run infinitely given any problem?"

It turns out that the answer is no. This famous problem was solved using knowledge of propositions to determine whether the proposition "there exists a Turing machine that can always determine if another Turing machine will eventually halt or run infinitely" is true or false. Turing proved that constructing a Turing machine with this ability is impossible, and that the Halting Problem is "undecidable" by using a proof by contradiction, where he proved that there is a problem a Turing machine can never solve.

Proof of the Halting Problem using Proof by Contradiction

Diagonalization Proof and Cantor

Proving the fact that the Halting Problem can't be decided (and that it's unsolvable) uses a form of logical reasoning known as *diagonalization*. A famous example of the use of diagonalization was used by Cantor, a mathematician that showed there are more real numbers than whole numbers [6]. Cantor posed trying to list all real numbers between 1 and 2. Then noted that by changing the first decimal of the first number, the second decimal of the second number and so on, we can create a new real number that wasn't in our original list. As this process can be repeated indefinitely, Cantor proved there are more real numbers than whole numbers.

Turing's Diagonalization Proof by Contradiction

Applied to the Halting Problem, let's assume that there is a Turing machine (a program) that can decide if a given problem will halt (and reach the end of its computation) and call this machine H.

We can create another Turing machine (we'll call D) that looks at the result of H, and outputs the opposite of what H determined. If H determines a particular Turing machine will halt, D will output that the Turing machine will never halt, and vice versa.

Let's look at an example; Seeing as D's programming says to output the opposite of H's determination, D will then be programmed to not halt and instead runs forever, making H incorrect, and vice versa.

This proof of contradiction demonstrates a problem that a Turing machine (H) can never solve, therefore proving that the Halting Problem can't be determined by any Turing machine.

As we saw earlier, Cantor showed there are more real numbers than whole numbers, by imagining listing all real numbers and then creating a new number by changing one digit in each number of the list. This new number couldn't be found anywhere on his list, so there is proof that there are more real numbers than whole numbers.

Turing's proof uses a similar idea. He imagines a list where each Turing machine (as an input to H) has a corresponding decision from H about whether it will halt or not. Turing then designs another machine D. This new machine's behavior is specifically designed to contradict H's predictions, where if H says a machine will halt, D will do the opposite. This "new entry" in the

list (that is, the behavior of the new machine D) is something H can never predict correctly, just like how Cantor's new real number wasn't on his list.

This process shows that, just like Cantor's list couldn't cover all the real numbers, no Turing machine (like H) can correctly decide every possible case of the Halting Problem. This creates a fundamental contradiction, proving that the Halting Problem can't be solved by any Turing machine.

Mathematical Proof of the Halting Problem

Let us demonstrate this proof mathematically. To demonstrate this proof by contradiction mathematically (using programs in place of Turing machines), we need to prove that there exists a problem that no Turing Machine can solve. Let us assume that H exists, where H is a program that can always decide whether any program P with a given input i halts or continues to run forever.

Defining H and P

The function of H can be represented as:

$$H(P, i)$$

which returns `true` if P halts based on input i, or `false` if not.

Defining D

We can then construct a new program D that instead takes the program P as input. We can then set D's result (halting or running infinitely) to the opposite of H's output.

Running $H(P, i)$ (using H to determine whether P halts or runs infinitely):

- If H determines P will halt (given input of i), D should do the opposite of H's determination and run infinitely.
- If H determines P will run forever (given input of i), D should do the opposite of H's determination and halt.

Formal Argument for Proof by Contradiction

To show a formal proof for the Halting Problem, let us define both machines H and D and demonstrate the contradiction.

H Definition

We define the Turing machine H as:

$$H = (Q_H, \Gamma_H, b, \Sigma_H, \delta_H, q_{0H}, F_H)$$

where:

- Q_H is the set of all states, including states like q_{halt} and q_{run}.
- Γ_H is the tape alphabet.
- b is the blank symbol.

- Σ_H is the input alphabet.
- δ_H is the transition function.
- q_{0H} is the initial state.
- F_H is the set of accepting states, specifically $F_H = \{q_{\text{halt}}\}$.

Machine H operates as follows:

- H takes another Turing machine P and its input i, and simulates P on i.
- If H decides that P halts on input i, it halts in state q_{halt}.
- If H decides that P runs forever on input i, it halts in state q_{run}.

In other words, H will accept if it determines that P halts on i, and reject if it determines that P does not halt.

D Definition

We construct a new Turing machine D defined as:

$$D = (Q_D, \Gamma_D, b, \Sigma_D, \delta_D, q_{0D}, F_D)$$

where:

- $Q_D = Q_H$ (the set of states of D is the same as H).
- $\Gamma_D = \Gamma_H$ (the tape alphabet is the same).
- b is the blank symbol.
- $\Sigma_D = \Sigma_H$ (the input alphabet is the same).
- δ_D is the transition function of D, modified from δ_H.
- $q_{0D} = q_{0H}$ (the initial state is the same).
- $F_D = \{q_{\text{run}}\}$ (the accepting state for D is q_{run}).

The transition function δ_D is defined as:

$$\delta_D = \delta_H \cup \left\{ (q_{\text{halt}}, a, q_{\text{halt}}, a, N) \;\middle|\; a \in \Gamma_D \right\}$$

Machine D is identical to H, with two distinct changes:

- It runs forever when reaching the state q_{halt}.
- It only halts on the state q_{run}, based on the Turing Machine it takes as input.

In the context of our proof, we can input H into D. The behavior of D depends on H's decision:

1. Where H decides its input will halt, D instead uses an additional transition:

$$\delta(q_{\text{halt}}, Y, Y, N, q_{\text{halt}})$$

   This causes D to run forever, never halting on q_{halt}.

2. Where H decides its input will never halt, D will instead accept the state q_{run}, and D will halt.

In other words, D accepts the state that H does not accept, and D enters into an infinite loop (running forever) on what H does accept.

Theorem 3.1: The Halting Problem is Undecidable

There is no Turing machine that can solve the Halting Problem for all possible inputs.

Proof 3.1: Proof of Theorem 3.1

Assume, for contradiction, that there exists a Turing machine H that can decide whether any Turing machine P halts on input i.
Construct a new Turing machine D that takes P as input and behaves as follows:

1. Run H on input (P, P).
2. If H decides that P halts on input P, then D enters an infinite loop.
3. If H decides that P does not halt on input P, then D halts.

Now consider running D on input D:

1. If H decides that D halts on input D, then by the definition of D, it will loop infinitely, which is a contradiction.
2. If H decides that D does not halt on input D, then D halts, which is also a contradiction.

This contradiction implies that our assumption that H exists must be false. Therefore, no Turing machine can solve the Halting Problem for all possible inputs.

Contradiction Explanation

Our contradiction arises when we ask H to determine if D will halt, given D (itself) as input. By running $H(D, D)$, we arrive at two cases:

1. In Case 1, the fact that H ends in state q_{halt} implies that its input (D) halts, implying that its own input (D) runs forever. Instead, D ran forever instead of halting, contradicting the fact that H halted in state q_{halt}, deciding D would halt. We can observe that H made an incorrect decision on the state of its input.
2. In Case 2, the fact that H ends in state q_{run} implies that its input (D) runs forever, which implies that its own input (D) halts. Instead, D halted instead of running forever, contradicting the fact that H halted in state q_{run}, deciding D would run forever. We can also observe that H made an incorrect decision on the state of its input.

The proposition presented is that a machine (H) exists that can solve the Halting Problem for all possible inputs to the machine. We have specified an input that is impossible for H to solve.

By demonstrating a possible input that cannot be solved by H, we have proven by contradiction that a machine that can solve the Halting Problem for all inputs cannot exist, and therefore proving the Halting Problem is undecidable.

Contradiction Cases Summary

Case 1

- H determines that D will halt, and H halts in state q_{halt}.
- D sees that H has determined it will halt, and uses its programming to run forever.
- This is a contradiction. H halted on q_{halt} and decided that D would halt, but instead D ran forever.
- H's decision was wrong.

Case 2

- H determines that D will run forever and halts on q_{run}.
- D has the accepting state of q_{run} and halts.
- This is another contradiction. H halted in q_{run} and decided D would run forever. D saw this and halted.
- H's decision was wrong.

The proposition presented is that a machine (H) exists that can solve the Halting Problem for all possible inputs to the machine. We have specified an input that is impossible for H to solve.

By demonstrating an input that H cannot solve, we have proven through contradiction that no machine can solve the Halting Problem for all possible inputs. Therefore, this proves Halting Problem is undecidable.

3.6 Turing Completeness

If a system, whether it's a programming language, an advanced video game, a program, a machine (or anything else), can replicate all the characteristics and abilities of a Turing machine, that system is known as being *Turing complete*.

What is Turing Completeness?

For a system to be considered Turing complete, it needs to be able to simulate a Turing machine in its entirety, meaning it needs to be able to solve any program a Turing machine can solve given enough time and memory. The system must:

1. Be able to manipulate data by reading, writing, and changing data, given an arbitrary amount of memory.
2. Be able to control flow through branching statements, making decisions based on logical conditions (like if-else statements).
3. Be able to repeat or recurse operations.

The Power of the Turing Machine

Although it may not immediately be apparent, a Turing machine is capable of successfully running any algorithm. Therefore, if something is to be determined as Turing complete, it also needs to be able to implement any algorithm too.

Proving Turing Completeness

To prove that a system is Turing complete, we can demonstrate that the system can simulate another system that has already been proven to be Turing complete. If we can show that our system can simulate a Turing machine and its capabilities, our system will be determined to be Turing complete. We can prove this through:

1. **Explicit Simulation**: Simulating a Turing machine to demonstrate our system can replicate every characteristic of a Turing machine by simulating a Turing machine's tape, states, and set of transition functions.
2. **Encoding**: Demonstrating our system can encode and interpret any algorithm that a Turing machine could execute. As an example, Python can be written to be able to accept the description of any Turing machine (a tape, states, and set of transition functions) as an input and execute those in the same way a Turing machine would.
3. **Reduction**: Showing that our system is complete because it can simulate something else that is known to be Turing complete. As an example, JavaScript can simulate lambda calculus, which is already known to be Turing complete, making JavaScript Turing complete too.

Examples of Turing Complete Real-World Systems

As an example of proof by explicit simulation (even though it wasn't by design by the game's programmers), the video game *Minecraft* has been determined to be Turing complete. In the game, players can build a wide variety of structures using a building block system. This system also contains an emulation of wiring and circuitry with many varying components. With the inclusion of players being able to create circuits using in-game functionality, an in-game simulation of all the capabilities of a Turing machine can be replicated, determining *Minecraft* to be Turing complete.

Some other surprising non-obvious examples of Turing complete systems are Microsoft PowerPoint (as the slides can represent states and transitions), the CSS/HTML web languages (through interactive elements), and Microsoft Excel (by using its inbuilt formulas with cells and recursive abilities).

3.7 Church Lambda Calculus and the Church-Turing Thesis

As we've seen, there were other computational models proposed other than the Turing Machine to be able to define computational functionality. The most interesting of these is arguably 'Church Lambda Calculus' [1], and exploring it in some details gives insight into why the Turing Machine model came to be more widely known and accepted as an ideal model for what is possible in computation.

A Different Approach to a Model

Both Alan Turing and Alonzo Church devised what is now known as the *Church-Turing thesis* [2], which states that any function that can be computed by hand can be computed by a Turing

machine, underlying our modern understanding of what the definition of a system is to be able to perform general-purpose computation.

In the 1930s, Church developed a formal mathematical system for demonstrating computations using functions systematically, including how to create and apply these functions, and how to incorporate variables for use within those functions, known as *lambda calculus*. With Alan Turing also creating the concept of the Turing machine as a different way of describing computation.

Church formalized effective computation through defining functions and transformations instead of using the tape symbols and states found in Turing machines. Interestingly, both lambda calculus and Turing machines were proved to be equivalent in expressive power (by Turing himself in 1937) even though they were developed independently.

Components of Lambda Calculus

Lambda calculus is (just like a Turing Machine) a mathematical concept to demonstrate how computation can be performed. Seeing as lambda calculus can be used to express and implement any algorithm or computation that a Turing machine can, lambda calculus is Turing complete. For a demonstration of how lambda calculus achieves this, we need to include:

1. **Functions**: The primary objects used in lambda calculus, written as expressions that can be evaluated when being applied to arguments.
2. **Abstractions and Applications**: Lambda calculus utilizes two main operations; abstractions (defining functions) and applications (applying those functions to arguments). As an example, an abstraction can be expressed as $\lambda x.M$, where λ is denoting function abstraction (basically defining a function before the contents of the function), M is an expression, and x is a variable, where this represents a function that computes M given x.
3. **Variables and Substitutions**: Variables represent inputs to functions, and the substitution of those variables (replacing a variable within an expression with a different expression), enabling the operational mechanism lambda calculus provides.

Lambda Calculus for Arithmetic Example

In lambda calculus, natural numbers are usually represented as Church Numerals, where each numeral is defined as a function that applies another function multiple times, corresponding to the numeral's value. In order to be able to perform a simple arithmetic calculation (such as adding one to one), we first need to define the zero function, the successor function, the addition function, and the function that represents one. In summary, these are written as:

- Zero Function: $\lambda f.\lambda x.x$ — Accepts function and value as an input, and returns the value without changing it.
- Successor Function: $\lambda n.\lambda f.\lambda x.f(nfx)$ — Takes a number n and creates a function that applies that function one more time than n, effectively adding one to the input n.
- One: $(\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.x)$ — Combining zero and the successor functions to yield one, simplified to $\lambda f.\lambda x.fx$.
- Addition: $\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$ — Applies successor function to input m, as many times as second input n, essentially adding n amount of ones to m, adding m to n.

Therefore, if we wanted to add one and one, we can use the addition function with both arguments being the above representation of one:

$$(\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))(\lambda f.\lambda x.fx)(\lambda f.\lambda x.fx)$$

This simplifies to $\lambda f.\lambda x.f(fx)$, which is also the representation of two in Church Numerals, meaning that the function is applied twice to x.

Lambda Calculus and Turing Machines

As we can see, lambda calculus is a system that is Turing complete, as it can perform any operation that a Turing machine can perform, and therefore any function that can be computed by a Turing machine can also be expressed in lambda calculus.

Limits of Computation Realization

Church and Turing both highlighted limits of computation as a result of this work and proved that there is no computational procedure (like a Turing machine) that can solve every possible mathematical question, meaning that some mathematical statements can't be proved or disproved by any algorithm. This demonstrates the limits of what a computer can do, providing a boundary for computer scientists to understand and work within.

3.8 Practical Uses for Turing Machines in the Real World

Above all else, our ability to understand the concept of Turing machines underpins the basis of our knowledge about computation by specifically defining what a system capable of computation can and cannot do, even if it is given infinite time and infinite memory. By understanding Turing machines, we are able to fully understand the fundamentals of algorithms (transition functions) and computation theory.

A Deep Understanding

Having a deep understanding of Turing machines also assists software developers in understanding the efficiency and feasibility of algorithms within programs so that software development can be optimized. This also leads to understanding the classifications of computational complexity, such as Big O notation, where the concept of Turing machines guides the development of more effective algorithms and computational models. To make sure devices and the internet are more secure, cryptography is also underpinned by the principles of Turing machines.

Turing Machines in the Future

As the world moves closer to the advent of Artificial Intelligence (AI), Turing machines provide the theoretical framework for understanding the abilities and limits of AI and its algorithms, including its ability to make decisions and the processes that allow it to. Quantum computing is also an emerging technology, and the concept of the Turing machine is being utilized through theoretical models of a Quantum Computer (known as a *Quantum Turing Machine* or QTM) as an extension of the classical Turing machine. The key difference between a regular Turing machine and its quantum counterpart is that the symbols that are written on the tape, the state of the machine, and the movements of the tape head on a QTM can exist in a superposition of states, following quantum mechanics. This ability to define what this machine can and cannot do is still assisting computer scientists to be able to create more powerful machines, using the original concepts formulated by Alan Turing.

3.9 Conclusions

When Alan Turing theorized the mathematical model and concept of a Turing machine in 1936, he thrust humanity into the age of computing. Although Alonzo Church and others were developing similar concepts and methods to solve the *Entscheidungsproblem*, Turing's model proved to be more intuitive and practical, establishing the foundation for the modern design of computers and our understanding of what it can and cannot do. Unlike Church's lambda calculus, Turing's model visualized a simpler but equally powerful model for defining the ability to perform general-purpose computation, which led to future computer scientists developing physical machines and computers, directly basing their work on the knowledge that Alan Turing conceptualized.

Context

This article was adapted from a report submitted for the SIT192 unit.

About the Author

Ari Robin is a Computer Science student at Deakin University in Melbourne, Australia, with interest in the development and applications of Quantum Computing.

Acknowledgements

I would like to thank Dr. Julien Ugon for assisting me in refining and inspiring the content of this article, and his relentless efforts in helping me pursue my interests in mathematics.

References

All diagrams were created using Tikz [10].

[1] Jesse Alama and Johannes Korbmacher. Lambda calculus. *The Stanford Encyclopedia of Philosophy*, E. N. Zalta (ed.), 2023.

[2] B. Jack Copeland. The Church–Turing Thesis. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, 2023.

[3] B. Jack Copeland. The rise and fall of the entscheidungsproblem. The Stanford Encyclopedia of Philosophy, edited by Edward N. Zalta, Metaphysics Research Lab, Stanford University, 2023.

[4] C. Critchlow and D. J. Eck. Turing machines. Hobart and William Smith Colleges, 2023.

[5] Editors of Encyclopaedia Britannica. Turing machine, computing device. Encyclopædia Britannica, 2024.

[6] Alexander Kharazishvili. Cantor's diagonalization method. *Inference: International Review of Science*, 2(3), 2016.

[7] Liesbeth De Mol. Turing machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2021 edition.

[8] K. Moore, I. Koswara, and C. Williams. Turing machines. Brilliant.org, 2024.

[9] Kenneth H. Rosen. Turing machines. In *Discrete Mathematics and its Applications (8th Edition)*, pages 927–936. McGraw-Hill Education, New York, 2019.

[10] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[11] S. Tuteja. Turing machine in TOC. Geeks for Geeks, 2023.

4

Equivalence relations and partial orders

Kate Suraev

Abstract

We encounter relationships and hierarchies daily, from social networks and food classifications to academic prerequisites and task scheduling. Equivalence relations and partial orders are fundamental concepts in discrete mathematics that provide a formal framework for understanding and modelling relationships and hierarchies between elements in a set. This article explores the properties of equivalence relations, equivalence classes, the relationship between equivalence relations and partitions, and the properties of partial orders, Hasse diagrams, and topological sorting. This article will also discuss the applications of equivalence relations and partial orders in real-world contexts, demonstrating their significance in structuring and organising data and solving complex problems.

4.1 Introduction

This article will initially focus on **equivalence relations**, particularly their properties, equivalence classes, and the relationship between equivalence relations and partitions.

We aim to gain a thorough understanding of equivalence relations and achieve a comprehensive proof of the Fundamental Theorem of Equivalence Relations.

The Fundamental Theorem of Equivalence Relations will provide insight into the one-to-one correspondence between equivalence relations and partitions. We will examine how these terms are often used interchangeably to describe dividing a set into non-overlapping subsets of related elements. We will also explore how this concept finds applications in various fields, including automata theory, software testing, data analysis, and machine learning.

Secondly, this article will focus on **partial orders**, particularly their properties, Hasse diagrams and topological sorting. We aim to understand some key concepts of partial orders, such as maximal and minimal elements, greatest and least elements and the restriction of partial orders to subsets of a poset. We will also examine an example of a topological sorting algorithm, using a series of Lemmas to prove the correctness of the algorithm and the existence of a topological sort for every finite poset. These studies will provide insight into the vast nature of partial orders and their applications and significance in various domains, including depth-first search algorithms, scheduling tasks, and dependency management.

4.2 Equivalence Relation Properties

For a relation R to be an equivalence relation, it must satisfy the properties of **reflexivity**, **symmetry**, and **transitivity**. We will define reflexive, symmetric, and transitive properties,

provide the methods to prove each property and offer examples to prove and disprove each. These properties are essential to prove that a relation is an equivalence relation and will be applied in the following sections.

Reflexive Relation

Definition 4.1: Reflexive Relation

A relation R on a set S is reflexive if:

$$\forall a \in S,\ (a, a) \in R \tag{4.1}$$

Figure 4.1: Directed Graph Representation of a Reflexive Relation.

Proving and Disproving Reflexive Relations

Taking note of the universal quantifier in Eqn. (4.1), we can use a direct proof to show that R is reflexive or provide a counterexample to show that R is not reflexive [14].

To prove that R is reflexive, we must show that for all elements $a \in S$, the ordered pair $(a, a) \in R$ [16]. To do this, we pick any arbitrary element $a \in S$ since the variable a is universally quantified, then we prove that $(a, a) \in R$. To disprove that R is reflexive, we need to show that there exists an element $a \in S$ such that $(a, a) \notin R$.

Example 4.1: Proving a Reflexive Relation

Consider the relation $R = \{(a, b) \mid a + b = even\}$ on the set of integers, $\mathbb{Z}$. To prove that R is reflexive, we need to show that for all elements $a \in \mathbb{Z}$, $(a, a) \in R$.

Using a direct proof, let a be an arbitrary element of $\mathbb{Z}$. Since $a + a = 2a$, and 2a is even, then $(a, a) \in R$ and thus R is reflexive.

Example 4.2: Disproving a Reflexive Relation

Consider the relation $R = \{(a, b) \mid a = m \times b\}$, where m is an integer, on the set of integers, $\mathbb{Z}$ [19]. To disprove that R is reflexive, we can provide a counterexample that shows there exists an element $a \in \mathbb{Z}$ such that $(a, a) \notin R$.

Let $a = 3$. For $(3, 3)$ to be in R, there must exist an integer m such that $3 = m \times 3$, which implies $m = 1$. However, if we consider $m \neq 1$, the equation $3 = m \times 3$ does not hold. For example, if $m = 2$, then $3 \neq 3 \times 3$.

Therefore, $(3, 3) \notin R$ when $m \neq 1$, and hence R is not reflexive.

Symmetric Relation

Definition 4.2: Symmetric Relation

A relation R on a set S is symmetric if:

$$\forall a, b \in S,\ (a, b) \in R \rightarrow (b, a) \in R \tag{4.2}$$

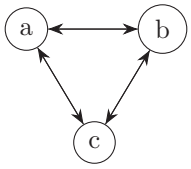


Figure 4.2: Directed Graph Representation of a Symmetric Relation.

Proving and Disproving Symmetric Relations

Taking note of the universal quantifier and implication in Eqn. (4.2), we can use a direct proof to show that R is symmetric or provide a counterexample to show that R is not symmetric [14].

To prove that R is symmetric, we need to show that for all elements $a, b \in S$, if $(a, b) \in R$, then $(b, a) \in R$. Using a direct proof, we assume that $(a, b) \in R$ for arbitrary elements $a, b \in S$ and show that $(b, a) \in R$.

To disprove that R is symmetric, we need to show that there exists a pair of elements $a, b \in S$ such that $(a, b) \in R$ but $(b, a) \notin R$.

Example 4.3: Proving a Symmetric Relation

Consider the relation $R = \{(a, b) | a + b = 0\}$ on the set of integers, $\mathbb{Z}$ [16]. To prove that R is symmetric, we need to show that for all elements $a, b \in \mathbb{Z}$, if $(a, b) \in R$, then $(b, a) \in R$.

Using a direct proof, let a, b be arbitrary elements of $\mathbb{Z}$ such that $(a, b) \in R$.

If $a + b = 0$, then $b + a = 0$ by the commutative property of addition[1].

Therefore, $(b, a) \in R$ and R is symmetric.

Example 4.4: Disproving a Symmetric Relation

Consider the relation $R = \{(a, b) | a \text{ is a multiple of } b\}$ on the set of integers, $\mathbb{Z}$ [16].

To disprove that R is symmetric, we can provide a counterexample that shows there exists a pair $(a, b) \in R$ such that $(b, a) \notin R$.

Consider the pair $(2, 1)$. Clearly, 2 is a multiple of 1, so $(2, 1) \in R$.

If R is symmetric, then $(1, 2) \in R$. This implies that 1 is a multiple of 2. However, 1 is not a multiple of 2 because there is no integer k such that $1 = 2k$.

Therefore, $(2, 1) \in R$ but $(1, 2) \notin R$, and consequently R is not symmetric.

[1] Commutative Property of Addition: For all real numbers x and y, $x + y = y + x$ [16].

Transitive Relation

Definition 4.3: Transitive Relation

A relation R on a set S is transitive if:

$$\forall a, b, c \in S,\ (a, b) \in R \land (b, c) \in R \rightarrow (a, c) \in R \tag{4.3}$$

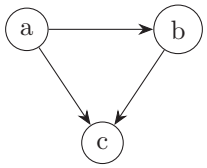


Figure 4.3: Directed Graph Representation of a Transitive Relation.

Proving and Disproving Transitive Relations

Again, taking note of the universal quantifier and implication in Eqn. (4.3), we can use a direct proof to show that R is transitive or a counterexample to show that R is not transitive [14].

To prove that R is transitive, we must show that $\forall a, b, c \in S$, if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$. Using a direct proof, we assume that $(a, b) \in R$ and $(b, c) \in R$ for arbitrary elements $a, b, c \in S$ and show that $(a, c) \in R$. To disprove that R is transitive, we must show that there exist elements $a, b, c \in S$ such that $(a, b) \in R$ and $(b, c) \in R$ but $(a, c) \notin R$.

Example 4.5: Proving a Transitive Relation

Consider the relation $R = \{(a, b) | a \text{ is a multiple of } b\}$ on the set of integers, $\mathbb{Z}$ [19]. To prove that R is transitive, we must show that $\forall a, b, c \in \mathbb{Z}$, if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$.

Let a, b, c be arbitrary elements of $\mathbb{Z}$. Assume that a is a multiple of b, and b is a multiple of c. This implies there are integers k and l such that $a = k \times b$ and $b = l \times c$.

Substituting the value of b into the equation $a = k \times b$, we get $a = k \times l \times c$. Since $k \times l$ is an integer[1], a is a multiple of c, and $(a, c) \in R$. Therefore, R is transitive.

Example 4.6: Disproving a Transitive Relation

Consider the relation $R = \{(a, b) | a - b \text{ is odd}\}$ on the set of integers, $\mathbb{Z}$. To disprove that R is transitive, we can provide a counterexample that shows there exist elements $a, b, c \in \mathbb{Z}$ such that $(a, b) \in R$ and $(b, c) \in R$ but $(a, c) \notin R$.

Consider the pair $(1, 2)$. Clearly, $1 - 2 = -1$ is odd, so $(1, 2) \in R$.
Consider the pair $(2, 3)$. Similarly, $2 - 3 = -1$ is odd, so $(2, 3) \in R$.

If R is transitive, then $(1, 3) \in R$. This implies that $1 - 3 = -2$ is odd. However, -2 is not odd as $-2 \mod 2 = 0$.

Therefore, $(1, 2) \in R$ and $(2, 3) \in R$ but $(1, 3) \notin R$, and thus R is not transitive.

[1] Product of Two Integers: If k and l are integers, then their product $k \times l$ is also an integer [3].

4.3 Equivalence Classes and Partitions

The previous section has provided us with the tools to prove that a relation is an equivalence relation. We can show that the relation is **reflexive**, **symmetric**, and **transitive** by combining the proofs of these properties.

Definition 4.4: Equivalence Relation

An equivalence relation is a binary relation that is reflexive, symmetric, and transitive.

Note: We denote an equivalence relation by R and write aRb to indicate that a is related to b by R.

Consider a set S and an equivalence relation R on S. The set S, under R, is subdivided into equivalence classes.

Equivalence Classes

Definition 4.5: Equivalence Class

Let R be an equivalence relation on a set S. The equivalence class of an element $a \in S$, denoted by $[a]_R$, is defined as:

$$[a]_R = \{x \in S \mid xRa\}$$

In other words, the equivalence class of a is the set of all elements in S that are related to a by R [8].

What do equivalence classes represent?

These equivalence classes have specific properties that allow us to partition the set S into non-overlapping subsets, where each subset contains elements related to each other by the equivalence relation R.

Properties of Equivalence Classes

Theorem 4.1: Properties of Equivalence Classes

Let R be an equivalence relation on a set S. Then the following properties hold for the equivalence classes of S under R [10]:

1. $\forall a \in S, a \in [a]$
2. $\forall a, b \in S, aRb \leftrightarrow [a] = [b]$
3. $\forall a, b \in S, [a] = [b] \vee [a] \cap [b] = \emptyset$

What do the properties of equivalence classes mean in Theorem 4.1?

1. Every element of S belongs to its own equivalence class; in other words, each element is related to itself by the equivalence relation.
2. Two elements of S are equivalent if and only if their equivalence classes are equal.
3. Any two equivalence classes are either equal or disjoint. [12]

Properties of Equivalence Classes Proof

We want to prove that if R is an equivalence relation on a nonempty set S, then we can partition S into equivalence classes that satisfy properties 1, 2, and 3 of Theorem 4.1 [12].

For the following proofs in this section:

Let S be a nonempty set and R be an equivalence relation on S.

Proof 4.1: Properties of Equivalence Classes: Property 1

Property 1 of Theorem 4.1 states that $\forall a \in S, a \in [a]$.

Let a be an arbitrary element of S. Since R is an equivalence relation, it has the property of reflexivity. Therefore, $\forall a \in S, aRa$, and so we can conclude that a is an element of its equivalence class $[a]$. Therefore, for every element $a \in S$, $a \in [a]$ and Property 1 holds for the equivalence classes of S under R.

Proof 4.2: Properties of Equivalence Classes: Property 2

Property 2 of Theorem 4.1 states that $\forall a, b \in S$, aRb if and only if $[a] = [b]$. This means that two elements of S are equivalent if and only if their equivalence classes are equal [10].

To prove this biconditional statement, we need to show that if aRb, then $[a] = [b]$ **and** if $[a] = [b]$, then aRb.

Part 1: If aRb then $[a] = [b]$.

Let a, b be arbitrary elements of S and assume aRb. To prove that $[a] = [b]$, we need to show that $[a] \subseteq [b]$, and $[b] \subseteq [a]$ [10].

To show that $[a] \subseteq [b]$:
Let x be an arbitrary element of $[a]$, $x \in [a]$. This implies that xRa, by Definition 4.5, of equivalence classes.
Since aRb and xRa, then xRb by the transitive property of equivalence relations.
Since xRb, $x \in [b]$. Therefore, if $x \in [a]$ and $x \in [b]$, then $[a] \subseteq [b]$ because x was an arbitrary element of $[a]$.

To show that $[b] \subseteq [a]$:
Let y be an arbitrary element of $[b]$, $y \in [b]$. This implies that yRb by Definition 4.5 of equivalence classes.
Since yRb, then bRy by the symmetric property of equivalence relations.
Since bRy and aRb, then aRy by the transitive property of equivalence relations.
Then, yRa by the symmetric property of equivalence relations.
This proves that y is an element of $[a]$, $y \in [a]$.
Therefore, if $y \in [b]$ and $y \in [a]$, then $[b] \subseteq [a]$.

Since we have shown that $[a] \subseteq [b]$ and $[b] \subseteq [a]$, we can conclude that $[a] = [b]$ by definition of subset[1] and set equality[2]. This completes the first of the proof: if aRb, then $[a] = [b]$.

[1] Definition of subset: A is a subset of B, denoted as $A \subseteq B$, if and only if $\forall x(x \in A \rightarrow x \in B)$. [16]
[2] Definition of set equality: A is equal to B, denoted as $A = B$, if and only if $\forall x(x \in A \leftrightarrow x \in B)$. [16]

Part 2: If $[a] = [b]$ then aRb.

Let $(a, b) \in S$. Assume that $[a] = [b]$ is true.

Using Part 1 of the proof and the assumption $[a] = [b]$, we know that $a \in [a]$ and since $[a] = [b]$, a is an element of $[b]$, $a \in [b]$. If $a \in [b]$, then aRb.

Therefore, we can conclude that if $[a] = [b]$, then aRb.

Proof 4.3: Properties of Equivalence Classes: Property 3

Property 3 states that for each $a, b \in A$, $[a] = [b]$ or $[a] \cap [b] = \emptyset$. This means that any two equivalence classes are equal or disjoint [10].

Let $a, b \in S$.

From Property 2, we know that if aRb, then $[a] = [b]$. Now we want to show that if $a \notin [b]$, then $[a] \cap [b] = \emptyset$, meaning no common element exists between the equivalence classes of a and b.

Let $a \notin [b]$. This implies that $(a, b) \notin R$.

Let $x \in [a]$ be an arbitrary element of S. Then, $(a, x) \in R$ by Definition 4.5 of an equivalence class.

We want to prove that $x \notin [b]$ using proof by contradiction.

Assume that $x \in [b]$. This implies that $(b, x) \in R$. By the definition of symmetry, if $(b, x) \in R$, then $(x, b) \in R$.

However, $(a, x) \in R$ and $(x, b) \in R$ implies that $(a, b) \in R$ by the transitive property of equivalence relations, which contradicts $(a, b) \notin R$.

Therefore, $x \notin [b]$ and $[a] \cap [b] = \emptyset$.

We have shown that if $a \notin [b]$ then $[a] \cap [b] = \emptyset$. Therefore, any two equivalence classes are either equal or disjoint.

By proving the properties of equivalence classes, we have shown that the equivalence classes form a partition of the set S under the equivalence relation R. However, our proof does not explicitly state this. This forms one half of the Fundamental Theorem of Equivalence Relations, and the focus of the next section will be to prove this.

Before we can show that the equivalence classes of a set S form a partition of S, we need to review the definition and properties of a partition.

4.4 Partitions

If we examine the definition and properties of a partition, we can observe similarities between the properties of equivalence classes and partitions. This can be used to show that the equivalence classes of a set under an equivalence relation satisfy the properties of a partition and, therefore, form a partition of the set.

Definition 4.6: Partition

A partition P of a set S is a subdivision of S into pairwise disjoint, nonempty sets [4]. P can be expressed as $P = \{S_i \mid i \in I\}$ where each S_i is a nonempty subset of S and the union of all S_i is equal to S [17].
Note: I is the index set of the partition $\{1, 2, 3, \ldots, n\}$.

Definition 4.7: Properties of a Partition

The collection of distinct subsets $\{S_1, S_2, \ldots, S_n\} \subseteq S$, where n is the number of subsets, forms a partition of S if the following three properties are satisfied [16]:

1. $S_i \neq \emptyset$ for all i
2. $S_i \cap S_j = \emptyset$ if $i \neq j$
3. $\bigcup_{i=1}^{n} S_i = S$

What do the properties of a partition mean in Definition 4.7?

1. Each subset of the partition is nonempty, meaning it contains at least one element.
2. The subsets are pairwise disjoint. This means that any two distinct subsets do not share any elements.
3. The union of the subsets is equal to S. [12]

We want to prove that the equivalence classes of a set S under an equivalence relation R form a partition of S. To do this, we need to show that the equivalence classes are nonempty, pairwise disjoint and that the union of the equivalence classes is equal to the starting set S.

Equivalence Classes Form a Partition

Theorem 4.2: Equivalence Classes Form a Partition

If R is an equivalence relation on a nonempty set S, then the distinct set of equivalence classes of S under R form a partition of S [10].

For the following proofs in this section:

Let R be an equivalence relation on a nonempty set S.

The collection of distinct equivalence classes of S under R are denoted as $\{S_1, S_2, ..., S_n\}$.

Proof 4.4: Equivalence Classes Form a Partition: Property 1

Property 1 of Partitions, Definition 4.7, states that each subset S_i is nonempty. From Proof 4.1, we have shown that each element of S belongs to its own equivalence class by the reflexivity property.

Specifically, for any $a \in S$, $a \in [a]$. This means that every equivalence class must contain at least one element and therefore, each subset S_i is nonempty.

Proof 4.5: Equivalence Classes Form a Partition: Property 2

Property 2 of Partitions, Definition 4.7, states that the subsets are pairwise disjoint. For any $i, j \in I$, if $i \neq j$, then $S_i \cap S_j = \emptyset$.

We will prove the contrapositive of this statement: for any $i, j \in I$, if $S_i \cap S_j \neq \emptyset$, then $i = j$.

Suppose $S_i \cap S_j \neq \emptyset$.

Let x be an arbitrary element of $S_i \cap S_j$. This implies $x \in S_i$ and $x \in S_j$.

Since $x \in S_i$ and $x \in S_j$, then S_i and S_j must be the same equivalence class because x can belong to only one equivalence class. If such an x exists, then $S_i = S_j$ by Theorem 4.1. Since the subsets are distinct, it follows that $i = j$.

Proof 4.6: Equivalence Classes Form a Partition: Property 3

Property 3 of Definition 4.7 states that the union of all subsets is equal to the original set S. We want to prove that the union of all equivalence classes equals S by showing that the union of the equivalence classes is a subset of S and that S is a subset of the union of the equivalence classes. [10]

Part 1: $S_1 \cup S_2 \cup S_3 \cup ... \cup S_i \subseteq S$

Let x be an element of $S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$. By the definition of union[1], x must belong to at least one equivalence class, S_i.

If $x \in S_i$, then $x \in [x]$ by the definition of an equivalence class. Therefore, $x \in S$.

Since, $x \in S$, then $x \in S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$.

Part 2: $S \subseteq S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$

If $x \in S$, then xRx by the reflexivity property of R. Therefore, $x \in [x]$ by the definition of an equivalence class.

Since $[x]$ is an equivalence class of R, then $S_i = [x]$ for some i.

Since every element $x \in S$ is in some equivalence class $[x]$ and each equivalence class is one of the subsets S_i of the partition, it follows that $S \subseteq S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$.

Therefore, since $S \subseteq S_1 \cup S_2 \cup S_3 \cup ... \cup S_i$ and $S_1 \cup S_2 \cup S_3 \cup ... \cup S_i \subseteq S$, then $S_1 \cup S_2 \cup S_3 \cup ... \cup S_i = S$ by definition of set equality.

What have we proven so far?

We have shown that a set S under an equivalence relation R can be subdivided into equivalence classes that satisfy the properties of a partition - nonempty, pairwise disjoint, and the union of the subsets is equal to the original set. As a result, the equivalence classes form a partition of the set S under the equivalence relation R.

Can we stop our proof here?

We have proven that the equivalence classes form a partition of the set S under the equivalence relation R, completing the proof of Theorem 4.2. However, we have yet to prove the converse -

[1] Definition of Union: The union of sets A and B, denoted as $A \cup B$, is the set of all elements that are in A, in B, or in both A and B. [16]

that a partition of a set S induces an equivalence relation on S for which xRy if and only if x, y are in the same subset of the partition P.

Establishing a complete relationship between equivalence relations and partitions is essential to show the natural correspondence between the two concepts.

Suppose we show that the equivalence classes of a set S under an equivalence relation form a partition P, and the partition P of S gives rise to an equivalence relation. In that case, there is a relationship between equivalence relations and partitions. This relationship is the Fundamental Theorem of Equivalence Relations. This is the next step in our proof.

4.5 Equivalence Relation Induced by a Partition

Definition 4.8: Equivalence Relation Induced by a Partition

R is the equivalence relation induced by a partition P on a set S if it satisfies the following property: xRy if and only if x, y are in the same subset S_i of P [10].
This is expressed as:

$$xRy \leftrightarrow \exists S_i \in P(x, y \in S_i) \tag{4.4}$$

Theorem 4.3: Equivalence Relation Induced by a Partition

If S is a set with the partition $P = \{S_1, S_2, S_3, ..., S_n\}$, where n is the number of subsets in P, and R is a relation induced by P, then R is an equivalence relation such that xRy if and only if $x, y \in S_i$ for some $S_i \in P$.

For the following proofs in this section:

Let S be a set with partition $P = \{S_1, S_2, S_3, ..., S_n\}$ and R be a relation induced by P.

We want to prove that R is an equivalence relation by showing that R is reflexive, symmetric, and transitive.

Proof 4.7: Equivalence Relation Induced by a Partition: Reflexive

As stated in Definition 4.1 and shown in Example 4.1, we need to show that for all elements $x \in S$, the ordered pair $(x, x) \in R$.

Let x be an arbitrary element of S.

Property 1 of equivalence classes, Theorem 4.1, states that every element of S belongs to its own equivalence class.

Since S is partitioned into subsets $S_1, S_2, \ldots, S_n$, every element $x \in S$ belongs to exactly one subset S_i in the partition P. Specifically, $\{x \in S \mid \exists i \in I$ such that $x \in S_i\}$.

Since $x \in S_i$ for some subset $S_i \in P$, by definition of R^1, xRx if and only if x and x are in the same subset S_i.

Since x is in the same subset as itself, xRx. Therefore, R is reflexive.

Proof 4.8: Equivalence Relation Induced by a Partition: Symmetric

As stated in Definition 4.2 and shown in Example 4.3, we need to show that for all elements $x, y \in S$, if xRy, then yRx.

Let x, y be arbitrary elements of S and assume xRy.

By the definition of R, xRy if and only if x and y are in the same subset S_i for some $S_i \in P$.

Since $x \in S_i$ and $y \in S_i$, x and y are in the same subset $S_i \in P$ and therefore xRy.

Since $x \in S_i$ and $y \in S_i$, then yRx by definition of R. Therefore, R is symmetric.

Proof 4.9: Equivalence Relation Induced by a Partition: Transitive

As stated in Definition 4.3 and shown in Example 4.5, we need to show that for all elements $x, y, z \in S$, if xRy and yRz, then xRz.

Let x, y, z be arbitrary elements of S and assume xRy and yRz.

By the definition of R, xRy if and only if x and y are in the same subset S_i for some $S_i \in P$. Similarly, yRz if and only if y and z are in the same subset S_j for some $S_j \in P$.

Since $y \in S_i$ and $y \in S_j$, and the sets in a partition are pairwise disjoint, either $S_i = S_j$ or $S_i \cap S_j = \emptyset$. Since y belongs to both these sets, $S_i \cap S_j \neq \emptyset$ and therefore $S_i = S_j$, as shown in Proof 4.2.

Therefore, x and z are in the same subset S_i, which implies xRz and thus, R is transitive.

We have shown that the relation R induced by a partition P satisfies the three properties of an equivalence relation: reflexive, symmetric and transitive.

Therefore, if S is a set with the partition $P = \{S_1, S_2, S_3, ..., S_n\}$ and R is a relation induced by P, then R is an equivalence relation and this completes the proof of Theorem 4.3.

We have shown that the equivalence classes of a set S are equivalent to its partition and that the partition of S induces an equivalence relation on S. This forms the Fundamental Theorem of Equivalence Relations.

Theorem 4.4: Fundamental Theorem of Equivalence Relations

If R is an equivalence relation on a set S, then the distinct set of equivalence classes of S under R form a partition of S. Conversely, if S is a set with a partition P, then the relation induced by P is an equivalence relation on S such that xRy if and only if x, y are in the same subset of P.

We can describe the equivalence relation that relates the elements of S as *"is in the same subset of the partition"* [4].

4.6 Relationship between Equivalence Relations and Partitions

These concepts of equivalence relations and partitions are often used interchangeably in mathematics to describe the idea of subdividing a set into non-overlapping subsets that are related to each other in a specific way. We can demonstrate this relationship with the following statement:

Definition 4.9: Relationship between Equivalence Relations and Partitions

Let S be a set and P be a partition of S. The relation R induced by P is defined as [10]:

$$\forall x, y \in S \ (xRy \leftrightarrow \exists S_i \in P \ (x \in S_i \wedge y \in S_i)) \tag{4.5}$$

The statement in Definition 4.9 means that two elements x and y are related by R if and only if they are in the same subset of the partition P.

On the left-hand side of Eqn. (4.5), xRy, the equivalence relation R that relates the elements x and y. For all x and y in S, x is related to y by R.

On the right-hand side of Eqn. (4.5), $\exists S_i \in P(x \in S_i \wedge y \in S_i)$, there exists a subset S_i in the partition P that contains the elements x and y.

This relationship shows that the equivalence relation R induced by a partition P relates elements in the same subset of P.

4.7 Bijection between Equivalence Relations and Partitions

To make our proof more precise, we can state the following theorem.

Theorem 4.5: Bijection between Equivalence Relations and Partitions

Let X be a set, let $S = \{R \mid R$ is an equivalence relation on $X\}$, and let $U = \{P \mid P$ is a partition of $X\}$,

There is a bijection $F : S \to U$, such that $\forall R \in S$, if xRy, then x and y are in the same set of $F(R)$ [15].

Let us examine Theorem 4.5:

- S is the collection of all equivalence relations on a set X.
- U is the set of all partitions of X.
- There is a bijection F from S to U, meaning there is a bijection between a set's equivalence relations and partitions.
- For every equivalence relation $R \in S$, if xRy, then x and y are in the same set of $F(R)$.

This theorem further formalises the relationship between equivalence relations and partitions. It shows a one-to-one correspondence between the equivalence relations of a set and its partitions.

Functions and Relations

A function is a special type of relation where each element in the domain is related to exactly one element in the codomain. Note that not all relations are functions.

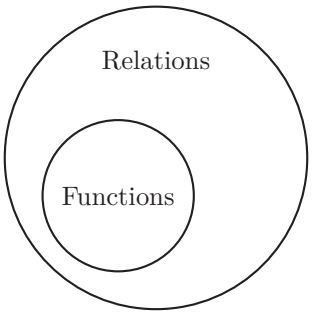


Figure 4.4: Diagram showing the relationship between relations and functions [13].

Definition 4.10: Function

Let X and Y be nonempty sets. A function f from X to Y is where each element $y \in Y$ is assigned to exactly one element $x \in X$ [16].

Definition 4.11: Bijection

A function f is a bijection if it is both **injective** and **surjective**. [16]

Injective: *Each element in the domain is mapped to a distinct element in the codomain.*

Surjective: *Each element in the codomain is mapped to by at least one element in the domain.*

Proof 4.10: Bijection between Equivalence Relations and Partitions

We must first define the function F that maps equivalence relations to partitions.

Let R be an equivalence relation on a set X. We define $F(R)$ as the set of equivalence classes of X under R.
Given R, we define the equivalences classes of X under R, for some element x as:

$$[x]_R = \{y \in X | xRy\}$$

The function F maps an equivalence relation R to the equivalence classes of X under R.

$$F(R) = \{[x]_R \mid x \in X\}$$

$$F(R) \textit{ is the set of all equivalence classes } [x]_R \textit{ for each element } x \textit{ in } X.$$

We want to show that F is a well-defined function, meaning that $F(R)$ is a pairwise disjoint partition of X and covers all its elements. Specifically, the codomain of F is U, the set of all partitions of X.

We have already completed the same steps to show this in our proof of Theorem 4.2.

To summarise, we have shown that $F(R)$ satisfies pairwise disjointness by showing that for any two distinct equivalence classes $[x]_R$ and $[y]_R$, $[x]_R \cap [y]_R = \emptyset$. We have also shown that $F(R)$ covers all elements of X by showing that every element of X is in its own equivalence class, and the union of all equivalence classes is equal to X.

Therefore, F is a well-defined function, $F : S \to U$.

Showing that F is a bijection: injective and surjective

To show that F is injective, we need to show that if $R_1 \not\equiv R_2$, then $F(R_1) \neq F(R_2)$. Let R_1 and R_2 be two distinct equivalence relations on X. Then, at least one pair $x, y \in X \times X$ exists, such that xR_1y and $x\not{R_2}y$.

Assume xR_1y and $x\not{R_2}y$. $F(R_1)$ and $F(R_2)$ are the partitions induced by R_1 and R_2 respectively.

Since xR_1y, x and y are in the same equivalence class under R_1. Therefore, $y \in [x]_{R_1}$. Since $x\not{R_2}y$, x and y are not in the same equivalence class under R_2. Therefore, $y \notin [x]_{R_2}$.

Therefore, $F(R_1) \neq F(R_2)$, and F is **injective**.

To show that F is **surjective**, we need to show that for every pairwise disjoint partition P of X, there exists an equivalence relation R such that $F(R) = P$.

Let P be a pairwise disjoint partition of X. We define R as the relation induced by P such that xRy if x and y are in the same subset of P.

From our proof of Theorem 4.3, we have shown that R is an equivalence relation as it satisfies the properties of reflexivity, symmetry, and transitivity.

By definition of F, $F(R)$ is the set of equivalence classes of X under R. Since R is defined as the relation induced by P, the equivalence classes of X under R are the subsets of P. Therefore, $F(R) = P$ and F is **surjective**.

Since F is both **injective** and **surjective**, it is a **bijection**.

This theorem establishes a one-to-one correspondence between equivalence relations and partitions of a set. It shows that the concepts of equivalence relations and partitions are equivalent ways of describing the same idea of subdividing a set into nonempty, non-overlapping subsets of related elements.

Equivalence relations provide a way to define partitions, which give rise to equivalence relations. This proof allows us to think about equivalence relations as a mathematically precise way to subdivide a set into a partition and consolidates our understanding of the Fundamental Theorem of Equivalence Relations [15].

4.8 Partial Orders

Partial orders are another fundamental mathematical notion with countless applications in various fields. Partial orders model relationships between elements in a set where the relation is reflexive, antisymmetric, and transitive. Order relations provide a formal framework for studying the concept of ordering in a general and abstract setting. In particular, they define a natural structure for representing hierarchies and dependencies.

Many properties, concepts and relationships can be explored in the context of partial orders. We will focus on a select few to provide a general overview of key concepts in this area and their valuable real-world applications.

Before exploring partial orders further, let us revise another property of relations crucial to understanding partial orders.

Definition 4.12: Antisymmetric Relation

A relation R on a set S is antisymmetric if:

$$\forall a \forall b \in S,\ ((a,b) \in R \land (b,a) \in R) \rightarrow a = b \tag{4.6}$$

Proving and Disproving an Antisymmetric Relation

Taking note of the universal quantifier and implication in Eqn. (4.6), using a direct proof, we can show that a relation is antisymmetric by assuming that the antecedent, $((a,b) \in R \land (b,a) \in R)$, is true and proving that the consequent, $a = b$, is also true. To disprove antisymmetry, we can provide a counterexample in which the antecedent is true, but the consequent is false.

Definition 4.13: Partial Order (weak)

A relation $\preceq$ on a set S is called a **partial order** or **partial ordering** if it is reflexive, antisymmetric, and transitive.
For all $x, y, z \in S$ the partial order relation $\preceq$ satisfies the following properties:

1. **Reflexive:** $x \preceq x$ for all $x \in S$.
2. **Antisymmetric:** If $x \preceq y$ and $y \preceq x$, then $x = y$.
3. **Transitive:** If $x \preceq y$ and $y \preceq z$, then $x \preceq z$.

Definition 4.14: Poset

A set S with a partial order relation is called a **partially ordered set** or **poset** [16]. We will denote a partial order relation as $\preceq$ and a poset as $(S, \preceq)$.
The term 'partial' indicates that not all elements in the poset are comparable. Some elements may be related (comparable[1]), while others are not (incomparable[2]). When all elements in a set are comparable, the relation is a total ordering.

4.9 Hasse Diagrams

Hasse diagrams are graphical representations that are useful for visualising finite partial orders. A Hasse diagram is similar to a directed graph; however, the graph does not show all possible edges. We remove all reflexive and transitive edges to simplify the diagram. Additionally, we draw the diagram without any directed edges, as the direction is implied by the assumption that edges point upwards. Removing reflexive and transitive edges allows us to focus on the essential relationships between elements in the poset [16].

[1] ***Comparable:*** *For any two elements x and y in a poset, if $x \preceq y$ or $y \preceq x$, then x and y are comparable. [16]*
[2] ***Incomparable:*** *For any two elements x and y in a poset, if neither $x \preceq y$ nor $y \preceq x$, then x and y are incomparable. [16]*

Example 4.7: Partial Order Relation

Let S be the power set of $\{1, 2, 3\}$. We define the partial order relation $\preceq$ on S as set inclusion, where $A \preceq B$ if A is a subset of B.

Proving $\preceq$ is a Partial Order Relation

We must show that $\preceq$ is reflexive, antisymmetric, and transitive.

Reflexive: For all $A \in S$, $A \subseteq A$ as every set is a subset of itself. Therefore, $\preceq$ is reflexive.

Antisymmetric: If $A \subseteq B$ and $B \subseteq A$, then $A = B$.
Assume $A \subseteq B$ and $B \subseteq A$. Since $A \subseteq B$, $\forall x \in A, x \in B$.
Since $B \subseteq A$, $\forall x \in B, x \in A$. Therefore, $A = B$ by definition of set equality and thus, $\preceq$ is antisymmetric.

Transitive: If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.
Assume $A \subseteq B$ and $B \subseteq C$.
Since $A \subseteq B$, $\forall x \in A, x \in B$. Since $B \subseteq C$, $\forall x \in B, x \in C$. Therefore, $\forall x \in A, x \in C$ and $A \subseteq C$. Thus, $\preceq$ is transitive.

Therefore, $\preceq$ is a partial order relation on the set S.

Comparable and Incomparable Elements in $(S, \preceq)$

As an example, the sets $\{1\}$ and $\{2\}$ are incomparable because neither $\{1\} \subseteq \{2\}$ nor $\{2\} \subseteq \{1\}$. On the other hand, the sets $\{1\}$ and $\{1, 2\}$ are comparable because $\{1\} \subseteq \{1, 2\}$.

Hasse Diagram

We will use a Hasse diagram to represent the set inclusion relation $\subseteq$ on S.

$$S = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

To construct the Hasse diagram, we first draw a directed graph representing $(S, \preceq)$ with all edges pointing upwards. We then remove all reflexive, transitive and directed edges [9].

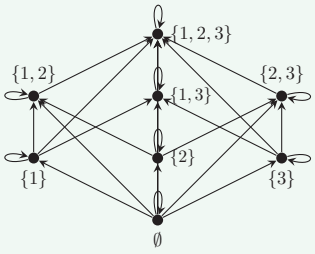


Figure 4.5: Directed Graph

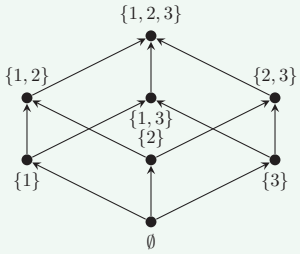


Figure 4.6: Remove reflexive and transitive edges

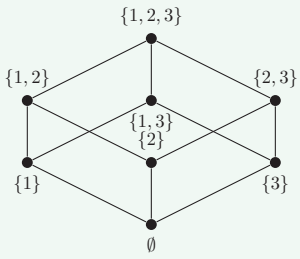


Figure 4.7: Hasse Diagram

4.10 Maximal and Minimal Elements

A poset's maximal and minimal elements are important concepts that describe the extremal properties. A maximal element is an element that is not less than any other element in the poset, while a minimal element is an element that is not greater than any other element in the poset [16].

Definition 4.15: Maximal and Minimal Elements

Let $(S, \preceq)$ be a partially ordered set. An element $x \in S$ is called:

1. **Maximal** if there is no $y \in S$ such that $x \prec y$. Written as $\forall y \in S, \neg(x \prec y)$.
2. **Minimal** if there is no $y \in S$ such that $y \prec x$. Written as $\forall y \in S, \neg(y \prec x)$.

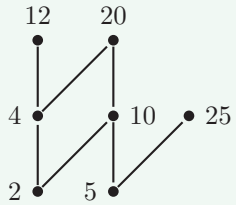
A poset can have multiple maximal and minimal elements.

Note: The notation $x \prec y$ denotes that $x \preceq y$ and $x \neq y$.

The maximal and minimal elements can be identified using the Hasse diagram. The maximal elements are the 'top' elements in the diagram, meaning the nodes adjacent are below them. Conversely, the minimal elements are the 'bottom' elements, meaning the nodes adjacent are above them in the diagram [20]. This method is valid, though it should be used with caution, as it depends on the accurate representation of the poset in the Hasse diagram.

Example 4.8: Maximal and Minimal Elements in a Poset

Consider the poset $(\{2, 4, 5, 10, 12, 20, 25\}, |)$ and the Hasse Diagram below. Identify the maximal and minimal elements in the poset [16].



Solution

The poset is defined by the divisibility relation, where $x|y$ if x divides y.

From the Hasse diagram, we can identify the maximal elements as 12, 20, and 25, while the minimal elements are 2 and 5.

To show that 12 is maximal, we need to show that there is no $y \in S$ such that $12 \mid y$ and $y \neq 12$. The only element in the poset that 12 divides is itself, so 12 is maximal. Similarly, 20 and 25 are maximal as they only divide themselves.

To show that 2 is minimal, we need to show that there is no $x \in S$ such that $x \mid 2$ and $x \neq 2$. The only element that divides 2 is itself, so 2 is minimal. Similarly, 5 is minimal as it is only divisible by itself.

4.11 Restriction of Partial Order Relation to Subset

Lemma 4.1

If $(S, \preceq)$ is a poset and $A \subseteq S$, then the restriction of $\preceq$ to A is a partial order relation [18].

Proof 4.11: Restriction of Partial Order Relation to Subset

Since $\preceq$ is a partial order relation on S, by Definition 4.13, it is reflexive, antisymmetric, and transitive.

To prove Lemma 4.1, we need to show that the restriction of $\preceq$ to A retains these properties.

Reflexivity: For all $a \in A$, $a \preceq a$ since $\preceq$ is reflexive.

Antisymmetry: If $a \preceq b$ and $b \preceq a$ for $a, b \in A$, then $a = b$ since $\preceq$ is antisymmetric.

Transitivity: If $a \preceq b$ and $b \preceq c$ for $a, b, c \in A$, then $a \preceq c$ since $\preceq$ is transitive.

Therefore, the restriction of $\preceq$ to A is a partial order relation, and A is a poset.

4.12 Existence of a Minimal Element

Lemma 4.2

Every finite nonempty poset $(S, \preceq)$ has at least one minimal element.

Proof 4.12: Lemma 4.2: Existence of Minimal Element

$P(n) =$ "Every finite nonempty poset with n elements has at least one minimal element." By induction, we will prove $P(n)$ is true for all $n \in \mathbb{Z} : n \geq 1$.

Base Case: Proving $P(1)$ is true.

For $n = 1$, the poset has only one element. This element is trivially the minimal element because there are no other elements to compare it to. Therefore, $P(1)$ is true.

Inductive Hypothesis: Assume $P(k)$ is true.

Assume that $P(k)$ is true for some $k \in \mathbb{Z} : k \geq 1$. That is, every finite nonempty poset with k elements has at least one minimal element.

Inductive Step: $\forall k \in \mathbb{Z} : k \geq 1 : P(k) \rightarrow P(k+1)$

We need to show that if $P(k)$ is true, then $P(k+1)$ is true. That is, every finite nonempty poset with $k+1$ elements has at least one minimal element.

Consider a poset $(S, \preceq)$ with $k+1$ elements. Let $x \in S$ be any element of the poset.

Consider the subset $S' = S \setminus \{x\}$, which has k elements. By Lemma 4.1, the restriction of $\preceq$ to S' is a partial order relation.

By the inductive hypothesis, the poset $(S', \preceq)$ has at least one minimal element.

Let $m \in S'$ be a minimal element of $(S', \preceq)$.

Now, consider the element m in the original poset $(S, \preceq)$. We must show that m is a minimal element in $(S, \preceq)$ or that another minimal element exists in $(S, \preceq)$.

Case 1: If m is a minimal element in $(S, \preceq)$, then we are done.
Case 2: If m is not a minimal element in $(S, \preceq)$, then there exists an element $y \in S$ such that $y \preceq m$ and $y \neq m$.

Consider the following two subcases for y as the minimal element:

Subcase 1: $y \in S'$. m is the minimal element in $(S', \preceq)$.

Subcase 2: $y = x$. The only other possibility is that y is the element x that we removed to form S'. In this case, x is the minimal element in $(S, \preceq)$.

Thus, the minimal element in $(S, \preceq)$ is either m or x. Therefore, $P(n)$ is true for all $n \in \mathbb{Z} : n \geq 1$.

The existence of a maximal element in a finite nonempty poset can be proven using a similar induction argument.

4.13 Greatest and Least Elements

Definition 4.16: Greatest and Least Elements

Let $(S, \preceq)$ be a partially ordered set. An element $x \in S$ is called:

Greatest if $\forall y \in S, y \preceq x$.

Least if $\forall y \in S, x \preceq y$.

If an element is greater than any other in the poset, it is called the **greatest** element. Similarly, if an element is less than any other in the poset, it is called the **least** element.

A maximal element is not necessarily the greatest element. Similarly, a minimal element is not necessarily the least element.

If an element is the greatest or least element, it must be comparable to all other elements in the set [6]. This crosses over with the concept of total orders, where all elements are comparable. However, it applies to understanding the differences between maximal and greatest, as well as minimal and least elements.

Note: When a set is totally ordered, the terms maximal and greatest are equivalent, as are minimal and least.

Theorem 4.6: Uniqueness of Greatest and Least Elements

If a poset has a greatest element, it is unique. Similarly, if a poset has a least element, it is unique. [20]

Proving uniqueness: If there are two elements x and y that both satisfy the definition of the greatest or least element in a poset, then we want to show that $x = y$.

For the following two proofs in this section:

Let S be a partially ordered set with the partial order relation $\preceq$.

Proof 4.13: Uniqueness of Greatest Element

Assume $x \in S$ is the greatest element. By Definition 4.15, $\forall s \in S, s \preceq x$.

Similarly, assume $y \in S$ is another greatest element. By definition, $\forall s \in S, s \preceq y$.

Since x is the greatest element, $y \preceq x$. Similarly, since y is the greatest element, $x \preceq y$. As a result, $x = y$ by Definition 4.13 and the antisymmetric property.

Therefore, x and y are equivalent, and the greatest element is unique.

Proof 4.14: Uniqueness of Least Element

Assume $x \in S$ is the least element. By Definition 4.15, $\forall s \in S, x \preceq s$.

Similarly, assume $y \in S$ is another least element. This means $\forall s \in S, y \preceq s$.

Since x is the least element, $x \preceq y$. Similarly, since y is the least element, $y \preceq x$. As a result, $x = y$ by Definition 4.13 and the antisymmetric property.

Therefore, x and y are equivalent, and the least element is unique.

4.14 Topological Sorting

Topological sorting is a fundamental notion that arranges the elements of a poset in a linear order such that for every pair of elements x and y in the poset, if $x \preceq y$, then x appears before y in the linear order. This linear order is called a topological ordering of the poset [16].

Topological sorting is used in various applications, such as scheduling tasks in project management systems, resolving dependencies in package management systems, and determining precedence in directed acyclic graphs (DAGs). A DAG is a directed graph with no directed cycles, and the topological sort of a DAG is a linear ordering of its vertices such that for every directed edge uv, vertex u comes before v in the ordering [5].

We have proven the restriction of a partial order relation to a subset in Lemma 4.1 and the existence of a minimal element in a finite nonempty poset in Lemma 4.2.

We will introduce an example of a topological sorting algorithm and use the results of Lemma 4.1 and Lemma 4.2 to prove the correctness of the algorithm and the existence of a topological sort for every finite poset.

Definition 4.17: Topological Ordering

A topological ordering of a poset $(S, \preceq)$ is a total ordering R, compatible with $\preceq$, such that if $x \preceq y$, then xRy.

Topological Sorting Algorithm

Example 4.9: Topological Sorting Algorithm

The following pseudocode describes the topological sorting algorithm for a finite poset $(S, \preceq)$ [16]

Input: A finite poset $(S, \preceq)$

$S_k = S$

while $S_k \neq \emptyset$:

    Select a minimal element x_k from S (which exists by Lemma 1)

    $S_{k+1} = S_k \setminus x_k$ (remove x_k from S_k)

    $k = k + 1$

return $x_1, x_2, \ldots, x_n$ (the topological ordering) where $n = |S|$

By continuously removing the minimal element from the poset, we return a compatible total ordering that is a topological ordering of the poset. Because the poset is finite, the algorithm

must terminate. The desired output is the sequence of elements $x_1, x_2, \ldots, x_n$ that represents the topological ordering of the poset.

When we remove the minimal element from the poset S_k, we are left with a smaller subset $(S_k \setminus x_k)$ of S that still satisfies the properties of a partial order relation. The restriction of $\preceq$ to the subset $(S_k \setminus x_k)$ is also a partial order relation.

Existence of a Topological Sort

Lemma 4.1 and Lemma 4.2 are essential in proving the correctness of the topological sorting algorithm, defined in Example 4.9, and the existence of a topological sort for every finite poset.

Theorem 4.7: Existence of a Topological Sort

Every finite poset has a topological sort [11].

Proof 4.15: Topological Sorting Algorithm

We want to prove the correctness of the topological sorting algorithm by showing that the algorithm terminates and the output is a topological ordering of the poset $(S, \preceq)$.

Termination of the Algorithm

Since the poset $(S, \preceq)$ is finite, the algorithm must terminate after a finite number of iterations. This is because the algorithm removes one element (x_k) from S_k in each iteration, and the size of S_k decreases by one in each step until $S_k = \emptyset$.

Therefore, the algorithm will terminate after n iterations, where n is the number of elements in the poset, $|S| = n$.

Correctness of the Output

We want to show that the algorithm's output is a topological ordering of the poset $(S, \preceq)$.

Minimal element selection:

At each step of the algorithm, we select a minimal element x_k from the poset S_k. By Lemma 2, every finite nonempty poset has at least one minimal element. Removing the minimal element x_k from S_k does not violate the properties of the partial order relation $\preceq$ shown by Lemma 1, so the algorithm can continue to select the minimal element in each iteration.

Retaining the partial order relation:

$P(n)$ = "For a poset $(S, \preceq)$ with n elements, if $x_i \prec x_j$ for $x_i, x_j \in S$, then the algorithm produces a topological ordering such that $i < j$."

By induction, we will prove that $P(n)$ is true for all $n \in \mathbb{Z} : n \geq 1$.

Base Case $P(1)$:

For $n = 1$, the poset $(S, \preceq)$ has exactly one element, and there are no other elements to compare it to, so the statement $P(1)$ is trivially true.

Inductive Hypothesis: Assume $P(k)$ is true.

Assume that $P(k)$ is true for some $k \in \mathbb{Z} : k \geq 1$. That is, for any poset $(S, \preceq)$ with $|S| = k$, if $x_i \prec x_j$ for $x_i, x_j \in S$, then $i < j$.

Inductive Step: $\forall k \in \mathbb{Z} : k \geq 1 : P(k) \rightarrow P(k+1)$

We need to show that if $P(k)$ is true, then $P(k+1)$ is true. That is, for any poset $(S, \preceq)$ with $|S| = k+1$, if $x_i \prec x_j$ for $x_i, x_j \in S$, then $i < j$.

Consider a poset $(S, \preceq)$ with $k+1$ elements. Let x_1 be the minimal element selected in the first iteration of the algorithm.

After we remove the minimal element x_1 from S, we are left with a poset $S_2 = S \setminus \{x_1\}$ with k elements. By the inductive hypothesis, since $|S_2| = k$, then $P(k)$ is true for S_2. Therefore, the induction hypothesis guarantees that the algorithm produces a topological ordering of S_2 such that $i < j$ if $x_i \prec x_j$ for $x_i, x_j \in S_2$. Specifically, $S_2 = x_2, x_3, \ldots, x_{k+1}$. Since x_1 is the minimal element of S, by definition, $\forall x \in S_2, \neg(x \prec x_1)$. Therefore, x_1 precedes all elements in S_2 and the topological ordering of S is $x_1, x_2, x_3, \ldots, x_{k+1}$.

To verify the topological ordering of S, we need to show that it is consistent with the partial order relation $\preceq$ such that $i < j$ if $x_i \prec x_j$ for $x_i, x_j \in S$.

Consider any $x_i, x_j \in S$ such that $x_i \prec x_j$.

Case 1: Neither x_i nor x_j is x_1. Then, $x_i, x_j \in S_2$ and $i < j$ by the induction hypothesis.
Case 2: $x_i = x_1$. Since x_1 is the minimal element, x_1 precedes all other elements in S as shown above. Therefore, $i < j$.
Case 3: $x_j = x_1$. Since $x_i \prec x_j$, this case is not possible as x_1 is the minimal element and precedes all other elements in S.

We have verified that the topological ordering of S satisfies $P(k+1)$.

Therefore, the algorithm produces a topological ordering of the poset $(S, \preceq)$ such that $i < j$ if $x_i \prec x_j$ for $x_i, x_j \in S$ for all $n \in \mathbb{Z} : n \geq 1$.

The correctness of the topological sorting algorithm is established by proving that the algorithm terminates and the output is a topological ordering of the poset. This, in turn, proves the existence of a topological sort for every finite poset as stated in Theorem 4.7.

We have explored the concepts of partial orders, maximal and minimal elements, greatest and least elements, and topological sorting. These concepts are fundamental in mathematics and computer science, providing a formal framework for studying relationships between elements in a set and ordering them in a linear sequence. There are significantly more properties, theorems, and algorithms that can be explored in the context of partial orders, such as the lattice structure, chain and anti-chain, order isomorphism, and more.

We have mentioned some applications throughout our exploration of partial orders and will now examine some real-world examples that demonstrate the significance of these concepts in various fields.

4.15 Applications

Equivalence Relations

The Fundamental Theorem of Equivalence Relations has applications in numerous mathematics and computer science fields. In the following section, we will explore several noteworthy use cases.

Myhill Nerode Theorem

The Myhill Nerode Theorem states that a language L is regular if and only if the equivalence relation of L partitions the set of all strings over an alphabet into a finite number of equivalence classes. This theorem is used in automata theory and formal languages to determine the regularity of a language [23].

Equivalence Class Partitioning

Equivalence class partitioning(ECP) is a software testing technique used to group test cases based on the same input-output behaviour. This concept is used to reduce redundancy in testing and improve test coverage. The idea behind the ECP is to partition the input domain into equivalence classes and select representative test cases from each class. The input data could be partitioned into valid and invalid data subsets; each value of every equivalent subset must exhibit the same behaviour.

For example, if we look at a simple password field requiring a minimum of 8 characters and a maximum of 16 characters (Fig. 4.8). In that case, we can partition the input domain into valid and invalid passwords [7].

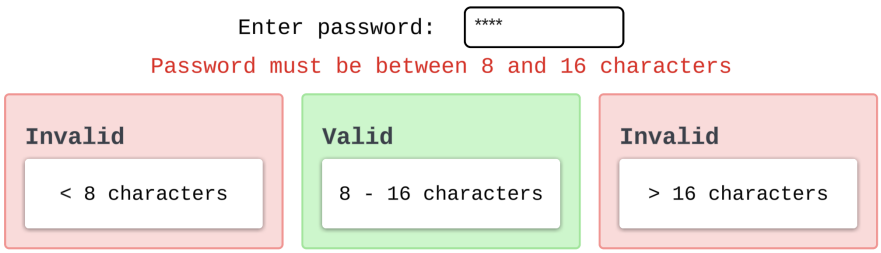


Figure 4.8: Example password validation. Image created using Lucidchart (lucid.co) based on [7].

The valid partition could be further divided into equivalence classes based on measures such as password strength [2].

Data Clustering

In data analysis and machine learning, data clustering is a technique used to group similar objects by partitioning the data into subsets based on similarity measures. Equivalence relations define the similarity between data points. For example, two data points x and y are equivalent if they satisfy a similarity condition, such as being within a certain distance [21].

Partial Orders

Partial orders are prevalent in various fields, such as computer science, mathematics, and engineering. They provide a formal structure for modelling relationships between elements in a set and are used to study and define hierarchies, dependencies, and ordering. The following section will explore some real-world applications of partial orders and discuss how they are utilised to solve problems in different domains.

Topological Sorting

Topological sorting is often used in programming to perform depth-first search (DFS) on directed acyclic graphs (DAGs). This is seen in Makefiles used to build software projects, where the *make* command reads a Makefile and its specified rules to create executable files from the source code.

By performing a topological sort on the dependency graph, the *make* utility can determine the order in which targets should be built, ensuring that dependencies are built before the target that depends on them. This prevents circular dependencies and ensures an efficient build process [24].

Another application of topological sorting is in scheduling tasks in project management systems. By representing tasks as nodes in a DAG and dependencies as edges, topological sorting can determine the order in which tasks should be executed to meet deadlines and optimise resource allocation. This is useful in industries such as construction, software development, and manufacturing, where tasks must be completed in a specific order [1].

The table below illustrates a basic example of a project management system, including tasks and dependencies. The minimal elements are the tasks that must be completed first, as they have no dependencies. By performing a topological sort, we can determine the optimal order in which tasks should be executed to complete the project. Note that there is more than one valid topological ordering for this example.

In this case, the topological sort is $C \prec A \prec E \prec B \prec F \prec D \prec G$.

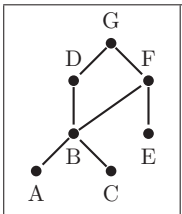
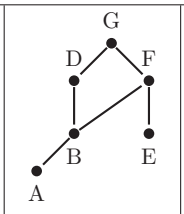
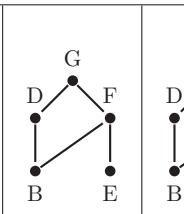
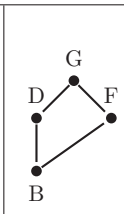
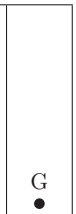
						
C	A	E	B	F	D	G

Table 4.1: Topological Sorting of Tasks Example [16]

The applications discussed do not solely rely on equivalence relations or partial orders. Still, the notions are fundamental to understanding them and are essential building blocks in these fields.

4.16 Summary

In this article, we have studied and explored equivalence relations and partial orders, providing a broader understanding of these concepts. We have highlighted their significance and vastness in mathematics and other fields through our array of proofs and discussion of real-world applications.

Equivalence relations model relationships between elements in a set based on a specific property, whereas partial orders define hierarchies and dependencies. The applications of these fundamental concepts are diverse and extend beyond the examples we have discussed.

Equivalence relations and partial orders underpin many systems and processes we encounter in our day-to-day lives, and we often do not realise or comprehend the underlying mathematical structures that govern these relationships and hierarchies. Studying these concepts can help us gain a deeper insight into the world around us, the tools we use, and the systems we interact with.

Context

This article was adapted from a report submitted as part of the coursework for SIT192.

About the Author

Kate Suraev is a first year undergraduate student pursuing a Bachelor's degree in Computer Science. As she progresses in her studies, she is interested in exploring the intersection of mathematics and computer science, particularly how mathematical concepts can be applied to address complex and intriguing real-world problems.

Acknowledgements

I would like to express my gratitude to Julien Ugon for his patience, guidance, and support during the writing process of this report. Julien inspired and encouraged me to think beyond the scope of the content, challenge myself, and explore mathematics in a new and engaging way. I am very thankful for his mentorship and the opportunity to contribute to this yearbook.

References

Unless otherwise stated, all diagrams created using Tikz [22].

[1] Topological sorting. *GeeksforGeeks*, 2024. Available online: https://www.geeksforgeeks.org/topological-sorting/.

[2] Equivalence partitioning technique. *TPoint Tech*, 2024. Available online: https://www.tpointtech.com/equivalence-partitioning-technique-in-black-box-testing.

[3] D. Arnold. Multiplication and division of integers. 2024.

[4] G. Baker. Equivalence relations [Online Notes], 2013.

[5] M. Burmester. Topological sort [Online Notes]. *Florida State University*, 2000.

[6] J. F. Feinstein. Partial orders: Maximality and minimality in partially ordered sets [Online Notes]. *University of Nottingham*, 2004.

[7] Equivalence Partitioning Method, 2024. Available online: https://www.geeksforgeeks.org/equivalence-partitioning-method/.

[8] T. Jenkyns and B. Stephenson. *Fundamentals of discrete math for computer science : a problem-solving primer (2nd Edition)*. Springer, 2018.

[9] V. Kaburlasos. Granular enhancement of fuzzy art/som neural classifiers based on lattice theory. *Studies in Computational Intelligence*, 67:3–23, 1970.

[10] H. Kwong. Equivalence relations and partitions. *LibreTexts Mathematics*, 2021.

[11] E. Lehman, T. Leighton, and A. Meyer. Relations and partial orders [Lecture Notes]. *MIT OpenCourseWare*, 2010. Available online: https://ocw.mit.edu/courses/6-042j-mathematics-for-computer-science-fall-2010/resources/mit6_042jf10_chap07/.

[12] S. Lipschutz and M. Lipson. *Schaum's outline of discrete mathematics (4th Edition)*. McGraw-Hill, 2022.

[13] D. E. McAdams. Relation. *All Math Words Encyclopedia. Life is a Story Problem LLC*, 2019.

[14] Standford University Department of Computer Science. Guide to proofs on discrete structures [Online Readings]. *Stanford University*, 2013.

[15] M. Radcliffe. Equivalence relations [Lecture Notes]. *Carnegie Mellon University*, 2019.

[16] Kenneth H. Rosen. *Discrete Mathematics and its Applications (8th Edition)*. McGraw-Hill Education, New York, 2019.

[17] T. R. Shemanske. Partitions and equivalence relations [Online Notes]. In *Abstract Algebra Refresher: Review, Amplification, Examples*. Dartmouth College, 2025.

[18] K. Siegrist. Partial orders. *LibreTexts Mathematics*, 2022.

[19] T. Sundstrom. Equivalence relations. *LibreTexts Mathematics*, 2022.

[20] J. Sylvestre. Maximal and minimal elements. *LibreTexts Mathematics*, 2022.

[21] P. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Pearson Addison Wesley, 2005.

[22] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[23] L. Trevisan. Notes on the myhill-nerode theorem [Online Notes]. *Stanford University*, 2014.

[24] Y. Zhou. Tutorial on writing makefiles [tutorial notes]. *Colorado State University*, 2003.

5

The Merkle-Hellman knapsack algorithm

Caleb Cross

Abstract

The article discusses the Merkle-Hellman Knapsack algorithm, a public-key cryptosystem. It explains how the algorithm works, providing proof for all the claims. It also contains information regarding the historical impact and significance of the public-key cryptosystem.

5.1 Introduction

Introduced by Ralph Merkle and Martin Hellman in 1978 [12], the Merkle-Hellman Knapsack Algorithm was one of the first attempts at creating a public-key cryptosystem. It was revolutionary in its approach, basing its security on the computational difficulty of solving the knapsack problem. Whilst the trap-door knapsack did not live up to the 'computationally infeasible' claims made by Merkle and Hellman, it was still a cryptographic feat, nonetheless. The Merkle-Hellman algorithm played a crucial role in the development of public-key cryptography and continues to be an important subject of study for cryptography students and researchers. This article will explore the fundamental principles of the Merkle-Hellman Knapsack Algorithm, its key generation process, encryption and decryption methods.

5.2 Preliminaries

Public-key cryptosystems

The Merkle-Hellman Knapsack Algorithm is a public-key, also known as an asymmetric-key, cryptosystem. This means that two keys are involved when securely communicating using this cryptosystem, a public and a private key. This cryptosystem implements a one-way philosophy, with the public key only being used for encryption and the private key only being used for decryption [3].

5.3 Principles of the Merkle-Hellman Knapsack Algorithm

The Knapsack problem

As the name suggests, a core underlying concept that went into the creation of the Merkle-Hellman Knapsack Algorithm is the knapsack problem.
The knapsack problem is defined as follows: Given a set of items, each with a specific weight and value, determine which items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible [10].

Example 5.1: Knapsack example

Imagine you have a knapsack (a small bag) that can at most hold 15kg. You would like to pack various items in this bag to sell at the market.

Table 5.1: Knapsack example

Item	Weight	Value
Clay pot	12 kg	4 dollars
Clay jar	2 kg	2 dollars
Clay vase	4 kg	10 dollars
Clay cup	1 kg	1 dollar
Clay tray	1 kg	2 dollars

The knapsack problem posed in this example is: what items should you take such that you pack the maximum value in your 15kg knapsack?
The solution to this would be three vases and three trays.
The general form of this problem is known to be NP-complete, which is what motivated Merkle and Hellman to use it as the base of their security.

Definition 5.1: NP

Nondeterministic Polynomial (NP) time is a class of yes or no problems whose solutions can be verified in polynomial time [5]. A polynomial-time algorithm runs in time that grows as a polynomial function of the size of its input [9]. Algorithms solved in polynomial-time are said to be "fast" [15].
NP-complete is a subsection of problems solved in NP, and it encompasses the hardest NP problems [16]. All problems classed as NP-complete have two distinct properties [8]:

- Their solutions can be verified fast (in polynomial time).
- An efficient algorithm that can solve any NP-complete problem can be used to efficiently solve all other problems in NP.

There currently exists no way to quickly find a solution to an NP-complete problem [4].

The Knapsack Problem's relationship with the cryptosystem

The Merkle-Hellman is based on the knapsack problem [12], but more specifically, on a variation of the knapsack problem known as the subset sum problem (which is also NP-complete). This simplified version of the knapsack problem has each item's weight equivalent to its value. The goal is to find a subset of numbers from a given set that adds up to a specific target sum. In a generic knapsack, finding the subset that adds to the target numbers is computationally infeasible.

However, if the knapsack (the set of numbers) is super-increasing, the problem is considered easy and can be solved quickly.

Definition 5.2: Super-Increasing Sequence

A super-increasing sequence is a sequence of numbers (s_0, s_1, ..., s_n) where each element is larger than the sum of all elements that proceed it. This is formally defined as $s_j > \sum_{i=0}^{j-1} s_i$.

Figure 1.1 showcases a sagemath function which creates a random super-increasing sequence when given a starting number and a sequence size.

```
import random
def superincreasing_sequence(sequence_size, starting_element):
    s = [starting_element] #starting element in the sequence
    for i in range(1, sequence_size):
        s.append(random.randint(sum(s)+1, sum(s)*2)) #random int between:
                                        #the sum of the sequence, sum(s)
                                        #2 x sum(s) (any upper limit > sum(s) works)
    return s

superincreasing_sequence(5, 7)
```

```
[7, 13, 27, 77, 134]
```

Figure 5.1: Super-increasing sequence in sagemath

5.4 Greedy method

When decrypting with the Merkle-Hellman algorithm, we need to use a greedy method. More specifically, we need to use a greedy method when solving the subset sum problem of a super-increasing knapsack. We can use the target variable calculated earlier in the decryption process to select the components in the super-increasing sequence. The greedy method is as follows [2]:

1. Start by selecting the largest value in the subset that does not exceed the target value. This value can't be selected again.
2. Select the next largest value (the largest unselected value) that when added onto the total sum will not exceed the target value.
3. Repeat step 2 until you have reached the target value.

The code in Figure 5.2 implements this greedy algorithm in sagemath.

```
def greedy(superincreasing_sequence, target_variable):
    message = ''
    for si in reversed(superincreasing_sequence): #reverse s to get largest values first
        if  target_variable >= si: #if p is greater than or equal to element in s

            if target_variable - si == 0: #if the loop will end, dont append ", " to message
                message += f"{si}"
                break

            target_variable -= si #subtract the element from p so we cant use it again
            message += f"{si}, "

    return message

greedy([2, 3, 7, 13], 18)
```

```
'13, 3, 2'
```

Figure 5.2: Greedy method in sagemath

Example 5.2: Greedy method

Sequence: *2, 3, 7, 13*
If the target variable is *10*: *7* and *3* are chosen.
If the target variable is *18*: *13*, *3* and *2* are chosen.

5.5 Key generation process

The process described in this section largely comes from [6] and [14].

Private key

1. Create a super-increasing sequence of integers, $s = s_0, s_1, \ldots, s_n$, where n represents the size of the sequence.
2. Select an integer for modulus, m, such that m is larger than the sum of all the elements in s. That is, select an m where $m > \sum_{i=0}^{n-1} s_i$ is true.
3. Select a multiplier, an integer w, that is smaller than m, and is coprime with m.
   In other words, select a w such that $w < m$ and $\gcd(w, m) = 1$.
4. Compute the modular inverse of w, $w^{-1} \pmod m$.
   $w \times w^{-1} \equiv 1 \pmod m$
5. **The private key will consist of** (m, w^{-1}, s)

The code in Figure 5.3 demonstrates private key generation in sagemath.

```
import random as r
def generate_private_key(s_size):
    if (s_size <= 1):
        return "Super-increasing sequence, s, is too small!"

    #super increasing sequence, s
    s = [r.randint(1, s_size)]

    for i in range(1, s_size):
        s.append(r.randint(sum(s) + 1, sum(s) * 2))

    #modulus, m
    m = r.randint(sum(s) + 1, sum(s) * 2)

    #multiplier, w
    w = r.randint(1, m - 1)

    while gcd(w,m) != 1:
         w = r.randint(1, m - 1)

    inv_w = inverse_mod(w, m)


    return s, m, inv_w
```

Figure 5.3: Private key generation in sagemath

Public key

1. Compute $e_i \equiv w \times s_i \pmod{m}$, for all elements s_i in the initial super-increasing sequence s.
2. Repeating this computation for all elements in s will produce the sequence e.
3. **The public key is sequence e.**

We next generate the public key in sagemath (Figure 5.4).

```
def generate_public_key(private_key):
    s, m, inv_w = private_key

    w = inverse_mod(inv_w, m)
    e = []
    for i in s:
        e.append(mod((w*i), m))

    return e
```

Figure 5.4: Public key generation in sagemath

Example 5.3: Private key generation

1. The super-increasing sequence is given as $s = \{2, 3, 7, 13\}$.
2. The modulus is $m = 29$.
   This satisfies the key generation conditions as $29 > \sum_{i=0}^{4-1} s_i$
3. The multiplier is $w = 24$.
   This satisfies the key generation conditions as $24 < 29$ and $\gcd(29, 24) = 1$
4. To find the modular inverse w^{-1} of w modulo m, use the extended Euclidean algorithm:

   Table 5.2: Extended Euclidean Algorithm Steps

$\gcd(29, 24)$	29	24
$29 - 24 \times 1 = 5$	$1 - 0 \times 1 = 1$	$0 - 1 \times 1 = -1$
$24 - 5 \times 4 = 4$	$0 - 1 \times 4 = -4$	$1 - (-1) \times 4 = 5$
$5 - 4 \times 1 = 1$	$1 - (-4) \times 1 = 5$	$-1 - 5 \times 1 = -6$

   $\therefore w^{-1} \equiv -6 \equiv 23 \pmod{29}$
5. private key $= (m, w^{-1}, s) = (29, 23, (2, 3, 7, 13))$

Example 5.4: Public key generation

$e_i \equiv w \times s \pmod{m}$
$e_1 \equiv 24 \times 2 \pmod{29} \equiv 19$
$e_2 \equiv 24 \times 3 \pmod{29} \equiv 14$
$e_3 \equiv 24 \times 7 \pmod{29} \equiv 23$
$e_4 \equiv 24 \times 13 \pmod{29} \equiv 22$
$e = (19, 14, 23, 22)$

5.6 Communication process with Merkle-Hellman

The process described in this section largely comes from [14] and [7].

Encryption process

The basic structure to ***encrypt*** with the Merkle-Hellman knapsack algorithm is as follows:

Compute $c = \sum_{i=0}^{n-1} plaintext_i \times e_i$, where *plaintext* is a binary string

What happens in the plaintext has a size different from n (size of super-increasing sequence)?

If the plaintext's size, z, is a multiple of n, we can simply split up the sequence into z/n number of n sized fragments.

If z (the size of the plaintext) is not a multiple of n, we add padding to the message such that z is large enough to create a whole number when divided by n. Padding can consist of a predetermined character, phrase, or other identifier that can be decided by between communicating parties to signify the end of a message.

Example 5.5: Plaintext has differing size to super-increasing sequence

If Bob wants to send Alice the message 01101101 and the super-increasing sequence only contains 4 characters, what does he do?

In this scenario, Bob would split the plaintext into two n sized portions, 0110 1101, which he would then encrypt.

What if Bob's plaintext is 011011?

In this case, Bob would add two predetermined characters to the end of his message, allowing him to break his plaintext down into n sized portions. This may look like 0110 1100.

A basic version of this is implemented in sagemath in Figure 5.5 below.

```
def encrypt(message, public_key):
    return sum(ei for ei, mi in zip(public_key, message) if mi == '1')
```

Figure 5.5: Encryption process in sagemath

Decryption process

The basic structure to ***decrypt*** with the Merkle-Hellman knapsack algorithm is as folows:

1. Compute the target variable used in a greedy method $p \equiv c \times w^{-1} \pmod{m}$.
2. Use a greedy method to find the subset of s that when added together sum to p, our target variable.
3. Replace the elements in s that sum to p with 1. Replace the remaining terms with a 0. The resulting string of 1's and 0's is the plaintext.

We next implement the decryption process in sagemath (Figure 5.6).

```
def decrypt(ciphertext, private_key):
    s, m, inv_w = private_key
    w = inverse_mod(inv_w, m)

    p = (ciphertext * inverse_mod(w, m)) % m
    message = ''
    for si in reversed(s): #reverse s to get largest values first
        if p >= si: #if p is greater than or equal to element in s
            p -= si #subtract the element from p and place append a 1 to the message
            message = '1' + message
        else:
            message = '0' + message
    return message
```

Figure 5.6: Decryption process in sagemath

Let us illustrate the encryption and decryption process with an example.

Example 5.6: Encryption process

Bob wants to send Alice the message 0110 1101

Alice's private key was computed in Example 5.3: (29, 23, (2, 3, 7, 13)).

Alice's public key, e, was computed in Example 5.4: (19, 14, 23, 22).

Seen as though the public key contains only 4 numbers, we need to separate the plaintext into 2 4-bit blocks.

Block 1 = 0110

Block 2 = 1101

$$c = \sum_{i=0}^{n-1} plaintext_i \times e_i$$

$$c_1 = 0 \times 19 + 1 \times 14 + 1 \times 23 + 0 \times 22 = 37$$
$$c_2 = 1 \times 19 + 1 \times 14 + 0 \times 23 + 1 \times 22 = 55$$
$$c = (37, 55)$$

Now we have our c value, where (37, 55) is the ciphertext.

In Example 5.4's public space, there is a non-super-increasing sequence of numbers (a generic knapsack) as the public key. After Example 5.6, the public space now also includes the ciphertext, c. Does this mean that the plaintext is at risk? No. Computing the plaintext in this instance is equivalent to solving the previously mentioned subset sum problem, see Section 5.3 - The relationship with the knapsack problem. In other words, it is equivalent to trying to find an efficient method to find a subset of the public key $(19, 14, 23, 22)$ that sums up to 37 or 55 (the ciphertext values), which is currently computationally infeasible. Note that a greedy method does not work in this instance – the knapsack is not super-increasing.

To demonstrate how quickly this problem gets out of reach to solve, try to find the subset in e that adds to 1776, where $e = (483, 473, 433, 373, 233, 476, 379, 225)$.

The answer is $(473, 233, 476, 379, 225)$. This is difficult to determine using only 8 numbers, now imagine if our public key had 100+ numbers in it.

Example 5.7: Decryption process

In order to decrypt Bob's message, Alice must compute the target variable used in a greedy method, p.

1. $p \equiv c \times w^{-1} \pmod{m}$
   $p_1 \equiv 37 \times 23 \pmod{29} \equiv 10$
   $p_2 \equiv 55 \times 23 \pmod{29} \equiv 18$

2. Alice now needs to use a greedy method to find the subsets of $s = (2, 3, 7, 13)$ that sum to her target variables p. This will reveal to her the plaintext.

   Following a greedy method, she needs to find the largest value in s that is less than or equal to her first target variable $p_1 = 10$. She selects 7. Because she has not reached her target variable, she locates the second largest value that when added to 7 doesn't exceed 10. She chooses 3. She has now hit her target variable and stops the process with 7 and 3 selected.

   When she replaces her selected values in s with $1's$ and her remaining values with $0's$ she gets $0, 1, 1, 0$, which is the first block of plaintext.

   Repeating the same steps for p_2 she get $1, 1, 0, 1$, which is the second block of plaintext.

We ran the decryption algorithm twice in this example because there are two blocks of plaintext.

Trap Doors

The brilliance of the Merkle-Hellman approach was in creating a "trapdoor" version of the knapsack problem.

A trapdoor in mathematics is secret information that once obtained makes it easy to solve a given problem, but without it, the problem is seemingly infeasible to solve [12].

Merkle and Hellman used this to design a cryptosystem where message decryption using only information in the public domain is equivalent to solving an NP-complete knapsack problem. However, decryption with the trap-door information (the private key) is equivalent to solving an easy knapsack problem.

Merkle and Hellman's 1978 paper [12] details the first known example of a trap-door public key system. They define a transformation relating:

- A knapsack problem $K(s, p)$, where s is the initial super-increasing sequence and p is the target variable used in a greedy method, and
- A knapsack problem $K(c, e, m)$, where c is the ciphertext sequence, e is the public key, and m is the modulus.

The use of modulus m is critical to the security of this cryptosystem, as it is used to obscure the values of the super-increasing sequence s during its transformation into the public key e: $e \equiv w \times s \pmod{m}$.

Merkle and Hellman designed this in such a way that the transformation $K(s,p) \rightarrow K(c,e,m)$ satisfies three properties [11]:

1. $K(s,p)$ and $K(c,e,m)$ are equivalent, or in other terms, have a common solution.
2. It's computationally infeasible (NP-complete) to solve the plaintext with only $K(c,e,m)$.
3. It is easy to solve for the plaintext with only $K(s,p)$.

A solution to $K(s,p)$ would be a $plaintext_i$ such that $p \equiv \sum_{i=0}^{n-1} s_i \times plaintext_i \pmod{m}$ is true. A solution to $K(e,c,m)$ would be a $plaintext_i$ such that $c \equiv \sum_{i=0}^{n-1} plaintext_i \times e_i \pmod{m}$ is true.

5.7 Historical impact and significance

The Merkle-Hellman Knapsack Cryptosystem holds a significant place in the history of cryptography, marking several important milestones and lessons in the field's development.

Proposed soon after the groundbreaking RSA algorithm, the Merkle-Hellman system was among the first workable public-key cryptosystems [11] and was instrumental in showcasing the feasibility and promise of public-key cryptography. It was first introduced when public-key encryption was still a relatively new idea and it contributed to the acceptance of the notion that secure communication could be accomplished without a pre-shared secret key.

The eventual break of the system by Adi Shamir in 1984 [13] was a turning point in cryptography. This event made clear how crucial it is to conduct thorough security analysis before implementing any new cryptographic schemes. It proved that unexpected vulnerabilities could exist in systems built on well-understood hard problems. This insight resulted in a more methodical and careful approach to the design and assessment of cryptosystems, thereby establishing a new benchmark for cryptographic research.

The Merkle-Hellman system was instrumental in generating interest in complexity-based cryptography [1]. By basing its security on the hardness of the knapsack problem, it opened up new avenues for cryptographic research beyond the traditional number-theoretic approaches. This creative cryptographic foundation has helped diversify our knowledge. It has played a crucial role in the ongoing search for secure and efficient encryption methods.

The Merkle-Hellman system serves as a cautionary tale in the field of cryptography. Its disappointingly short lifespan illustrates the challenges of translating theoretical security into practical, unbreakable systems [6]. This lesson has been invaluable in shaping the modern approach to cryptographic design, which emphasises not only theoretical security proofs but also resistance to a wide range of potential attacks and real-world considerations.

5.8 Proofs and explanation

Transformation

The proofs in this section were largely drawn from [11].

Proof 5.1: $K(s,p)$ and $K(c,e,m)$ have a common solution.

Suppose that $K(e,c,m)$ has a plaintext solution:

$$c \equiv \sum_{i=0}^{n-1} plaintext_i \times e_i \pmod{m}$$

If $p \equiv w^{-1} \times c \pmod{m}$, then there exists some integer j such that $w^{-1} \times c = p + jm$. This equation expresses the modular congruence in terms of standard equality.
It is true that $e_i \equiv w \times s_i \pmod{m}$, if and only if, $w^{-1} \times e_i \pmod{m} \equiv s_i$. In other words, these are equivalent statements, and one implies the other. Multiplying the ciphertext by w^{-1} produces:

$$jm + p = w^{-1} \times c = \sum_{i=0}^{n-1} w^{-1} \times e_i \times \mathit{plaintext}_i, 0 \leq p < m$$

Since $s_i \equiv w^{-1} \times e_i \pmod{m}$, then there exists some integer k_i such that $w^{-1} \times e_i = s_i + k_i m$. This equation expresses the modular congruence in terms of standard equality.

$$jm + p = \sum_{i=0}^{n-1} (s_i + k_i m) \times \text{plaintext}_i = \sum_{i=0}^{n-1} k_i m \times \text{plaintext}_i + \sum_{i=0}^{n-1} s_i \times \text{plaintext}_i$$

It is true that $jm \equiv 0 \pmod{m}$ and $k_i m \equiv 0 \pmod{m}$ because any multiple of m evenly divides into the modulus m, leaving no remainder. Knowing this, we can apply mod m to both sides of the equation, leaving us with:

$$p \equiv \sum_{i=0}^{n-1} s_i \times \mathit{plaintext}_i \pmod{m}$$

This equation is the definition of a solution in $K(s, p)$, meaning the same plaintext values satisfy the key equation for $K(s, p)$ using the secret key values s_i. Therefore, the plaintext is a solution to $K(s, p)$.

Conversely, suppose $K(s, p)$ has a plaintext solution:

$$p \equiv \sum_{i=0}^{n-1} s_i \times \text{plaintext}_i \pmod{m}$$

If $c \equiv w \times p \pmod{m}$, then there exists some integer, h, such that $w \times p = c + hm$. This equation expresses the modular congruence in terms of standard equality.
It is true that $s_i \equiv w^{-1} \times e_i \pmod{m}$ if and only if $w \times s_i \pmod{m} \equiv e_i$. In other words, these are equivalent statements. Multiplying the plaintext by w gives:

$$hm + c = w \times p = \sum_{i=0}^{n-1} w \times s_i \times \text{plaintext}_i, 0 \leq p < m$$

If $e_i \equiv w \times s_i \pmod{m}$, then there exists some integer l_i such that $w \times s_i = e_i + l_i m$. This equation expresses the modular congruence in terms of standard equality.

$$hm + c = \sum_{i=0}^{n-1} (e_i + l_i m) \times \text{plaintext}_i = \sum_{i=0}^{n-1} l_i m \times \text{plaintext}_i + \sum_{i=0}^{n-1} \text{plaintext}_i \times e_i$$

It is true that $hm \equiv 0 \pmod{m}$ and $l_i m \equiv 0 \pmod{m}$ because any multiple of m evenly divides into the modulus m, leaving no remainder. Knowing this, we can apply mod m to both sides of the equation, leaving us with:

$$c \equiv \sum_{i=0}^{n-1} \text{plaintext}_i \times e_i \pmod{m}$$

This equation is the definition of a solution in $K(e, c, m)$, meaning the same plaintext values satisfy the key equation using the public key values e_i. Therefore, the plaintext is a solution in $K(e, c, m)$.

Key generation

Private key

Explanation - Why does the private key need a super-increasing sequence, s?

As previously stated, a super-increasing knapsack sequence is easy and quick to solve. That is why we choose it for our private key and decryption. Remember the goal is to make it easy to decrypt with the trap door information.

It is quick for the greedy method to solve a super-increasing knapsack for a target variable as each element is larger than the sum of all previous elements, making the greedy choice always optimal.

Public key

Proof 5.2: The transformation to the public key does not preserve the super-increasing property of s

Given that $s = s_0, \ldots, s_{n-1}$ is a super-increasing sequence, w is coprime to m, and $m > \sum_{i=0}^{n-1} s_i$.

For any i and n where $i < n$:

- $e_i \equiv w \times s_i \pmod{m}$
- $e_n \equiv w \times s_n \pmod{m}$

The super-increasing property requires that $s_j > \sum_{i=0}^{j-1} s_i$ for each $j < n$.

Multiplying both sides by w:

$$w \times s_j > w \times \sum_{i=0}^{j-1} s_i$$

However, when taken modulo m, this inequality may not hold (the sequence is no longer ordered):

$$w \times s_j \pmod{m} \not\leq w \times \sum_{i=0}^{j-1} s_i \pmod{m}$$

This is because modular arithmetic can "wrap-around", which can disrupt the original inequality.

Therefore:

$$e_j \not\leq w \times \sum_{i=0}^{j-1} e_i \pmod{m}$$

Thus, the public key e may not preserve the super-increasing property of s, obscuring the structure of the original sequence.

Encryption and decryption

Definition 5.3: One-way functions

A one-way function is a function which is easy to compute in one direction, but hard to compute in the other direction.

Encryption

Proof 5.3: Without the private key, the encryption process is a one-way function

An attacker has the ciphertext c and the public key $e_0, e_1, \ldots, e_{n-1}$. They know that the ciphertext c is computed as:

$$c = \sum_{i=0}^{n-1} plaintext_i \times e_i \equiv \sum_{i=0}^{n-1} plaintext_i \times w \times s_i \pmod{m}$$

The attacker needs to find the bits in $plaintext_i$. To do this, they try to find a subset of the numbers in the public key that sum to the ciphertext. This is the subset sum problem, which as mentioned before, is NP-complete.

Therefore, the encryption process is a one-way function without the private key.

Decryption

The computation $p \equiv c \times w^{-1} \pmod{m}$ undoes the multiplication by w in the public key. The reveals the target number that the subset of the original super-increasing sequence must total to.

Proof 5.4: The decryption process will correctly recover the original plaintext

Given the ciphertext, c, and the private key, (m, w^{-1}, s)

$$p \equiv w^{-1} \times c \pmod{m} \equiv w^{-1} \times (\sum_{i=0}^{n-1} plaintext_i \times e_i) \equiv w^{-1} \times (\sum_{i=0}^{n-1} plaintext_i \times w \times s_i) \pmod{m}$$

$$\equiv 1 \times \sum_{i=0}^{n-1} plaintext_i \times s_i \pmod{m} \equiv \sum_{i=0}^{n-1} plaintext_i \times s_i \pmod{m}$$

Given the target number, p, and the super-increasing sequence, s, we can uniquely determine the plaintext (as explained in the first theorem about super-increasing sequences).

Proof 5.5: An element selected by a greedy method is the optimal choice

Suppose that an element of s, s_i, is selected with a greedy method, implying that $p \geq s_i$. If s_i was not the optimal solution, then the optimal solution would only contain elements smaller than s_i. However, by definition of a super-increasing sequence, the sum of all elements smaller than s_i is less than s_i. This means that if s_i is selected, it must be the optimal choice at this step.

Proof 5.6: An element not selected by a greedy method is not the optimal choice

Suppose that an element of s, s_i, was not selected by a greedy algorithm, implying that $p < s_i$. Any subset that includes s_i would sum to more than p. Therefore, s_i cannot be the optimal solution if it is not selected by a greedy method.

Proof 5.7: A greedy method will always find the plaintext

The super-increasing property guarantees that choosing the largest number does not 'block' us from using other numbers to reach the target sum. Specifically, the remaining sum $s = s - s_i$ is less than s_i, as each element of a super-increasing sequence must be larger than the sum of all prior elements. $s_j > \sum_{i=0}^{j-1} s_i$

This property ensures that smaller elements can still contribute in a greedy method.

In a super-increasing sequence, if there exists a subset that sums to s, a greedy method will find it. The reason is that if we don't choose the largest number that fits, we would leave a larger remaining sum than what can be achieved by the smaller numbers.

5.9 Conclusions

Public-key cryptography has advanced significantly since the invention of the Merkle-Hellman Knapsack algorithm. This algorithm showed that public-key systems could be built using a variety of mathematical foundations, not just the factorisation problem used in RSA, by basing its security on the knapsack problem.

While the algorithm itself is no longer considered secure for practical use, its study provides valuable insights into the principles of public-key cryptography, the use of knapsack problems in cryptography, and the importance of rigorous security analysis for cryptographic systems.

We can learn several important lessons from the Merkle-Hellman algorithm:

1. The importance of the trapdoor concept in public-key cryptography.
2. The potential of using computationally hard problems as the basis for cryptographic systems.
3. The necessity of careful cryptanalysis and the dangers of relying solely on the perceived difficulty of a mathematical problem for security.

The insights gained from the Merkle-Hellman Knapsack Algorithm are still applicable today as we encounter new information security challenges. It reminds us of the creative problem-solving needed in cryptography as well as the continuous requirement for thorough examination and testing of cryptographic systems.

Context

This article was adapted from a report submitted in SIT281 - Cryptography.

About the Author

Caleb is a bright and bubbly student with a passion for cybersecurity and computing. He demonstrates an eager attitude when it comes to learning new material, which has allowed him to excel in his studies. In his free time, Caleb enjoys going to the gym, playing sports, and researching whatever topic has recently grabbed his attention.

Acknowledgements

I would like to acknowledge Dr Guillermo Pineda-Villavicencio, who provided me the opportunity to publish my paper. I would also like to acknowledge my friends and family, who allowed me to prioritise my academic passions and have encouraged me on my learning journey.

References

[1] B. Barak. "The Complexity of Public-Key Cryptography". In *Tutorials on the Foundations of Cryptography*, pages 45–77. Springer International Publishing, Cham, 2017.

[2] Abdul Bari. "3.1 Knapsack problem - greedy method". *YouTube*. Feb. 06, 2018. [Online]. Available: https://www.youtube.com/watch?v=oTTzNMHM05I.

[3] CryptoWiki Contributors. "Merkle–Hellman knapsack cryptosystem". *Crypto Wiki*. [Online]. Available: https://cryptography.fandom.com/wiki/Merkle%E2%80%93Hellman_knapsack_cryptosystem.

[4] Wikipedia Contributors. "NP-completeness". *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/NP-completeness.

[5] Wikipedia Contributors. "NP (complexity)". *Wikipedia*. [Online]. Available: https://en.wikipedia.org/wiki/NP_(complexity).

[6] Study Force. "Merkle-Hellman Knapsack Algorithm Part 1: Generating the Public Key cryptography algorithm". *YouTube*. Aug. 04, 2024. [Online]. Available: https://www.youtube.com/watch?v=6SuhCPy8oKE.

[7] Study Force. "Merkle-Hellman Knapsack Algorithm Part 2: Encryption and Decryption cryptography algorithm". *YouTube*. Aug. 04, 2024. [Online]. Available: https://www.youtube.com/watch?v=UsryH-AK5S0.

[8] hackerdashery. "P vs. NP and the Computational Complexity Zoo". *YouTube*. Aug. 26, 2014. [Online]. Available: https://www.youtube.com/watch?v=YX40hbAHx3s.

[9] B. Kaliski. "Polynomial time". In *Encyclopedia of Cryptography and Security*, pages 948–949. Springer US, Boston, MA, 2011.

[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, Berlin, Heidelberg, 2004.

[11] A. G. Konheim. *"Computer Security and Cryptography"*. Wiley-Interscience, 2007.

[12] R. Merkle and M. Hellman. "Hiding information and signatures in trapdoor knapsacks". *IEEE Transactions on Information Theory*, 24(5):525–530, September 1978.

[13] A. Shamir. "A polynomial-time algorithm for breaking the basic Merkle-Hellman cryptosystem". *IEEE Transactions on Information Theory*, 30(5):699–704, September 1984.

[14] Sudiptadandapat. "Knapsack encryption algorithm in cryptography". *GeeksforGeeks*. Jun. 19, 2024. [Online]. Available: https://www.geeksforgeeks.org/knapsack-encryption-algorithm-in-cryptography/.

[15] D. Terr. "Polynomial Time". *MathWorld*. [Online]. Available https://mathworld.wolfram.com/PolynomialTime.html.

[16] Computational Thinking. "NP-Hardness". *YouTube*. Oct. 04, 2022. [Online]. Available: https://www.youtube.com/watch?v=KCFCcRmJAU8.

6

Complex numbers and complex matrices in linear algebra

Visal Dam and Amanda Roberts

Abstract

The study of matrices in linear algebra traditionally focuses on real numbers. However, extending these concepts to the domain of complex numbers opens up a richer framework for analyzing and solving multidimensional problems. This article explores the interplay between complex numbers and matrices, presenting their properties, operations, and key applications. Beginning with an introduction to complex numbers and their geometric representation, the article delves into the behavior of complex matrices, including eigenvalues, eigenvectors, and special types such as Hermitian and unitary matrices. The article also examines the use of complex numbers in solving systems of linear equations and matrix decomposition techniques like Schur decomposition and the Spectral Theorem. These tools are applied to real-world scenarios in quantum computing and data analysis, highlighting their significance in advanced fields like signal processing, machine learning, and quantum mechanics. By unifying abstract mathematical principles with practical applications, this work underscores the indispensable role of complex matrices in modern science and engineering.

Introduction

A cornerstone of modern mathematics and its applications, linear algebra is the study of vectors and linear functions [5]. Traditional linear algebra focuses on real numbers; extending these concepts to the complex number system unveils a richer structure and a broader spectrum of applications, including science and engineering. Matrices are important in linear algebra, and can contain both real and/or complex numbers. The behaviours of real and complex numbers differ, and complex matrices exhibit different behaviours to real matrices. This article explores five key areas where complex numbers and matrices extend the power of linear algebra:

1. **The properties of complex numbers**: The article begins by examining the fundamental properties of complex numbers and their geometric representation on the complex plane. These numbers, expressed as a combination of real and imaginary parts, are crucial in handling multidimensional phenomena in quantum computing and data analysis. Their arithmetic operations—addition, subtraction, multiplication, and division—provide the foundation for complex transformations and signal processing tasks that involve both magnitude and phase relationships in these advanced fields.

2. **The importance of eigenvalues and eigenvectors**: This section investigates how complex eigenvalues and eigenvectors provide deeper insights into the behavior of linear

transformations. In quantum computing, complex eigenvalues correspond to oscillatory behavior, essential for understanding wave functions and quantum state evolution.

3. **The uniqueness of the behaviors of complex matrices and their applications in matrix theory**: This section explores the roles and properties of complex matrices by exploring the conjugate transpose, complex determinants, and the standard inner product. Each of these plays a pivotal role in simplifying matrix operations and ensuring stability in linear systems. This section also discusses how these tools are used to solve problems in quantum systems and data analysis, where matrices often have complex entries.
4. **The use of complex numbers in solving systems of linear equations**: This section examines how complex numbers are incorporated into systems of linear equations, providing methods for solving problems where both coefficients and unknowns have real and imaginary components. These techniques are crucial for algorithms in data processing and quantum information systems.
5. **The importance of Hermitian and unitary matrices and their roles in decomposing complex matrices**: Hermitian and unitary matrices are fundamental in complex linear algebra, especially in quantum computing. Hermitian matrices, with real eigenvalues, represent observable physical quantities, while unitary matrices preserve norms, ensuring the stability of quantum states during transformations. Both types of matrices play a key role in decomposing complex matrices into simpler forms that are more managable.

After presenting the properties of complex numbers, basic matrix operations, and fundamental theorems—this article seeks to delve deeper into the intricate relationship between complex numbers and linear algebra, highlighting their unified power in modern mathematical applications including matrices, and their application to the real world through quantum computing and data analysis.

6.1 Complex Numbers

A Brief History of Numbers

What is a number? In its most fundamental sense, a number is an abstract concept used to represent quantity and order. In the ancient world, numbers were restricted to only **natural** numbers, $\mathbb{N} = \{1, 2, 3, ...\}$. These numbers occurred in nature and could support daily life: one apple, four trees, or 52 silver ingots. This concept is often introduced via the number line, as shown in Figure 6.1.

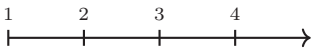


Figure 6.1: Number line representing the set of natural numbers $\mathbb{N}$

The existence of only natural numbers was eventually challenged by the concept of zero. The introduction of negative numbers in mathematics followed, based on their usefulness and importance in solving quadratics. To reflect these new values, a new set of numbers named **integers**, $\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$, was developed. The number line representing the set of natural numbers was then extended to reflect the integers, as shown in Figure 6.2.

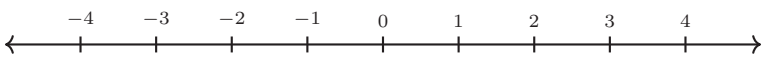


Figure 6.2: Number line representing the set of integers $\mathbb{Z}$

There exists a relationship between the two sets; $\mathbb{N}$ is a subset of $\mathbb{Z}$, represented as $\mathbb{N} \subset \mathbb{Z}$. Numbers can now be grouped and categorized based on their characteristics.

Extending this beyond whole numbers, there are also **rationals**. Rationals are the set of numbers that can be expressed in the form $\frac{a}{b}$ where $a, b \in \mathbb{Z}$. Denoted $\mathbb{Q}$, rationals contain decimals and fractions, as well as all integers $\mathbb{Z}$, where $\mathbb{Z}$ is a subset of $\mathbb{Q}$ ($\mathbb{Z} \subset \mathbb{Q}$). The set of **real** numbers, denoted $\mathbb{R}$, includes $\mathbb{Q}$ and irrational numbers such as π and e. The next largest set is **complex** numbers, denoted $\mathbb{C}$, and for which all others are subsets.

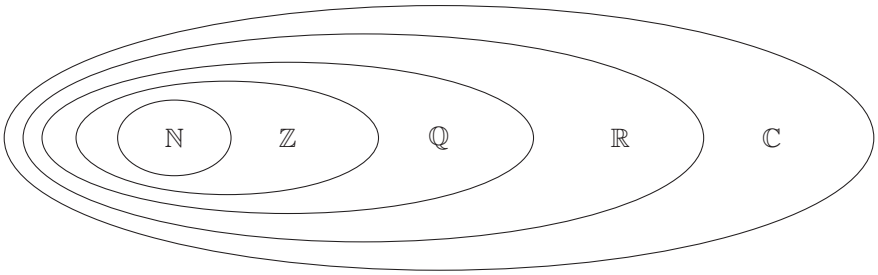


Figure 6.3: Hierarchy of number sets; $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$

The Complex Plane

Complex numbers, like negative numbers, were born out of necessity due to the lack of solutions to quadratics. In the past, quadratics were previously used to define the area of fields or houses, with the root x representing the length. A negative root would have indicated a negative length, which did not make sense and was thus deemed absurd.

As the use of quadratics was broadened, so was the definition of the values. There are three common methods to solve quadratics, one of which makes use of the quadratic formula, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, where $b^2 - 4ac$ is known as the *discriminant*, which determines the nature of x:

1. If $b^2 - 4ac > 0$, then there are two real roots: $x_1, x_2 \in \mathbb{R}$ and $x_1 \neq x_2$.
2. If $b^2 - 4ac = 0$, then there is one real root: $x_1 = x_2 = x \in \mathbb{R}$.
3. If $b^2 - 4ac < 0$, then there are no real roots. This is because the square root evaluates to the form $\sqrt{-k}$ where k is some number. Instead, the roots are complex.

Using the example $x^2 + 1 = 0$, there are two possible solutions:

$$x^2 = -1$$

$$x = \pm\sqrt{-1}$$

These solutions are not possible if restricted to only real numbers. The value $\sqrt{-1}$ is represented as i and known as the **imaginary unit**. This is a building block of complex numbers, much like how 1 is the building block of integers. The visual representation of numbers is then extended from the 1-dimensional number line in Figure 6.2 to the 2-dimensional plane in Figure 6.4.

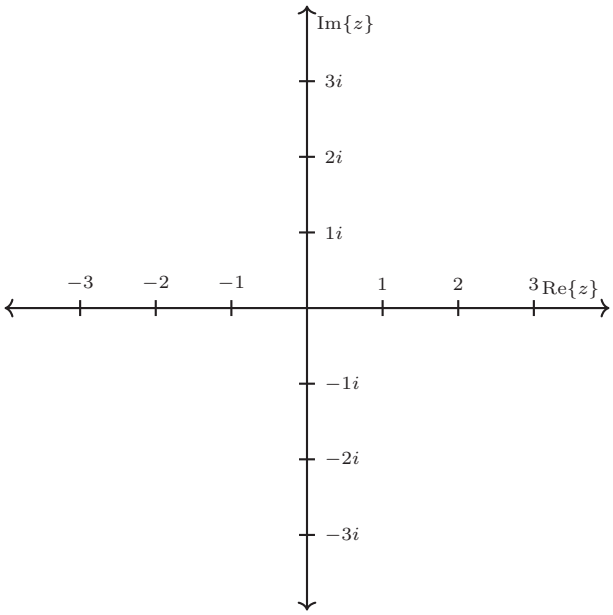


Figure 6.4: The Complex Plane (adapted from [3])

Complex numbers can then be represented geometrically on the complex plane, also known as the **Argand plane**. In this plane, the horizontal x-axis represents the real part of the complex number, and the vertical y-axis represents the imaginary part of the complex number. The complex number $z = a + bi$ is represented as the point (a, b) on the plane or as a vector from the origin to the point (a, b). The real part a determines the horizontal position, and the imaginary part b determines the vertical position.

Properties of Complex Numbers

It is understood that i is not the *only* building block of complex numbers. This is because complex numbers, denoted by z, consist of two parts: the real part, a (or $\text{Re}(z)$), and the imaginary part, bi (or $\text{Im}(z)$). It is important to note that a and b are both real numbers, but by multiplying by i, bi exists along the imaginary axis of the complex plane.

Definition 6.1: Complex Number

A complex number $z \in \mathbb{C}$ is of the form

$$z = a + bi$$

where $i = \sqrt{-1}$ and $a, b \in \mathbb{R}$.

The **Argand diagram** is used to visualize or plot complex numbers, as shown in Figure 6.5. This reflects the hierarchy of sets represented in Figure 6.3, where $\mathbb{R}$ is a *specific subset* of $\mathbb{C}$ wherein every element can be expressed as $r + 0i$ and consisting of just the real part.

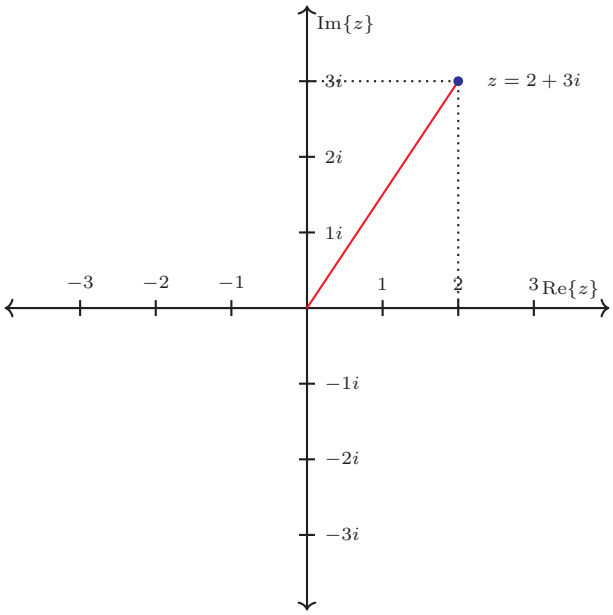


Figure 6.5: Argand diagram showing the complex number $z = 2 + 3i$; $a = 2, b = 3$, $i, z \in \mathbb{C}$ (adapted from [3])

Elementary Operations

Complex numbers can be added, subtracted, multiplied, and divided like real numbers. The operations must account for the imaginary unit i, making some elementary operations in $\mathbb{C}$ rather unique. These operations are fundamental in both quantum computing and data analysis, where the manipulation of complex numbers plays a significant role in analyzing and solving problems.

Addition: Consider two complex numbers $z = a + bi$ and $w = c + di$. This can be simplified to $z + w = (a + c) + (b + d)i$. The real and imaginary parts interact with their own kind separately.

Example 6.1

Let $z, w \in \mathbb{C}$ where $z = 2 + 3i$ and $w = 4 - 9i$. Then,

$$\begin{aligned} z + w &= (2 + 3i) + (4 - 9i) \\ &= (2 + 4) + (3i + -9i) \\ &= 6 - 6i \end{aligned}$$

Let $z, w \in \mathbb{C}$ where $z = -2 + 3i$ and $w = 18i$. Then,

$$\begin{aligned} z + w &= (-2 + 3i) + (0 + 18i) \\ &= (-2 + 0) + (3i + 18i) \\ &= -2 + 21i \end{aligned}$$

Note that subtraction acts in the same way as addition.

Multiplication: Consider the scalar $n \in \mathbb{R}$. Then, $nz = n(a + bi) = n(a) + n(b)i$.

When multiplying complex numbers, apply the distributive property and simplify using $i^2 = -1$. For complex numbers $z = a + bi$ and $w = c + di$,

$$zw = (a + bi)(c + di) = ac + adi + cbi + bdi^2,$$

and as $i = \sqrt{-1}$, $i^2 = -1$,

$$\begin{aligned} zw &= ac + adi + cbi - bd \\ &= ac - bd + adi + cbi \end{aligned}$$

Note that the sum $z + w$ and product zw have both a real part and an imaginary part.

For complex numbers z and w such that $zw = 1$, w is the inverse of z. Therefore, $w = z^{-1}$ and

$$z^{-1} = \frac{a - bi}{a^2 + b^2}$$

where $a - bi$ is known as the **conjugate** of z, denoted by $\overline{z}$. The complex conjugate is formed by negating the imaginary part of the complex number, keeping the real part unchanged. This means that if $z = a + bi$, then $\overline{z} = a - bi$ as conjugation only affects the imaginary part of the equation.

Theorem 6.1: Complex Inverses

If $z \in \mathbb{C}$, then its inverse z^{-1} is such that $zz^{-1} = z^{-1}z = 1$ and

$$z^{-1} = \frac{a - bi}{a^2 + b^2}$$

On the complex plane, the complex conjugate reflects the point representing the complex number across the real axis. If z is above the real axis (positive imaginary part), $\overline{z}$ will be below it (negative imaginary part), and vice versa.

The product of a complex number z and its conjugate $\overline{z}$ results in a real number (see Proof 6.1):

$$z\overline{z} = (a + bi)(a - bi) = a^2 + b^2$$

This is helpful when dividing complex numbers, as multiplying by the conjugate removes the imaginary part from the denominator.

The modulus of a complex number $z = a + bi$, denoted by $|z|$, is the distance from the origin $(0, 0)$ to the point (a, b) on the complex plane and is related to its conjugate via:

$$|z| = \sqrt{z\overline{z}} = \sqrt{a^2 + b^2}$$

Proof of Theorem 6.1

Let $z \in \mathbb{C}$ where $z = a + bi$ and its conjugate $\overline{z} = a - bi$. Then,

$$\begin{aligned} z\overline{z} = (a + bi)(a - bi) &= a^2 - abi + abi - b^2i^2 \\ &= a^2 + 0 - b^2(-1) \\ &= a^2 + b^2 \end{aligned}$$

In Figure 6.5, the right-triangle formed where the base (adjacent) is a and the height (opposite) is b. Using Pythagoras' theorem, $c = \sqrt{a^2 + b^2}$, c can be determined as the magnitude or length of z. This is denoted as $|z|$, and $|z|^2 = a^2 + b^2$. Therefore,

$$\frac{z\overline{z}}{|z|^2} = \frac{a^2 + b^2}{a^2 + b^2} = 1$$

$$z * \frac{\overline{z}}{|z|^2} = z * \frac{a - bi}{a^2 + b^2} = 1$$

and

$$\frac{1}{z} = z^{-1} = \frac{a - bi}{a^2 + b^2}$$

Example 6.2

Let $z \in \mathbb{C}$ where $z = -8 + i$. Find and verify z^{-1}.
To find,

$$z^{-1} = \frac{a - bi}{a^2 + b^2}$$

$$= \frac{-8 - i}{(-8)^2 + 1^2}$$

$$= \frac{1}{65} * (-8 - i)$$

$$z^{-1} = \frac{-8}{65} - \frac{1}{65}i$$

To verify,

$$zz^{-1} = (-8 + i)(\frac{-8}{65} - \frac{1}{65}i)$$

$$= (-8 * \frac{-8}{65} - 1 * \frac{-1}{65}) + (-8 * \frac{-1}{65} + 1 * \frac{-8}{65})i$$

$$= (\frac{64}{65} - \frac{-1}{65}) + (\frac{8}{65} + \frac{-8}{65})i$$

$$= 1 + 0i$$

Therefore,

$$zz^{-1} = 1$$

Division of complex numbers requires multiplying the numerator and denominator by the complex conjugate of the denominator to eliminate the imaginary part. For complex numbers $z = a + bi$ and $w = c + di$,

$$\frac{z}{w} = \frac{a + bi}{c + di} = \frac{(a + bi)(c - di)}{(c + di)(c - di)} = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2}$$

Popularized by Gauss [1], the term and use of complex numbers has prevailed. Complex numbers are crucial in extending the number system beyond real numbers, allowing for the representation of two-dimensional quantities, such as amplitude and phase, in one unified framework. As previously demonstrated, the use of complex numbers captures, in essence, rotation, and thus they are commonly used in many mathematical and engineering disciplines [2], such as, most notably, in quantum physics via Schrödinger's wave equation [4]. In data analysis, complex numbers are

central to signal processing tasks such as the discrete Fourier transform, as well as principal component analysis (PCA) when working with multidimensional datasets that involve periodic or oscillatory data. Complex numbers are also critical in certain machine learning algorithms, where they are employed to enhance the algorithm's ability to capture patters in the data.

6.2 Complex Matrices

A complex matrix is an $n \times m$ matrix A where every entry a_{ij} is a complex number. This means that all the real matrices are technically complex matrices, as $\mathbb{R} \subset \mathbb{C}$ where the imaginary part is just 0.

The key difference is that real matrices do not exhibit the same properties as complex matrices. However, the properties of complex matrices can often be applied to real matrices.

Importance of Eigenvalues and Eigenvectors

Eigenvalues and **eigenvectors** are critical concepts in linear algebra. When extended to the complex domain, they provide a powerful framework for understanding linear transformations such as rotations, scaling, and reflections in vector spaces. Further, eigenvalues and eigenvectors have significant applications in data analysis and quantum computing.

Eigenvalues are the factors by which a transformation stretches or compresses vectors, while eigenvectors are the directions that remain unchanged (except for scaling) during the transformation. When complex numbers are involved, these transformations often reveal oscillatory or rotational behavior.

Recall that an eigenvalue λ and its corresponding eigenvector v of a square matrix A satisfy the equation:

$$Av = \lambda v$$

Here, matrix A is an $n \times n$ matrix, v is a non-zero vector in $\mathbb{C}^n$ (the space of n-dimensional complex vectors), and λ is a scalar (which may be a complex number). In the context of complex numbers, even if the entries of matrix A are real, the eigenvalues can be complex numbers and the eigenvectors can also have complex components. A matrix can be diagonalized with complex eigenvectors. Matrices with complex eigenvalues correspond to transformations involving rotations and scaling in the plane.

In linear transformations, complex eigenvalues often indicate ***rotational*** behavior in transformations. For example, in two-dimensional real spaces, a matrix with complex eigenvalues represents a rotation combined with scaling. In differential equations and systems dynamics, complex eigenvalues correspond to oscillations.

In higher dimensions, eigenvectors represent directions in which a linear transformation (represented by a matrix) acts as scaling. For real eigenvalues, these directions are straightforward as the transformation *stretches* or *compresses* vectors along the eigenvector direction by the eigenvalue's magnitude. However, for complex eigenvalues, the interpretation involves *both* scaling and rotation.

For a 2×2 matrix with complex eigenvalues, the transformation represents a rotation in the plane, in addition to any scaling. For the matrix A representing a 90-degree rotation

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

the eigenvalues are $\lambda = i, -i$, which are imaginary. The corresponding eigenvectors are complex, and reflect the fact that the transformation involves circular motion.

In three or more dimensions, the interpretation of complex eigenvalues generalizes to rotation in multiple planes. Complex eigenvalues indicate that the transformation does not simply stretch

or compress vectors, but also rotates them in the multidimensional space. For a matrix A representing a 3-dimensional rotation

$$A = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

the matrix rotates vectors in the xy-plane while leaving the z-axis unchanged. The eigenvalues are $\lambda_1 = 1$, $\lambda_2 = i$, and $\lambda_3 = -i$. The complex eigenvalues correspond to rotation in the xy-plane, while the real eigenvalue $\lambda_1 = 1$ corresponds with the z-axis, where values will be unchanged (scaled by 1). Thus, the complex eigenvalues reflect a circular rotation within the plane, while the real eigenvalue reflects scaling along the remaining axis.

In quantum computing, complex eigenvalues are fundamental for representing the behavior of quantum states. The Schrödinger equation, which governs the evolution of quantum systems, often involves matrices or operators with complex eigenvalues and eigenvectors. In data analysis, eigenvalue decomposition is often used in principal component analysis (PCA) to reduce the dimensionality of datasets. Complex eigenvalues can arise in situations where the data involves cyclic patterns or wave-like behaviors, as in the analysis of signals or financial time series.

The Standard Inner Product

The **standard inner product** is to complex matrices what the dot product is to real matrices.

Let complex vectors $\mathbf{z} = (z_1, z_2, ..., z_n)$ and $\mathbf{w} = (w_1, w_2, ..., w_n)$ where each $z_k, w_k \in \mathbb{C}$. The standard inner product is

$$\langle \mathbf{z}, \mathbf{w} \rangle = z_1\overline{w}_1 + z_2\overline{w}_1 + ... + z_n\overline{w}_n = \mathbf{z} \cdot \overline{\mathbf{w}}$$

where $\overline{w}_k$ is the conjugate of w_k and $\cdot$ denotes the dot product.

Imagine that the elements in $\mathbf{z}$ and $\mathbf{w}$ were real numbers instead of complex ones. Recalling that conjugation only affects the imaginary part, then each w_k would remain unaffected, and the standard inner product would be the same as the dot product.

Definition 6.2: Properties of the Standard Inner Product

Let $\mathbf{z}, \mathbf{z}_1, \mathbf{w}$, and $\mathbf{w}_1$ be vectors in $\mathbb{C}^n$ and let λ be a complex number $\in \mathbb{C}$. Then,

1. $\langle \mathbf{z}, \mathbf{w} \rangle = \mathbf{z}^T\overline{\mathbf{w}}$
2. $\langle \mathbf{z} + \mathbf{z}_1, \mathbf{w} \rangle = \langle \mathbf{z}, \mathbf{w} \rangle + \langle \mathbf{z}_1, \mathbf{w} \rangle$
3. $\langle \lambda\mathbf{z}, \mathbf{w} \rangle = \lambda\langle \mathbf{z}, \mathbf{w} \rangle$ but when $\langle \mathbf{z}, \lambda\mathbf{w} \rangle = \overline{\lambda}\langle \mathbf{z}, \mathbf{w} \rangle$
4. $\langle \mathbf{z}, \mathbf{w} \rangle = \overline{\langle \mathbf{w}, \mathbf{z} \rangle}$ i.e., $= \langle \overline{\mathbf{w}}, \overline{\mathbf{z}} \rangle$
5. $\langle \mathbf{z}, \mathbf{z} \rangle \geq 0$ and $\langle \mathbf{z}, \mathbf{z} \rangle = 0$ if and only if $\mathbf{z} = \mathbf{0}$

The length of the complex number z is $|z|$ (see Proof 6.1), and the length of the complex vector $\mathbf{z} = (z_1, z_2, ..., z_n)$ in $\mathbb{C}^n$ is:

$$||\mathbf{z}|| = \sqrt{|z_1|^2 + |z_2|^2 + ...|z_n|^2} = \sqrt{\langle \mathbf{z}, \mathbf{z} \rangle}$$

This gives rise to orthogonality in $\mathbb{C}^n$. If $\langle \mathbf{z}_i, \mathbf{z}_j \rangle = 0$ and $i \neq j$, then the set of non-zero vectors $\{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z_n}\}$ is orthogonal. Further, if each $||\mathbf{z}_i|| = 1$, then it is also orthonormal.

Matrix Conjugates and Complex Determinants

In matrix theory, the complex conjugate and determinant play critical roles in solving problems involving complex matrices. These concepts are pivotal in understanding and simplifying matrix operations, particularly in advanced fields like quantum computing and data analysis, where matrices with complex entries often arise.

The complex conjugate of a matrix is an operation applied to each element individually. For a matrix A with complex entries, the complex conjugate, denoted as $\overline{A}$, is formed by replacing each element a_{ij} of matrix A with its complex conjugate, $\overline{a}_{ij}$. This operation is essential in simplifying matrix calculations, particularly when working with complex-valued inner products, decompositions, and eigenvalue problems.

If matrix A is

$$A = \begin{bmatrix} 1+i & 2 \\ -3i & 4i \end{bmatrix}$$

then the complex conjugate, matrix $\overline{A}$, is

$$\overline{A} = \begin{bmatrix} 1-i & 2 \\ 3i & -4i \end{bmatrix}$$

The determinant of a matrix with complex entries is calculated in the same way as for real matrices. The determinant of a matrix indicates how much a transformation (represented by the matrix) scales the volume of a geometric shape. In quantum computing and data analysis, the determinant plays a crucial role in determining whether a matrix is invertible and in the analysis of linear transformations.

For any square matrix A, the matrix is invertible if and only if its determinant is non-zero: $\det(A) \neq 0$. The determinant is also a useful tool in analyzing how matrices affect space. For example, if the determinant is positive, the transformation preserves the orientation of vectors, whereas if it is negative, the transformation reverses orientation. This concept is important in data analysis, where transformations of data points are applied in methods like principal component analysis (PCA) and signal processing.

The determinant of a matrix with complex entries is computed in the same way as for real matrices and can be complex in nature, for example $\det(A) = a + bi$. A matrix is invertible if and only if its determinant is non-zero, and for real values the determinant represents how much a transformation (represented by the matrix) scales the volume of geometric shapes. For two matrices A and B, $\det(AB) = \det(A) \times \det(B)$.

The properties of the complex conjugate and determinant are especially useful in simplifying matrix calculations involving complex numbers. In data analysis and machine learning, matrices with complex entries often arise in fields such as signal processing, image analysis, and computer vision. The complex conjugate and determinant are used to optimize algorithms, especially in the areas of feature extraction, filtering, and dimensionality reduction. Additionally, in algorithms such as Shor's algorithm (for factoring large numbers) and Grover's search algorithm (for database search), matrix operations involving complex numbers are used to encode and manipulate quantum information.

The Conjugate Transpose

In complex matrices, transposition works similarly to real matrices, except that every complex entry must also be conjugated. In $\mathbb{C}^n$, the **conjugate transpose** of a complex matrix A is denoted by A^* and defined as

$$A^* = (\overline{A})^T = \overline{(A^T)}$$

where the transpose of A is used and every complex entry a_{ij} is transposed. When A is a real matrix, A^* is just A^T as the conjugation only affects the "imaginary" parts.

The conjugate transpose is also known as the **Hermitian transpose** after the mathematician Charles Hermite [5]. The term conjugate transpose will be used to avoid confusion with *Hermitian matrices* in later sections.

For the matrix A,

$$A = \begin{bmatrix} 1+i & 2 \\ -3i & 4i \end{bmatrix}$$

the conjugate transpose A^* is

$$A^* = \begin{bmatrix} 1-i & 3i \\ 2 & -4i \end{bmatrix}$$

The determinant of the conjugate transpose of a matrix, $\det(A^*)$, is the complex conjugate of the determinant of the original matrix, $\det(A)$:

$$\det(A^*) = \overline{\det(A)}$$

as complex matrix determinants can be complex in nature.

Definition 6.3: Properties of the Conjugate Transpose

Let A and B be complex matrices in $\mathbb{C}^n$, and λ be a complex number. Then,

1. $(A^*)^* = A$
2. $(A+B)^* = A^* + B^*$
3. $(\lambda A)^* = \overline{\lambda} A^*$
4. $(AB)^* = B^* A^*$

Example 6.3: Conjugate Transpose in $\mathbb{C}^n$

For the matrix

$$A = \begin{bmatrix} 2+i & 4 & 4-3i \\ i & -5-7i & -8+9i \\ 10i & -9+i & -2i \end{bmatrix}$$

the conjugate tranpose is given by

$$A^* = \begin{bmatrix} 2-i & -i & -10i \\ 4 & -5+7i & -9-i \\ 4+3i & -8-9i & 2i \end{bmatrix}$$

Both the complex conjugate and conjugate tranpose play vital roles in matrix decompositions. The conjugate transpose is particularly important in quantum mechanics and quantum computing, where it helps preserve key properties of quantum states and transformations.

Complex Numbers in Linear Systems

When solving **systems of linear equations with complex coefficients**, the techniques are similar to those for real-number systems, but they require careful management of the arithmetic of complex numbers. Common methods include Gaussian elimination, matrix inversion, and

substitution. These systems typically involve solutions where both the coefficients and the unknowns are complex numbers, leading to more nuanced calculations.

Consider a system of n linear equations in n unknowns, which can be written in matrix form as $A\mathbf{x} = \mathbf{b}$, where $A \in \mathbb{C}^n$ is the matrix of coefficients with complex entries, $\mathbf{x} \in {}^n$ is the vector of unknowns, and $\mathbf{b} \in \mathbb{C}^n$ is the vector of constants. Both A and $\mathbf{b}$ may contain complex numbers, and the goal is to find the vector $\mathbf{x}$, which will also consist of complex values in general. The solution to the system involves both real and imaginary components, reflecting the structure of the underlying problem.

Gaussian elimination is a method used to solve systems of linear equations by reducing the matrix A to row-echelon form (or reduced-row-echelon form) using a sequence of elementary row operations. These operations include swapping rows (to move a non-zero pivot element to the diagonal), multiplying rows by non-zero scalars (to normalize pivot elements), and adding multiples of rows to other rows (to eliminate entries below the pivot).

For complex systems, the arithmetic of complex numbers must be handled when performing these row operations. This includes multiplying and dividing complex numbers, which requires the use of conjugates in some cases.

Consider the following system of two equations with complex coefficients:

$$(1+i)x_1 + 2x_2 = 3 + 4i$$
$$3x_1 + (2-i)x_2 = 1 - i$$

This system can be written in matrix form as:

$$\begin{bmatrix} 1+i & 2 \\ 3 & 2-i \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3+4i \\ 1-i \end{bmatrix}$$

To apply Gaussian elimination, normalize the first pivot by dividing the first row by $1 + i$. Then, eliminate the first element in the second row by subtracting 3 times the first row from the second row, and finally, back-substitute after obtaining row-echelon form to find the values of x_1 and x_2. This process is similar to Gaussian elimination with real numbers, but it requires managing complex arithmetic at each step.

If the system can be written as $A\mathbf{x} = \mathbf{b}$, where A is an invertible matrix, the solution can be found using matrix inversion:

$$\mathbf{x} = A^{-1}\mathbf{b}$$

To compute the inverse of a matrix $A \in \mathbb{C}^n$, apply the formula:

$$A^{-1} = \frac{1}{\det(A)}\text{adj}(A)$$

where $\det(A)$ is the determinant of A, which may be complex, and $\text{adj}(A)$ is the adjugate matrix of A, which also involves complex arithmetic.

A matrix is invertible if and only if $\det(A) \neq 0$. For complex matrices, calculating the determinant and adjugate involves more steps, especially when complex conjugates are needed for certain elements.

In quantum computing, complex systems of linear equations arise naturally when describing quantum states and transformations. Techniques such as Gaussian elimination and matrix inversion are essential in simulating quantum algorithms, where matrix operations involving complex numbers describe the dynamics of quantum bits (qubits). Additionally, in data analysis, solving systems of linear equations with complex coefficients is crucial in various applications, such as signal processing. Complex numbers are used to represent signals in the frequency domain, where solving systems of linear equations allows us to extract key features such as amplitude and phase.

6.3 Special Complex Matrices

Hermitian and **unitary** matrices hold special significance in linear algebra, especially when working with complex numbers. These matrices are integral in ensuring that transformations preserve the geometric structure of vector spaces, such as orthogonality and vector norms.

Hermitian Matrices

A square complex matrix A is called **Hermitian** (or said to be self-adjoint) if its original structure and conjugate transpose are equal, that is, if

$$A = A^*$$

or if its conjugated form is equal to its tranpose, that is, if

$$\overline{A} = A^T.$$

This makes Hermitian matrices the complex equivalent of real symmetric matrices.

This in turn implies that the diagonal elements of a Hermitian matrix are real numbers, since each diagonal element must be equal to its own complex conjugate (see Proof 6.3), and the off-diagonal elements are complex, but they must satisfy

$$A_{ij} = \overline{A_{ij}}$$

meaning that each element a_{ij} and its transpose $\overline{a_{ij}}$ are complex conjugates of each other.

Definition 6.4: Identification of Hermitian Matrices

Let matrix A be a complex $n \times n$ matrix. Matrix A is Hermitian if and only if:

1. The diagonal entries are real, and
2. The reflection of each non-diagonal entry is the conjugate of that entry.

Example 6.4: Hermitian Matrices

Consider the matrix A,

$$A = \begin{bmatrix} 3 & 2+i \\ 2-i & 5 \end{bmatrix}$$

To check if it is Hermitian, take its conjugate transpose:

1. Find A^T by transposing A
$$A^T = \begin{bmatrix} 3 & 2-i \\ 2+i & 5 \end{bmatrix}$$
2. Find A^* by conjugating A^T
$$A^* = \overline{A^T} = \begin{bmatrix} 3 & 2+i \\ 2-i & 5 \end{bmatrix}$$

Thus A is indeed Hermitian as $A = \overline{A^T} = A^*$.

Similarly, for the the matrix B,

$$B = \begin{bmatrix} 2 & 4+i & -3i \\ 4-i & -5 & -9-i \\ 3i & -9+i & 1 \end{bmatrix} = B^*$$

Theorem 6.2

Let matrix A be a complex $n \times n$ matrix. Matrix A is Hermitian if and only if

$$\langle A\mathbf{z}, \mathbf{w} \rangle = \langle \mathbf{z}, A\mathbf{w} \rangle$$

for any and all n-tuples $\mathbf{z}$ and $\mathbf{w}$ in $\mathbb{C}^n$.

Proof of Theorem 6.2

If matrix A is Hermitian, $A^* = A^T = \overline{A}$.
As $\mathbf{z}$ and $\mathbf{w}$ are vectors in $\mathbb{C}^n$, Definition 6.1 can be used to prove that $\langle \mathbf{z}, \mathbf{w} \rangle = \mathbf{z}^T \overline{\mathbf{w}}$. Hence,

$$\langle A\mathbf{z}, \mathbf{w} \rangle = (A\mathbf{z})^T \overline{\mathbf{w}}$$

as $A\mathbf{z}$ and $\mathbf{w}$ are considered two separate vectors, and

$$(A\mathbf{z})^T \overline{\mathbf{w}} = \mathbf{z}^T A^T \overline{\mathbf{w}} = \mathbf{z}^T \overline{A} \overline{\mathbf{w}} = \mathbf{z}^T (\overline{A\mathbf{w}})$$

which is equivalent to

$$\langle \mathbf{z}, A\mathbf{w} \rangle$$

All eigenvalues of a Hermitian matrix are real, even though the matrix itself may have complex entries. This property is vital for interpreting the results of transformations, particularly in quantum mechanics and data analysis. For any eigenvalue λ and corresponding eigenvector $\mathbf{v}$, we have $A\mathbf{v} = \lambda\mathbf{v}$. Taking the conjugate transpose of both sides gives

$$\mathbf{v}^* A = \mathbf{v}^* \lambda$$

Since $A = A^*$, this implies that

$$\lambda = \overline{\lambda}$$

Thus, λ must be real.

Additionally, the eigenvectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n$ of an n x n Hermitian matrix corresponding to distinct eigenvalues are orthogonal. These properties are reflected in Theorem 6.3.

Theorem 6.3

Let A be a complex $n \times n$ Hermitian matrix. Then,

1. The eigenvalues of A are real, and
2. The corresponding eigenvectors to distinct eigenvalues are orthogonal.

Proof of Theorem 6.3

Let matrix A be a complex matrix, with eigenvalues λ and μ. It is unknown whether or not the eigenvalues λ and μ are real numbers. Recall that $\mathbb{R} \subset \mathbb{C}$, so let the eigenvalues λ and μ be complex numbers. Let $\mathbf{z}$ and $\mathbf{w}$ be the corresponding non-zero complex eigenvectors of λ and μ respectively. This gives $A\mathbf{z} = \lambda\mathbf{z}$ and $A\mathbf{w} = \mu\mathbf{w}$.
Using Definition 6.3

$$\begin{aligned}\lambda\langle \mathbf{z}, \mathbf{w}\rangle &= \langle \lambda\mathbf{z}, \mathbf{w}\rangle \\ &= \langle A\mathbf{z}, \mathbf{w}\rangle\end{aligned}$$

Using Theorem 6.2

$$\begin{aligned}\langle A\mathbf{z}, \mathbf{w}\rangle &= \langle \mathbf{z}, A\mathbf{w}\rangle \\ &= \langle \mathbf{z}, \mu\mathbf{w}\rangle\end{aligned}$$

Therefore, using Definition 6.3 again

$$\langle \mathbf{z}, \mu\mathbf{w}\rangle = \overline{\mu}\langle \mathbf{z}, \mathbf{w}\rangle$$

This implies that λ and μ are conjugates of one another. If $\lambda = \mu$ and $\mathbf{z} = \mathbf{w}$, then $\lambda\langle \mathbf{z}, \mathbf{z}\rangle = \overline{\lambda}\langle \mathbf{z}, \mathbf{z}\rangle$ where the standard inner product of each eigenvector $\mathbf{z}$ is preserved. As $\mathbf{z}$ is non-zero, then $\langle \mathbf{z}, \mathbf{z}\rangle = ||\mathbf{z}||^2 \neq 0$, and thus $\lambda = \overline{\lambda}$. The complex number and its conjugate are one and the same, which is only possible if the number is real. Hence λ is real, and by iteratively taking the standard inner product of every eigenvector, (1) is proven. The same therefore applies to μ, thus giving

$$\lambda\langle \mathbf{z}, \mathbf{w}\rangle = \mu\langle \mathbf{z}, \mathbf{w}\rangle$$

If $\lambda \neq \mu$, then equality is only made possible if $\langle \mathbf{z}, \mathbf{w}\rangle = 0$, where the eigenvectors $\mathbf{z}$ and $\mathbf{w}$ ($\mathbf{z} \neq \mathbf{w}$) are orthogonal. This can be applied to any two distinct eigenvectors as a consequence of the iterative argument used to prove (1), hence proving (2).

These properties ensure that Hermitian matrices can be diagonalized using an orthogonal or unitary matrix, keeping the geometric structure of the space. Additionally, every Hermitian matrix can be diagonalized by a unitary matrix. That is, for any Hermitian matrix A, there exists a unitary matrix U such that

$$A = U\Lambda U^*$$

where Λ is a diagonal matrix containing the real eigenvalues of A, and U contains the corresponding eigenvectors, if and only if $A = A^*$ and $UU^* = U^*U = I$.

Unitary Matrices

A **unitary matrix** is a matrix A where its conjugate transpose A^* is equal to its inverse A^{-1}. It is thus denoted as U and satisfies the condition

$$UU^* = U^*U = I$$

as $U^* = U^{-1}$. It is important to note that not every unitary matrix is Hermitian.

A unitary matrix preserves the length (or norm) of vectors. For any vector $\mathbf{v} \in \mathbb{C}^n$, the transformation $U\mathbf{v}$ satisfies

$$||U\mathbf{v}|| = ||\mathbf{v}||.$$

This means that the transformation does not scale vectors, but it may change their direction, often corresponding to rotations or reflections in complex space.

The columns of a unitary matrix form an orthonormal set. That is, for any two columns $\mathbf{u}_i$ and $\mathbf{u}_j$ of the unitary matrix U, the inner product satisfies

$$\mathbf{u}_i^* \mathbf{u}_j = \delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

where δ is the *Kronecker Delta*, which is 1 if $i = j$ or 0 if not. This makes $\mathbf{u}_i$ and $\mathbf{u}_j$ orthogonal and implies that unitary matrices are analogous to orthogonal matrices in the real domain, preserving the structure of complex vector spaces.

Additionally, the eigenvalues of a unitary matrix lie on the unit circle in the complex plane, which is the set Z of complex numbers such that for every complex number z, $|z| = 1$. This means that for any eigenvalue λ of a unitary matrix U

$$|\lambda| = 1$$

This property reflects the fact that unitary matrices represent rotations and phase shifts rather than scaling transformations.

Definition 6.5: Identification of Unitary Matrices

Let matrix A be a complex square $n \times n$ Hermitian matrix. Matrix A is unitary if the rows and columns of A are orthonormal sets (in $\mathbb{C}^n$) and

$$A^* = A^{-1}$$

Let A be denoted by U. Thus,

$$UU^* = U^*U = I$$

Just like in the diagonalisation of real matrices using the equation $P^{-1}AP = D$, obtaining a unitary matrix U from a complex square matrix A is the same process as obtaining P from a real matrix, only with complex roots when solving the characteristic polynomial.

Example 6.5: Unitary Matrices

Consider the matrix U,

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ i & -i \end{bmatrix}$$

To check if it is unitary, calculate the conjugate transpose U^*,

$$U^* = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix}$$

Now verify that $U^*U = I$ by multiplying U^* and U,

$$U^*U = \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -i \\ 1 & i \end{bmatrix} \right) \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ i & -i \end{bmatrix} \right)$$

$$= \frac{1}{2} \begin{bmatrix} 1(1) + i(-i) & -i(-i) + 1(1) \\ i(i) + 1(1) & i(-i) + 1(1) \end{bmatrix}$$

$$= \frac{1}{2}\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

Since $U^*U = I$, the matrix is unitary. The eigenvalues of this matrix are 1 and -1, both of which lie on the unit circle, confirming that the matrix performs a rotation or reflection without scaling.

6.4 Decomposition of Complex Matrices

Unitary diagonalisation is a powerful tool that involves finding a unitary matrix U from a square matrix $A \in \mathbb{C}^{nxn}$ such that an upper triangular matrix T, with its diagonal entries being the eigenvalues of A, can be determined

$$A = UTU^*$$

thus providing a way to decompose a complex matrix into simpler components, especially in the context of complex vector spaces. This is described by **Schur's Theorem**, named after mathematician Issai Schur [5].

Theorem 6.4: Schur's Theorem

If A is a square $n \times n$ matrix, then there exists a unitary matrix U such that

$$U^*AU = T$$

where T is an upper triangular matrix with the entries of its main diagonal being the eigenvalues $\lambda_1, \lambda_2, ..., \lambda_n$ of A (including multiplicities).

Proof of Thoerem 6.4

Schur's Theorem can be proved using induction.

Let matrix A be a complex square $n \times n$ matrix. If $n = 1$, then A is already upper triangular and the theorem holds; this will be the base case. If $n > 1$, then assume the theorem is valid for $(n-1) \times (n-1)$ matrices, as if it had been for the case where $n = 2$, $n = 3$, and so on.

Let λ_1 be an eigenvalue of matrix A and $\mathbf{y}_1$ be an orthonormal eigenvector where $||\mathbf{y}_1|| = 1$. Then, extend $\mathbf{y}_1$ to an orthonormal basis $\{\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_n\}$ in $\mathbb{C}^n$ using the Gram-Schmidt Process and construct the unitary matrix $U_1 = [\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_n]$, where $U_1^* = U_1^{-1}$.

If this transformation is applied to matrix A, then

$$U_1^*AU_1 = \begin{pmatrix} \lambda_1 & b_{12} & b_{13} & \cdots & b_{1n} \\ 0 & & & & \\ 0 & & A' & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

Note how the first column begins with λ_1 and then all zeros, where $b_{12}, b_{13}, ..., b_{1n}$ are some values. This is because the first column in U_1 is an eigenvector $\mathbf{y}_1$ of A corresponding to the eigenvalue λ_1.

Additionally, A' is some $(n-1) \times (n-1)$ matrix with the other eigenvalues $\lambda_2, \lambda_3, ..., \lambda_n$ of A, though not necessarily in that order, representing the part of A that interacts with the other eigenvectors $\mathbf{y}_2, \mathbf{y}_3, ..., \mathbf{y}_n$ that are orthogonal to $\mathbf{y}_1$.

Now, apply the transformation again on A'. Assume that the theorem is true for smaller matrices using the base case where $n = 1$, and find another unitary matrix U_2.

$$U_2^* A' U_2 = \begin{pmatrix} \lambda_2 & b_{22} & b_{23} & \cdots & b_{2n} \\ 0 & & & & \\ 0 & & A'' & & \\ \vdots & & & & \\ 0 & & & & \end{pmatrix}$$

This follows on from the first established transformation. Again, the first column of U_2 being an eigenvector $\mathbf{y}_2$ interacting with A ensures that the corresponding eigenvalue λ_2 is at the upper left corner and the rest of the column are all zeroes. A'' is the next smaller matrix, with eigenvalues $\lambda_3, ..., \lambda_n$. By induction, this transformation continues until there is only a 1×1 matrix with entry λ_n. Hence, the diagonal entries are the eigenvalues of A, and the lower triangle of zeroes formed makes the upper triangle.

Schur's Theorem can also be proved as follows [6].

If $U^*AU = T$, then $AU = UT$. Let A, A', U_1, and U_2 be as above, where A' and U_2 is $(n-1) \times (n-1)$. Similarly, as $U_2^* A' U_2 = T'$, so is $A' = U_2 T' U_2^*$. Now, let

$$U = U_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix}$$

Then,

$$AU = AU_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix}$$

and $U_1 U_1^* AU = AU = U \begin{bmatrix} \lambda_1 & ... \\ 0 & A' \end{bmatrix}$

$$= U_1 \begin{bmatrix} \lambda_1 & ... \\ 0 & A' \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix} = U_1 \begin{bmatrix} \lambda_1 & ... \\ 0 & A'U_2 \end{bmatrix}$$

Therefore, $A'U_2 = U_2 T'$

$$= U_1 \begin{bmatrix} \lambda_1 & ... \\ 0 & U_2 T' \end{bmatrix}$$

which can be rewritten as

$$= U_1 \begin{bmatrix} 1 & 0 \\ 0 & U_2 \end{bmatrix} \begin{bmatrix} \lambda_1 & ... \\ 0 & T' \end{bmatrix}$$

Now, notice how $\begin{bmatrix} \lambda_1 & ... \\ 0 & T' \end{bmatrix}$ is in upper triangular form, which is the result of the initial unitary transformation. Hence, the result is

$$UT$$

thereby proving that

$$AU = UT$$

and hence there exists some U such that

$$U^* AU = T$$

The matrix U being unitary preserves the orthogonality and norm of vectors, ensuring that the transformation A does not distort the underlying space. Since T is upper triangular, the Schur decomposition simplifies the study of eigenvalues and eigenvectors of complex matrices. The eigenvalues of A appear on the diagonal of T, and U contains a basis of orthonormal vectors for the vector space. The Schur decomposition process is illustrated in Example 6.6.

Example 6.6

Given $A = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 2 \end{bmatrix}$, find U and T such that $U^*AU = T$.

Begin by finding the eigenvalues and eigenvectors of A by solving

$$\det(\lambda I - A) = 0$$

$$(\lambda I - A) = \begin{bmatrix} \lambda & 1 & 0 \\ -1 & \lambda & 0 \\ 0 & 0 & \lambda - 2 \end{bmatrix}$$

Expanding across column 3,

$$\det(\lambda I - A) = (\lambda - 2)(\lambda^2 + 1) = 0$$

The solutions are $\lambda_1 = 2$, and

$$(\lambda^2 + 1) = 0$$

$$\lambda^2 = -1$$

$$\lambda = \pm\sqrt{-1} = \pm i$$

hence

$$\lambda_2 = i, \lambda_3 = -i$$

This means that the eigenvalues are complex.

Next, find the complex eigenvectors $\mathbf{z}_1, \mathbf{z}_2$, and $\mathbf{z}_3$ corresponding to λ_1, λ_2, and λ_3 respectively. As an example, for $\lambda_2 = i$:

$$(\lambda I - A) = \begin{bmatrix} i & 1 & 0 \\ -1 & i & 0 \\ 0 & 0 & -2 + i \end{bmatrix}$$

As $\mathbf{z}_2 = \begin{bmatrix} z_{21} \\ z_{22} \\ z_{23} \end{bmatrix}$ satisfies $(\lambda I - A)\mathbf{z}_2 = 0$, then

$$\begin{bmatrix} i & 1 & 0 \\ -1 & i & 0 \\ 0 & 0 & -2 + i \end{bmatrix} \begin{bmatrix} z_{21} \\ z_{22} \\ z_{23} \end{bmatrix} = 0$$

Therefore, $z_{23}(-2 + i) = 0$ and $z_{23} = 0$. Remember that $-2 + i$ is a complex number. Also

$$z_{21}(i) + z_{22} = 0$$

$$-z_{21} + z_{22}i = 0$$

$$z_{21} = z_{22}i$$

and

$$z_{22}i(i) + z_{22} = z_{22}i^2 + z_{22} = -z_{22} + z_{22} = 0$$

Hence, z_{22} is free. Thus, assuming z_{22} to be 1 gives

$$\mathbf{z}_2 = \begin{bmatrix} z_{22}i \\ z_{22} \\ 0 \end{bmatrix} = \begin{bmatrix} i \\ 1 \\ 0 \end{bmatrix}$$

It follows that $\mathbf{z}_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$ and $\mathbf{z_3} = \begin{bmatrix} -i \\ 1 \\ 0 \end{bmatrix}$.

Now, normalize the eigenvectors. $\mathbf{z}_1$ is already normalized.

$$||\mathbf{z}_2|| = \sqrt{|1|^2 + |1|^2 + |0|^2} = \sqrt{2}$$

$$||\mathbf{z_3}|| = \sqrt{|(-1)|^2 + |1|^2 + |0|^2} = \sqrt{2}$$

Hence, normalizing $\mathbf{z}_2$ and $\mathbf{z_3}$ and combining them to make U gives

$$U = \begin{bmatrix} 0 & \frac{i}{\sqrt{2}} & \frac{-i}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & 0 & 0 \end{bmatrix}$$

U can be confirmed as unitary, $U^{-1} = U^*$.

$$U^* = \overline{U^T} = \begin{bmatrix} 0 & 0 & 1 \\ \frac{-i}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{i}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

and it follows that

$$UU^* = U^*U = U^{-1}U = UU^{-1} = I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Hence,

$$U^*AU = \begin{bmatrix} 2 & 0 & 0 \\ 0 & i & 0 \\ 0 & 0 & -i \end{bmatrix} = T$$

The Spectral Theorem

The **Spectral Theorem** provides a deeper insight into Hermitian matrices [5], particularly those with complex entries. Hermitian matrices are those that satisfy the condition $A = A^*$, where A^* is the conjugate transpose of A. Hermitian matrices are significant because they have real eigenvalues, even though their entries may be complex.

While the eigenvalues of a Hermitian matrix are real, the eigenvectors associated with these eigenvalues are generally complex. These complex eigenvectors form an orthonormal basis for the complex vector space, which means they are not only orthogonal but also normalized in the context of complex inner products. The unitary matrix U in the Spectral Theorem contains these complex eigenvectors as its columns. Further, since Hermitian matrices involve complex conjugate

pairs in their off-diagonal elements, their diagonalization through the Spectral Theorem relies on complex arithmetic. The diagonal matrix Λ contains the real eigenvalues of the Hermitian matrix, while the unitary matrix U preserves the orthonormal structure of the complex eigenvectors during the diagonalization process.

Consider the Hermitian matrix A,

$$A = \begin{bmatrix} 5 & 2i \\ -2i & 3 \end{bmatrix}$$

Here A contains complex numbers in the off-diagonal element, but it is still Hermitian since $A = A^*$. Applying the Spectral Theorem,

$$A = U\Lambda U^*$$

where U is a unitary matrix containing complex eigenvectors, and Λ is a diagonal matrix containing the real eigenvalues of A. The real eigenvalues of A are 6 and 2, but the complex numbers in the matrix are handled through the unitary transformation by U. The unitary matrix U, being composed of complex vectors, ensures that the transformation does not distort the lengths or angles of the vectors in the complex space.

Both Hermitian and unitary matrices are essential in preserving the structure of vector spaces in the complex domain. Hermitian matrices guarantee real eigenvalues and orthogonal eigenvectors, ensuring that the transformations they represent can be easily interpreted, even in complex spaces, while unitary matrices ensure that the length (or norm) of vectors is preserved, allowing for stable transformations that do not distort the underlying vector space.

Additionally, both the Schur decomposition and the Spectral Theorem involve complex numbers in critical ways. The eigenvalues of a matrix, which may be complex, appear on the diagonal of the triangular matrix T (in Schur decomposition) or the diagonal matrix Λ (in the Spectral Theorem). The eigenvectors of the matrix are often complex-valued, especially in the case of unitary transformations. These complex vectors form an orthonormal basis in the complex vector space, meaning they preserve the geometric structure during transformations. Complex conjugation plays a key role, particularly in the Spectral Theorem, where Hermitian matrices (with potentially complex entries) can still be diagonalized, yielding real eigenvalues and complex eigenvectors.

6.5 Real-world Applications of Complex Matrices

In data analysis, Hermitian matrices, the Spectral Theorem, and Schur decomposition are frequently used in techniques such as principal component analysis (PCA), covariance matrix computations, and clustering algorithms. Covariance matrices, which capture the variance and relationships between variables, are often Hermitian when the data involves complex numbers. This property ensures that the eigenvalues derived from the matrix are real, which is essential when interpreting the principal components or clusters in high-dimensional data.

In quantum computing, unitary matrices play a pivotal role in describing quantum gates—the building blocks of quantum algorithms. Quantum gates, such as the Hadamard gate and Pauli matrices, are represented by unitary matrices. These gates transform quantum states while preserving their overall probability distribution (since the norm of the quantum state vector must remain 1). Further, the Spectral Theorem plays a crucial role in analyzing quantum operators. Quantum systems are described by state vectors in a complex Hilbert space, and the evolution of these systems is governed by Hermitian operators. Since Hermitian operators have real eigenvalues, they represent observable quantities in quantum mechanics (e.g., energy, position, momentum).

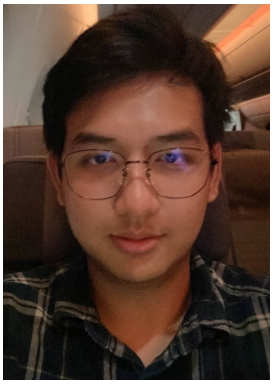
6.6 Conclusion

This article explored the foundational role of complex numbers in linear algebra and their applications in fields like quantum computing and data analysis. It examined the properties of complex numbers, which extend real-number systems into multidimensional spaces, providing a more comprehensive framework for analyzing complex transformations. The importance of eigenvalues and eigenvectors in understanding linear transformations involving scaling and rotation in vector spaces was highlighted, especially in quantum systems where complex eigenvalues often correspond to oscillatory behaviors.

The role of the complex conjugate and determinant in matrix theory were investigated, showing how these concepts simplify matrix operations and ensure the stability of linear systems. The use of complex numbers in solving systems of linear equations was discussed in depth, illustrating the power of techniques like Gaussian elimination and matrix inversion in advanced fields. Finally, the significance of Hermitian and unitary matrices, which preserve orthogonality and norms in complex spaces, and their central role in quantum computing and machine learning algorithms, was discussed.

By connecting these abstract mathematical tools with real-world applications, this article demonstrated how complex numbers, through their role in eigenvalue decomposition, matrix theory, and linear systems, are indispensable in modern data analysis and quantum technologies. The power of these tools lies in their ability to capture and manipulate multidimensional relationships, providing deeper insights into systems that are otherwise difficult to interpret using real numbers alone.

About the Authors



Visal Dam is a second-year Bachelor of Cybersecurity student at Deakin University, where he is also currently employed as a Maths Mentor. Visal enjoys teaching, reading Wikipedia articles, and discussing cultural traditions over tea. As a new-found lover of Data Science and Machine Learning, he hopes to utilize such approaches to Cybersecurity to better support the field in the modern digital age.

Amanda Roberts is a second-year Bachelor of Computer Science student at Deakin University, majoring in Data Science and minoring in Computational Mathematics. Passionate about life-long learning, Amanda has returned to study as an online student, carefully balancing professional commitments alongside her coursework. She is hopeful that her eager promotion of her university experience will encourage more women to return to study in STEM fields, and demonstrate that there is space for everyone to be involved in this vibrant environment.

Acknowledgements

(**Visal**) First and foremost I would like to thank my family, namely my parents, grandmother, and little sister, for their continuous support of my academic journey. I dedicate my contributions to this article to them. I would also like to acknowledge my former high school Physics, Computer

Science, and Mathematics teachers Mr. Zay Yar Thun, Mr. Ye Htut Win, and Mr. Chacko Thommey, respectively, as it was their extensive knowledge in their fields and inspirational teaching styles that sparked my passion to share what I know as a peer mentor. Finally, I would like to thank Dr. Simon James for his insightful support and patient mentorship during my study of Linear Algebra, as well as my co-author Amanda Roberts for her hard work and collaboration. I am incredibly grateful and honored to have been able to contribute to this yearbook with her.

(**Amanda**) I would like to thank my co-author, Visal Dam, for being a generous collaborator and for his contribution to this article. It has been my privilege to work alongside him. My thanks also to Associate Professor Simon James and Sherry Shaharyar for their guidance in SIT292 Linear Algebra for Data Analysis, which provided the foundation for this article. When gifted educators pass along their passion for a particular subject, you can't help but catch fire and love what you learn. Finally, thanks to family and friends who always patiently listen while I talk about all things computer science, and make me feel like a rockstar in the process.

References

All figures presented were created using Tikz [7] in LaTeX.

[1] Charles R. Card and Gary G. Miller. Bootstraps and scaffolds: What a cognitive-historical analysis of the complex number system reveals about numerical cognition. *Journal of Humanistic Mathematics*, 14(2):452–514, 2024.

[2] Harold Cohen. *Complex analysis with applications in science and engineering*. Springer, 2nd edition, 2007.

[3] E. Fowler. *Introduction to Complex Numbers*. Latex Community, 2014. Available online: https://www.overleaf.com/articles/introduction-to-complex-numbers/nszvsffyrcrh.

[4] Ricardo Karam. Schrödinger's original struggles with a complex wave function. *American Journal of Physics*, 88(6):433–438, 2020.

[5] W. K. Nicholson. *Linear Algebra with Applications*. Lyryx, Revised A edition, 2021.

[6] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 4th edition, 2009.

[7] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

7

Linear algebra in neural network training

Ari Robin and Tuan Thanh (Chris) Lu

Abstract

In this article, we explore how important linear algebra and matrix operations are in the context of machine learning, and in the construction and training of neural networks. Without these tools, modern machine learning couldn't exist, and by understanding and leveraging concepts such as matrix multiplication, and the properties of vector spaces, we'll see how linear algebra enables these processes within machine learning such as forward propagation, back-propagation, and the optimization techniques of neural networks. These tools are vital in various technological applications that we'll begin to explore, including image recognition and the explosion recently seen in natural language processing. By understanding these tools, we can begin to understand the ability of allowing neural networks to learn from data and make accurate predictions based on that data.

7.1 Introduction to Neural Networks

This article investigates how neural networks utilize matrix operations to update weights during training, perform error correction through back-propagation, while maintaining model accuracy. It will also focus on how to efficiently and accurately compute both forward and backward propagation passes and enhance the stability of the network through proper weight adjustments.

7.2 Preliminaries

Here we introduce the notation, background knowledge and conventions used in this paper.

Matrices

- Matrices are denoted by uppercase letters, e.g., W, A, B. The element in the i-th row and j-th column of a matrix W is w_{ij}.
- Vectors will usually be denoted by bold lowercase letters, e.g, $\mathbf{u}$, $\mathbf{v}$.

Operations

- The transpose of a matrix W is denoted by W^T.
- Matrix multiplication follows standard rules: if $W \in \mathbb{R}^{m\times n}$ and $\mathbf{x} \in \mathbb{R}^n$, then $W\mathbf{x} \in \mathbb{R}^m$.

Inner Products

For two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$, their inner (dot) product is:

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 & b_2 & ... & b_n \end{bmatrix} = \sum_{i=1}^{n} a_i b_i.$$

Hadamard Product

For two vectors with the same size $\mathbf{x}$ and $\mathbf{y}$, we define their Hadamard product $\mathbf{x} \odot \mathbf{y}$ by:

$$\mathbf{x} \odot \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \odot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_n y_n \end{bmatrix}$$

Sets and Indexing

We typically index from 1 to n, unless otherwise stated.

Neural Network Layers

An individual neural network layer applies a linear transformation to an input vector $\mathbf{x}$ resulting in $\mathbf{z} = W\mathbf{x} + \mathbf{b}$, where W is the weight matrix, and $\mathbf{b}$ is the bias vector.

Activation Functions

The two most common activation functions used in deep neural network are:

- ReLU function: A real function in the form:

$$f(x) = \max(0,\ x)$$

- Sigmoid function: A real, differentiable and continuous function of the form:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Loss Functions

Let $\mathbf{a}$ represent the network's predicted output and $\mathbf{y}$ represent the target output. A common loss function we refer to is the Mean Squared Error (MSE):

$$L = \frac{1}{2} \sum_i (a_i - y_i)^2.$$

Gradients and Updates

Using backpropagation, we calculate the gradient of the loss with respect to the weights W and denote this as ∇W. The weights are then updated through the gradient descent process as:

$$W_{\text{new}} = W_{\text{old}} - \eta(\nabla W),$$

where η is the learning rate of the model.

Backpropagation Concepts

We summarise the outputs for each layer of the network:

- $\mathbf{o}_i$: output (after applying activation function) produces for layer $i+1$ (so o_0 will be the vector denoting the input layer).
- $\mathbf{z}_i$: linear combination output of layer i.
- δ_i: Error on the i-th layer.

Unless specified otherwise, we will use these notations and conventions throughout this paper.

7.3 Linear Algebra and Neural Network Components

In neural networks, the concepts required to create and train those networks require a deep understanding of linear algebra, particularly matrix multiplication and the results of those operations. In this article we'll explore how these matrix operations are foundational to neural network training, especially in tasks like forward propagation, back-propagation, and regularization. We're going to start by exploring some of the key linear algebraic terms used within neural networks before we dive deeper into how these networks are formed from a mathematical standpoint.

Matrix Operations in Neural Networks

Neural networks rely on matrix operations and linear algebra for both forward propagation (to compute its outputs) and back-propagation [15] (to adjust weights based on errors, vital for making a system usable and effective). As we'll see, forward propagation involves multiplying the input vector (that represents features of the data) by the weight matrix. The result is then passed through a function (specifically, an activation function) to generate the neural network's output.

In contrast, backpropagation is the process of using the error (the difference between the predicted and actual output of the network) to update the weight matrix, ensuring the network learns over time. This process simulates a learning process, where enough passes of this with larger amounts of data will drastically improve a network's ability to predict (and even generate) text or content of its own. There are more pieces to this Artificial Intelligence (AI) puzzle, but understanding this is critical as a concrete foundation to how these relatively new technologies work.

Neural Networks Components

A neural network is a computational program or model that mimics the complex functions of the human brain [9].

We explore the structure of a neural network as in Figure 7.1. A neural network consists of:

1. **Nodes/Units/Neurons** - the most fundamental part of a neural network. The units will be arranged in different layers to constitute a complete neural network.
2. **Input layer** - consisting of inputs x_1, x_2, ..., x_n denotes the values obtained from data analysis and other outside world sources.
3. **Hidden layer(s)** - can have one or many hidden layers. Each layer consists of nodes a_1, a_2, ..., a_n and has the responsibility of transforming the input to data with useful insights for the output layer.
4. **Output layer** - denoted as a column vector $\mathbf{y}$, this layer provides an output for comparing with provided inputs.

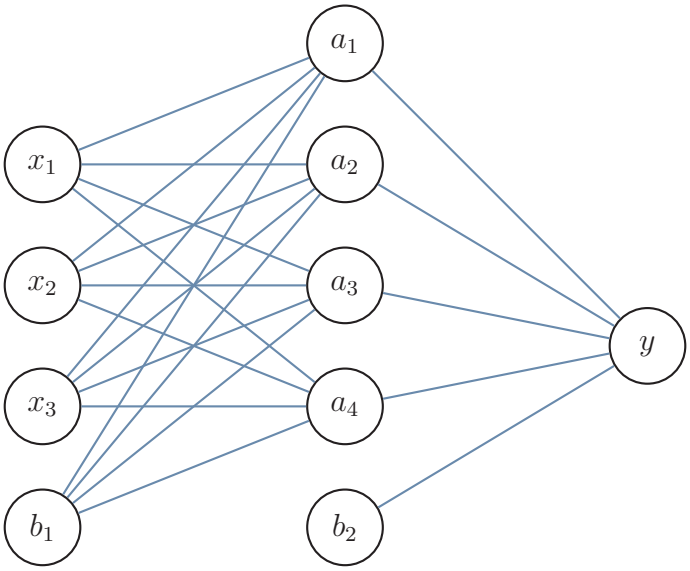


Figure 7.1: Neural Network with 1 hidden layer

5. **Weights** - determine the influence of one unit on another unit. The matrix of weights with connecting edges from layer i to layer $i+1$ is denoted as W_i.
6. **Bias** - denoted as $\mathbf{b}_i$ for each layer $i+1$. Bias is used to adjust the outputs for successful learning.
7. **Activation functions** - defines the output of the neuron in terms of the activity level at its input. Its main purpose is to make the model more flexible with unusual patterns.

So a neural network can have a custom number of hidden layers. A neural network with *at least 2 hidden layers* is called a deep neural network.

7.4 Fundamental Processes of Neural Networks

We first explore how the processes that enable neural networks to learn and perform their functions work, including presenting proofs and additional concepts.

Forward Propagation

In forward propagation, each layer in the network performs a matrix multiplication. The input to the layer is multiplied by the existing weight matrix, and the result is adjusted by the bias term. The bias term is an additional parameter added to the weighted sum of inputs. This allows the network to shift the activation function with the goal of creating a more suitable structure for the data being received [19].

This process is represented as:

$$\mathbf{z} = W(\mathbf{x} + \mathbf{b}) \tag{7.1}$$

where W is the weight matrix, $\mathbf{x}$ is the input vector, and $\mathbf{b}$ is the bias term. With $\mathbf{z}$ being the linear combination of these terms, it serves as the input to the activation function. This introduces non-linearity into the model, allowing the network to capture complex patterns in the data. The value of $\mathbf{z}$ directly affects how the network transforms the input at each layer.

It's also worth noting that including the bias term as an additional constant vector term is equivalent to appending an extra input of 1 to each layer, and then expanding the weight matrices. Doing this means the bias calculation can be handled within the matrix operation itself, simplifying the process and saving on computational cost.

Forward Propagation Visualization and Numerical Example

Figure 7.2 provides a visualization of a forward propagation iteration cycle through a simple neural network with three inputs, three hidden nodes, and one output.

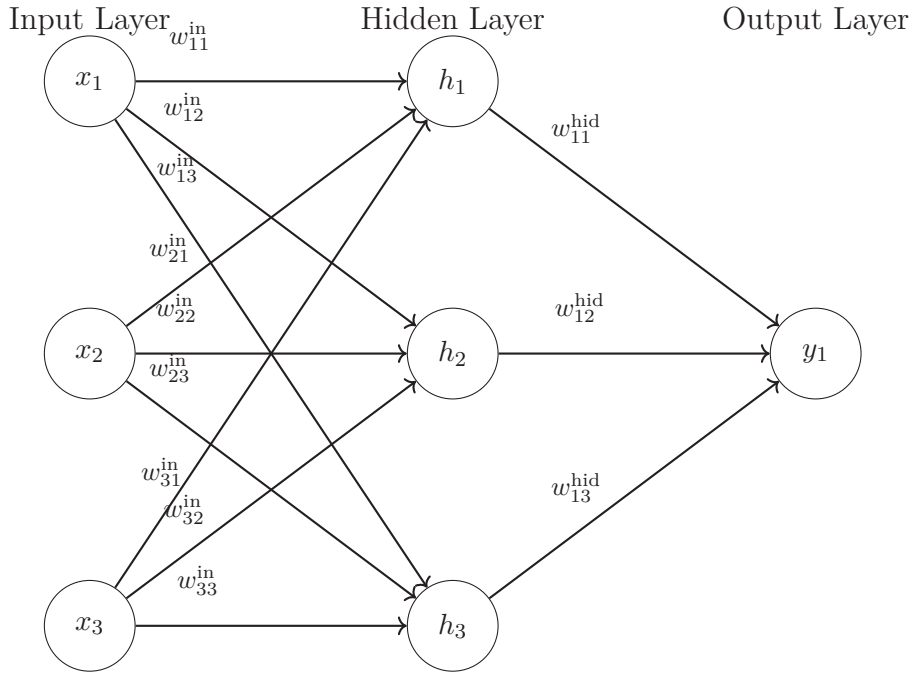


Figure 7.2: A simple three-layer neural network diagram.

Below is a visualization of a forward propagation iteration cycle through a simple neural network with three inputs, three hidden nodes, and one output. Forward propagation is the core process that allows a neural network to take in input data and transform it into a prediction by passing it through however many layers of the network may exist. Each node's inner layer (the hidden layer in our example) performs a weighted sum of the inputs at each layer. We'll omit the activation function and biases here to highlight the general logic of forward propagation. When the process reaches the output layer, the values converge into an output layer value. By visualizing and computing each stage, we can better understand how a network arrives at its prediction [10].

As we can see, each of the input layer elements has a connection to each node in the next layer. Then each node within the hidden layer has its own connection to the node in the output layer. Each of these weight connections has its own value, multiplying the node's value by the weight value defined in the weight matrix. While real networks often have many hidden layers, we focus on one for demonstration.

Below is a numerical example that defines a value for each node in the diagram, as well as the weight values. We can then calculate what our output value y_1 will be after the first iteration of this neural network. If tasked with solving this problem, we would be given the input values and the weight matrix for each set of connections between the layers.

Example 7.1: Neural network calculation

Input Values:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Weight Matrix for Input Layer to Hidden Layer:

$$W^{\text{in}} = \begin{pmatrix} 0.5 & 0.1 & 0.2 \\ 0.4 & 0.3 & 0.8 \\ 0.9 & 0.7 & 0.6 \end{pmatrix}$$

Weight Matrix for Hidden Layer to Output Layer:

$$W^{\text{hid}} = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.3 \end{pmatrix}$$

Compute Hidden Layer Values (for example, h_1):

$$h_1 = (0.5 \cdot 1) + (0.4 \cdot 2) + (0.9 \cdot 3) = 4.0$$

Similarly, we would compute h_2 and h_3. Suppose:

$$\begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} = \begin{pmatrix} 4.0 \\ 2.8 \\ 3.6 \end{pmatrix}$$

Now compute the output:

$$y_1 = (0.2 \cdot 4.0) + (0.5 \cdot 2.8) + (0.3 \cdot 3.6) = 3.28$$

Thus, the output (prediction) of our model is 3.28.

This demonstrates how neural networks transform input data into predictions using matrix operations. We will later see how errors from predictions feed into backpropagation, allowing the network to learn and adjust its weights.

Backpropagation

In backpropagation, matrix derivatives (or gradients) are computed to adjust the weights. These gradients show how the error changes in relation to the weights. By calculating gradients and updating the weights accordingly, the network's predictions become more accurate over time. First, we're going to look at how the resulting process of backpropagation is used to calculate this weight update [1]. The formula for this is:

$$W_{\text{new}} = W_{\text{old}} - \eta \, (\nabla W) \tag{7.2}$$

Here, W_{new} and W_{old} represent the new and old weight matrices, respectively. The parameter η (eta) is the learning rate, and ∇W is the gradient of the weight matrix with respect to the error. By employing this formula, each weight is adjusted in proportion to its contribution to the overall error. This iterative process enables the network to learn from its mistakes. Over many iterations, as the weights are continually updated, the network moves toward an optimal set of weights that allow it to produce the most accurate predictions possible, given its training data and computational resources.

This iterative weight adjustment is a core principle of AI and machine learning, particularly in training deep learning models. Without such techniques, capturing complex patterns and generalizing effectively to new data would be unattainable with current methodologies.

Backward Propagation Diagram

Backward propagation works in the opposite direction to forward propagation, traveling backward through the network to update each weight. It uses the error gradient and the learning rate to slightly alter each weight value as it moves from the output layer back toward the input layer. This process typically occurs after error correction and any regularization steps, which we will discuss in more detail later. For now, let's focus purely on the order and logic of the backward propagation steps.

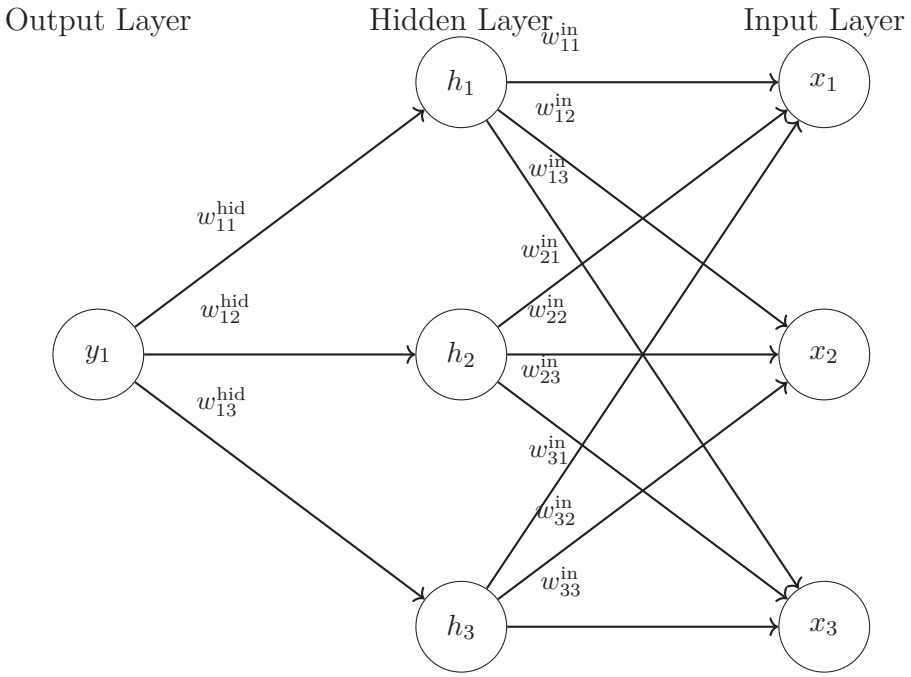


Figure 7.3: The backward propagation process following the previous diagram.

Following our earlier forward propagation numerical example, in the case of the diagram shown in Figure 7.3:

1. The process starts at the output layer by computing the error at y_1, the single output node in our earlier example.
2. This error is then propagated back to the hidden layer, where the weights W_{hidden} (e.g., w_{41}, w_{42}, w_{43}) connecting the hidden nodes to the output node are adjusted based on factors like the error gradient.
3. Finally, the error is propagated back to the input layer, where the weights from the initial weight matrix W_{input} connecting the input nodes to the hidden nodes are updated.

This visualization demonstrates how the backpropagation process moves backward through the network, adjusting the weights to minimize the error with each iteration. The next time that the forward propagation process is run, the weights will be slightly different and will therefore produce a different output value. This is where the learning of machine learning comes in. The

network iterates forward and backward each time, gradually correcting itself and converging to a more accurate prediction [7].

Gradient Descent Visualization

As we've seen, backpropagation utilizes a concept known as gradient descent, which uses the gradient ∇W to update the weights. The weight matrix is updated gradually by incrementally changing towards the direction that minimizes the calculated error. It's important for our network to understand its errors to improve. We'll look into how to calculate error shortly, where we'll fully understand how gradient descent is a fundamental optimization technique used to minimize errors in machine learning models, including neural networks. The blue dots in Figure 7.5 indicate where the weight value is at each iteration. It's important to understand that gradient descent enables weights of the network to be adjusted incrementally in the direction that reduces the error the most, by calculating the gradient of the error function (seen as the blue line in the example below). This informs us as to how the error changes when responding to the weight changes. In each iteration of the descent, the weights are updated and moved toward the most optimal direction. We can see in this visualization that this is to minimize the error value (seen on the y-axis).

As we'll see shortly, the amount the weights are permitted to adjust is dictated by a pre-defined learning rate η. When used within the backpropagation propagation, gradient descent updates the weight matrices in our network, enabling learning and error correction.

Multiple Iterations of Training

If we ran this model a thousand more times, we would find that the output value of y_1 found in each forward propagation cycle would adjust as the weights are updated during training when being processed through forward propagation, backpropagation, and gradient descent. The network would minimize the error between its predictions and the target values, gradually converging towards an optimal solution where y_1 closely matches the desired output, given enough iterations.

In a real-world context, some modern cars train themselves to self-drive by processing camera video data through neural networks using forward propagation. By converting visual input into numerical information and data, the network multiplies this data by weights and applies activation functions (as we'll discuss soon), allowing the network to detect and classify objects like pedestrians and traffic signs. Just like we saw with the example above, the car's neural network adjusts its weights through backpropagation and gradient descent over many iterations. This continuous refinement minimizes the error between its predictions and actual observations, enabling the car to accurately interpret its environment and drive itself safely after extensive training [4]. The important thing to note is that the car is able to improve its ability to drive itself through enabling the process of forward propagation by being able to convert the camera video input into structured data, which is then fed into a network in a very similar way as the concepts explained above.

Backpropagation Weight Update Formula

We'll now present proof of the formula used to update the weight matrix, based on the existing weight matrix W_{old}, the learning rate η (eta), and the error gradient matrix ∇W, used towards the end of the backpropagation process:

$$W_{new} = W_{old} - \eta\,(\nabla W) \tag{7.2}$$

This one is more complex, so we're going to take it one step at a time while also touching on the individual steps and concepts that make this iterative process of improving a model work.

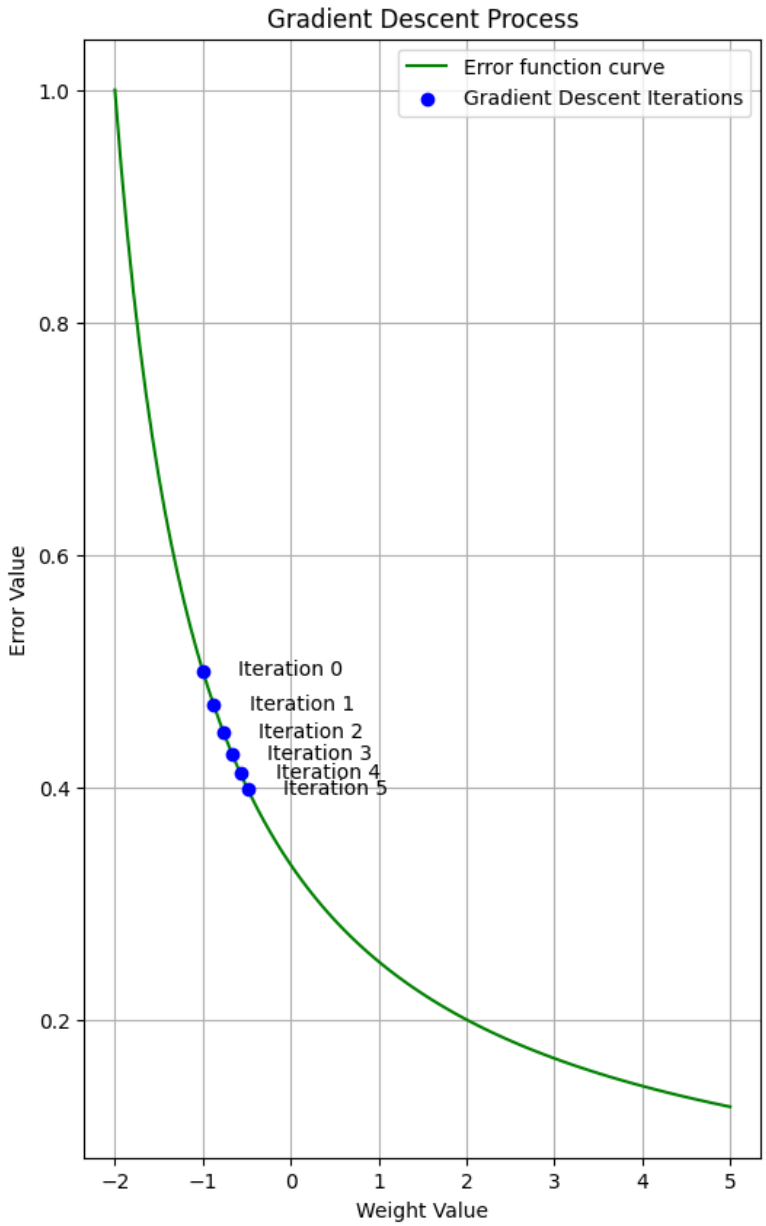


Figure 7.4: Gradient Descent Visualization. Created using `matplotlib` in Python.

1. Loss Function

Firstly, consider a neural network with actual output (or predicted output) of its model $\mathbf{a}$ and a target output $\mathbf{y}$. The loss function L has the purpose of quantifying the difference between the predicted output $\mathbf{A}$ and the desired target output $\mathbf{y}$. The loss function we'll use in this article is the Mean Squared Error (MSE) [13] (although there are many others used, this is very popular). This loss function is defined as:

$$L = \frac{1}{2} \sum (a_i - y_i)^2 \tag{7.3}$$

This measures the squared difference between each of the actual outputs of $\mathbf{a}$ and the target

outputs $\mathbf{y}$, which is important for determining the final error gradient used in our backpropagation formula. For this purpose, we aim to minimize the loss function L by adjusting the weights W by computing the gradient of L with respect to W. This gradient informs us as to how much L changes in contrast/response to the changes in the weights W [2].

2. Forward Propagation

As before, we know that the output $\mathbf{a}$ is a function of the weighted sums in $\mathbf{z}$, where:

$$\mathbf{z} = W(\mathbf{x} + \mathbf{b}) \tag{7.1}$$

and $\mathbf{a}$ is the result of applying an activation function, like σ, to $\mathbf{z}$. Using this, we know that:

$$\mathbf{a} = \sigma(\mathbf{z}) \tag{7.4}$$

As an example, the sigmoid activation function is commonly used, defined as:

$$\sigma(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}} \tag{7.5}$$

We'll go into more depth with what this means shortly; however, this function frames the input z into the range between 0 and 1, while introducing non-linearity into the network. It's important for the activation function to introduce non-linearity into the model. This means that the model can recognize and represent more complex relationships between its data inputs and model outputs. This is in contrast to simply using combinations of linear transformations.

Without a non-linear activation function, the entire neural network would behave like a single linear transformation, regardless of the number of layers. This would limit the model's ability to capture complex patterns severely, due to linear transformations only being able to model simple relationships because of their preservation of the proportionality and additivity of the inputs. By introducing non-linearity with the activation function, the network gains the ability to better approximate more complex functions and boundaries so it can learn far more intricate patterns within the data, vastly improving its predictive power.

3. Chain Rule Application

We are starting to see the dependent chain of relationships within this framework. The loss function result L depends on output $\mathbf{a}$, which in turn depends on forward propagation result $\mathbf{z}$, which depends on the weight matrix W. For us to calculate our resulting error gradient $\frac{\partial L}{\partial W}$, this represents the gradient of loss function L with respect to the weight matrix W, used to update the weights and minimize the loss. Seeing as we have a chain of dependencies, we can define and use the following chain rule [14]:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \mathbf{a}} \cdot \frac{\partial \mathbf{a}}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial W} \tag{7.6}$$

Here, each term describes how the loss changes in response to changes in each variable. The ∂ (partial derivative) is used to indicate how the loss function changes with respect to each specific variable while holding others constant. This distinction is very important, as it captures the incremental changes in the network as each variable is adjusted. By taking these derivatives, we're able to quantify how errors propagate back through the network during backpropagation and how the weights should be adjusted to reduce the overall error. To calculate each of these, we use a very similar structure for each:

Derivative of loss with respect to output A:

$$\frac{\partial L}{\partial \mathbf{A}} = \mathbf{A} - \mathbf{y} \tag{7.7}$$

being the error between each element of the actual/predicated output $\mathbf{a}$ and the target output $\mathbf{y}$.

Similarly, we have the same for the sigmoid activation function to the weighted sum $\mathbf{z}$, introducing non-linearity into the model:

$$\frac{\partial \mathbf{a}}{\partial \mathbf{z}} = \mathbf{a} \cdot (\mathbf{1} - \mathbf{a}) \tag{7.8}$$

Finally, the derivative of the weighted sum $\mathbf{z}$ with respect to the weights W:

$$\frac{\partial \mathbf{z}}{\partial W} = \mathbf{x} \tag{7.9}$$

Notice that this is simply each element of the input vector $\mathbf{x}$. This is due to $\mathbf{z}$ being a linear combination of W and $\mathbf{x}$.

4. Error Gradient Calculation

We now combine these partial derivatives, yielding the gradient of the loss function with respect to the weights W:

$$\nabla W = \frac{\partial L}{\partial W} = (\mathbf{a} - \mathbf{y}) \cdot \mathbf{a} \cdot (\mathbf{1} - \mathbf{a}) \cdot \mathbf{x}^T \tag{7.10}$$

We can see that the first two dot product terms $\mathbf{a} - \mathbf{y}$ and $\mathbf{a} \cdot (\mathbf{1} - \mathbf{a})$ are simply the way we obtain the derivatives for our first two explanations above. Seeing as our third formula leads to the input vector $\mathbf{x}$, it's important that we use the transpose $\mathbf{x}^T$ in this case. Seeing as $\mathbf{x}$ was the input used during forward propagation, we need to transpose this input vector to be able to work back through the network to backpropagate the error, hence the "back" in backpropagate! Using this formula, we obtain both the direction and magnitude of the steepest ascent of the loss function, with respect to the weights.

5. Applying the Learning Rate to Obtain the Scaled Gradient

We now have our error gradient ∇W, informing the direction in which the error would increase if weights were adjusted in that direction. This is important for us to know how we can improve our model. We're almost there! When optimizing a model, we pre-define a learning rate, denoted as η (eta) [5]. This learning rate scales the adjustment of the weights during each iteration so that each incremental update isn't too much or too little of a change, so that each iteration is a smooth and gradual step towards the most optimal model of the network. To get our scaled gradient, we therefore multiply η by ∇W, being the result of our calculations above:

$$\eta \cdot \nabla W = \eta \cdot \left((\mathbf{a} - \mathbf{y}) \cdot \mathbf{a} \cdot (\mathbf{1} - \mathbf{a}) \cdot \mathbf{x}^T\right) \tag{7.11}$$

6. Obtaining Our Updated Weight Matrix

We now subtract our scaled gradient $\eta \cdot \nabla W$ from the matrix representing the current state of the weights W_{old}, to update and obtain our new weight matrix. This subtraction means that we move the weights (and therefore the performance of the model) in the direction that reduces the scale of error, since ∇W represents the direction of increasing the error. Subtracting this means that

we move away from the direction of error in terms of scale. This subtraction gradually adjusts the weights to minimize the loss. Over time, the weights gradually move toward values resulting in fewer errors occurring, meaning that the functionality of the model improves over time by iterating over this process. Putting this all together, we can see that to obtain our new weights we can define the formula as:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \left((\mathbf{a} - \mathbf{y}) \cdot \mathbf{a} \cdot (\mathbf{1} - \mathbf{a}) \cdot \mathbf{x}^T\right) \tag{7.12}$$

and seeing as we know that the result of multiplying the derivatives in the correct way, we obtain ∇W, we can instead claim:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \nabla W \tag{7.13}$$

We can now see that by working through these steps from known principles, we have arrived at the formula:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \nabla W \tag{7.13}$$

proving that:

$$\nabla W = (\mathbf{a} - \mathbf{y}) \cdot \mathbf{a} \cdot (\mathbf{1} - \mathbf{a}) \cdot \mathbf{x}^T \tag{7.14}$$

as a result of multiplying the derivatives of the loss function with respect to the output, the activation function with respect to the weighted sum, and the weighted sum with respect to the weights [12].

This demonstrates how backpropagation efficiently propagates error through the network, and how this formula successfully updates the weights to minimize the loss and therefore the error of the network.

7.5 Activation Functions and Neurons

Neural networks are constructed using layers of interconnected neurons. Each neuron performs a weighted sum of inputs, followed by an activation function to bring non-linearity to the system. Usually, data is passed through the network from the input layer (where data is received) to the output layer (where the use of the model can be extracted).

At each layer of the network of neurons, sets of matrix multiplications are constantly being performed between the input vector and the weight matrix (followed by the addition of a bias vector), giving the network flexibility to model more complex patterns. These operations give the network the flexibility to model more complex patterns, preventing it from oversimplifying the data and improving its ability to capture intricate relationships. Although we've seen forward and backward error propagation, it's important that we also introduce error correction into our calculations so our model can move closer to being as accurate as possible.

Non-linear Importance

Before walking through the specific steps of network training, it's important to understand why non-linearity is an essential feature to strive for within neural networks. Non-linear activation functions are what enable the network to be able to learn and model complex relationships and patterns in its input data, preventing it from being reduced to a simple linear system. If this was permitted to happen, it would render the model being trained ill-suited to many situations.

The Sigmoid Activation Function

Let's demonstrate how we introduce non-linearity to our model. To introduce the complexity required that allows the network to learn and grow, we apply a non-linear activation function of our choosing at each layer during forward propagation. The sigmoid activation function is a popular choice that we're going to explore in the context of our proof example above. The sigmoid function is defined as:

$$\sigma(\mathbf{z}) = \frac{\mathbf{1}}{\mathbf{1} + e^{-\mathbf{z}}} \tag{7.5}$$

The terms of this function are:

- z is the input to the sigmoid function, which in our case is the result of our formula from forward propagation $\mathbf{z} = W(\mathbf{x} + \mathbf{b})$.
- e is Euler's number (approx. 2.718), a constant used in mathematical functions for modelling exponential growth or decay.
- $e^{-\mathbf{z}}$ is the exponential decay term, ensuring that the sigmoid function outputs values between 0 and 1, framing in a way that is useful for activating the neurons.
- $1 + e^{-\mathbf{z}}$ controls how the input $\mathbf{z}$ scales, ensuring the function is bounded between 0 and 1.

The whole function itself frames the input $\mathbf{z}$ into the range between 0 and 1, while introducing non-linearity into the network.

In the context of our proof example above, let's apply the sigmoid activation function to each layer's output and observe the change to the resulting formula:

The First Layer:

$$\mathbf{z}_1 = \sigma(W_1\mathbf{x} + \mathbf{b}_1) \tag{7.15}$$

The Second Layer:

$$\mathbf{z}_2 = \sigma(W_2\mathbf{z}_1 + \mathbf{b}_2) \tag{7.16}$$

The Third Layer:

$$\mathbf{z}_3 = \sigma(W_3\mathbf{z}_2 + \mathbf{b}_3) \tag{7.17}$$

After combining these layers, simulating our learning process as we did in our proof example, we observe:

$$z_{\mathbf{3}} = \sigma(W_{\mathbf{3}}\sigma(W_{\mathbf{2}}\sigma(W_{\mathbf{1}}\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) + \mathbf{b}_3) \tag{7.18}$$

This is the non-linear equivalent of the previous resulting linear equation. Each layer's output is transformed by the sigmoid function, introducing non-linearity at each stage. This nested function application allows the network to model more complex, non-linear patterns in the data. Instead of having a linear combination of weights and biases, the network layers have non-linear transformations at each step.

As we've seen, the sigmoid function limits the range of the output at each layer to between 0 and 1, allowing the network to capture intricate relationships in the data, highlighting the importance of using activation functions within neural networks [18].

Activation functions like σ are typically non-linear. They are also usually monotonic, meaning that they are always either increasing or decreasing. Additionally, they are also usually differentiable, meaning they have a precisely defined derivative for every point in their domain, smoothing the backpropagation process. For the network to learn complex relationships and make the gradient-based optimization work effectively, these properties are essential for an activation function to contain to implement into the network.

Sigmoid Visualization

In Figure 7.5 below, we can see a visualization of a sigmoid function, where the inputs are being transformed into a value between a 0 or a 1. It's worth noting that these input values can also be negative, as the input values to generate this visualization range between -8 and 8.

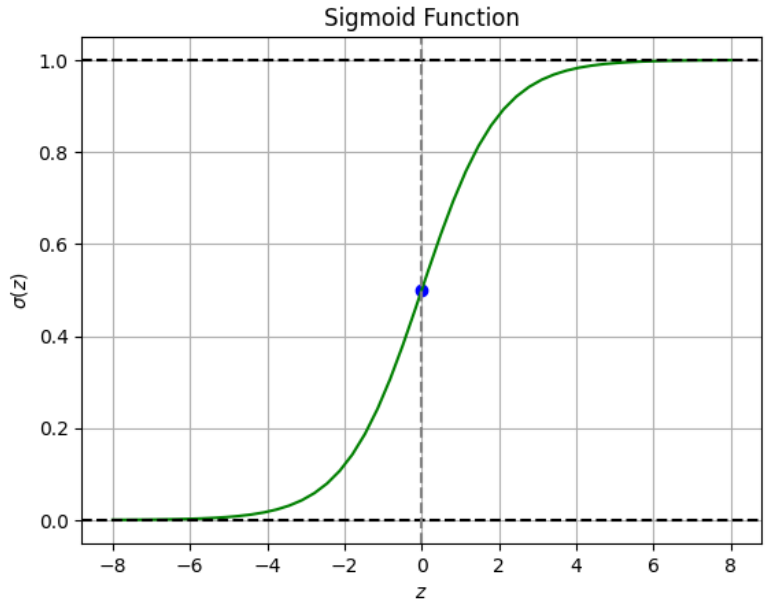


Figure 7.5: Sigmoid Function Visualization. Created using `matplotlib` in Python.

As we can see, the input values closer to -8 (the minimum value of our input set) produce outputs that approach 0, while inputs near 8 (the maximum value of our input set) move the output closer to 1. The blue dot marks an input of 0 into the function, where the sigmoid function outputs 0.5. This visualization demonstrates how the sigmoid function smoothly transforms inputs into values between 0 and 1, with small changes in input close to 0 leading to small changes in output. We can also see that the function curves flatten at the extremes (approaching the maximum values of -8 and 8 in this case), meaning further changes in input cause minimal changes in output.

We can see that as the input values approach the maximum or minimum, they approach 1 or 0 more slowly respectively, leading to the flattening of the curves. This is important to notice because it helps stabilize the learning process by compressing the output. This helps to mitigate issues like exploding or vanishing values too, but it can also lead to smaller gradients, which has the potential to slow down learning in deeper networks.

This function also supersedes what is known as on/off firing neurons, instead replacing this behaviour with smooth, continuous outputs. Instead of more traditional functions that sharply change between 0 and 1, sigmoid functions permit more steady changes, allowing a network to understand subtler patterns. In addition, the introduced differentiability across all input values by the function is very important for backpropagation, without which the network couldn't compute gradients and adjust weights as effectively.

Implementation Overview Including Error Correction

We're going to give a general overview of the iterative steps of training a neural network in the context of matrix operations. Keep in mind that more steps in between these steps are frequently used to further optimize a network for a specific purpose, and that this process (steps 2, 3, and 4 specifically) is run millions of times on a regular basis.

1. **Initialization**
   Our simple neural network will consist of an input layer, some hidden layers, and an output layer. First, the initial weight matrices and bias vectors are initialized with small random values. Seeing as we don't know what they should be, we'll make them relatively small, so they have room to move in the direction the data takes them after being iterated by the network.
2. **Forward Propagation**
   We multiply the input vector by the existing weight matrix, add the bias vector, then pass the result through an activation function to get the output of that layer.
3. **Error Calculation**
   There will always be some form of error when performing these enough times, so we need to calculate the error delta between the predicted output and the actual label (what the model thought would appear or occur). We use the target output y to represent the accurate output we're wanting out of our model. Using that error calculation, we compute the gradients of the error in observation of the weights, using the chain rule and matrix derivatives. This adjusts the weights and biases, which is then used back in step two with more accurate and "closer-to-the-truth" configurations that will allow the model to improve its ability of prediction.
4. **Backpropagation and Weight Updates**
   Backpropagation occurs by propagating the error backwards in the reverse order, back through the network. The weights are adjusted to reduce the error for future predictions.

Implementation Steps with Error Calculation

Now that we understand the three iterative steps networks take to improve, let's have a closer look at what it actually takes from a more mathematical perspective:

Implementation Overview Including Error Correction

We're going to give a general overview of the iterative steps of training a neural network in the context of matrix operations. Keep in mind that additional optimization steps are often used to tailor a network for specific tasks, and that steps 2, 3, and 4 are typically repeated millions of times during training.

1. Initialization

Our simple neural network consists of an input layer, some hidden layers, and an output layer. First, the initial weight matrices and bias vectors are initialized with small random values. Since we don't yet know their ideal values, we make them relatively small so they can adjust in response to the data as the network iterates.

2. Forward Propagation

We multiply the input vector by the existing weight matrix, add the bias vector, and then pass the result through an activation function to generate the output of that layer.

3. Error Calculation

There will always be some level of error during training, so we calculate the error delta between the predicted output and the actual target output y. Using this error calculation, we compute the gradients of the error with respect to the weights, applying the chain rule and matrix derivatives. These gradients guide the adjustment of weights and biases, refining the model to make more accurate predictions in subsequent iterations.

4. Backpropagation and Weight Updates

Backpropagation propagates the error backward through the network, adjusting the weights in a way that reduces the error for future predictions. This process iteratively improves the model's performance.

Using the implementation steps we listed above, let us work through an example to find the updated weight matrix after processing has completed one iteration.

First, we perform forward propagation to calculate the output before applying the activation function, just as we did in the previous section. Then calculate the error using MSE, (as the chosen loss function for this example) comparing it to our target output. Finally, we update the weight matrix using backpropagation as we did in the previous section to obtain our result.

Example 7.2: Network calculation with all steps

Step 1: Forward Propagation

We first need to perform forward propagation on a neural network. Given:

- Weight matrix of $\begin{pmatrix} 0.6 & 1.2 \\ 0.8 & 0.4 \end{pmatrix}$
- Input vector of $\mathbf{x} = (3, 5)$
- Bias vector of $\mathbf{b} = (0.3, 0.7)$
- Learning rate η of 0.02

To find $\mathbf{z}$, being the linear combination of these terms, we use:

$$\mathbf{z} = W(\mathbf{x} + \mathbf{b})$$

First, we use matrix multiplication to multiply the existing weight matrix W, and the input vector $\mathbf{x}$:

$$W \cdot x = \begin{pmatrix} 0.6 & 1.2 \\ 0.8 & 0.4 \end{pmatrix} \cdot \begin{pmatrix} 3 \\ 5 \end{pmatrix} = \begin{pmatrix} 7.8 \\ 4.4 \end{pmatrix}$$

Then we add the bias vector:

$$(W \cdot \mathbf{x}) + \mathbf{b} = \begin{pmatrix} 7.8 \\ 4.4 \end{pmatrix} + \begin{pmatrix} 0.3 \\ 0.7 \end{pmatrix} = \begin{pmatrix} 8.1 \\ 5.1 \end{pmatrix}$$

Therefore, $\mathbf{z} = \begin{pmatrix} 8.1 \\ 5.1 \end{pmatrix}$.

Step 2: Error Calculation

Next, we calculate the Mean Squared Error (MSE) between our calculated output z and the target output y:

$$\text{MSE} = \frac{1}{2}\sum(z_i - y_i)^2$$

Substituting our values:

$$\text{MSE} = \frac{1}{2}\left((8.1-8)^2 + (5.1-5)^2\right) = \frac{1}{2}\left(0.01 + 0.01\right) = 0.01$$

Therefore, our total error value is 0.01.

Step 3: Backpropagation

a) Compute error gradient matrix (using partial derivatives) with respect to z_i (each element of the result of forward propagation) and the target output.
Now that we have our error value, we can approximate the gradient matrix, which will be used in backpropagation to obtain our updated weight matrix. Firstly, we need to calculate the partial derivative [3] of the MSE ($\frac{\partial MSE}{\partial z_i}$) with respect to the output z_i, being each element from the result of forward propagation as the predicted output after passing through each layer of our neural network. This is represented (and then calculated) as the formula:

$$\frac{\partial MSE}{\partial z_i} = z_i - y_i \tag{7.19}$$

where y_i is each element of the target output (given to us at the start of this problem). We calculate this for each element of $\mathbf{z}$ from step 1 being $(8.1, 5.1)$ and our target output $\mathbf{y}$ being $(8, 5)$:

$$\frac{\partial MSE}{\partial z_1} = z_1 - y_1 = 8.1 - 8 = 0.1$$

$$\frac{\partial MSE}{\partial z_2} = z_2 - y_2 = 5.1 - 5 = 0.1$$

Therefore, our error gradient for the output is $(0.1, 0.1)$.
b) Calculate the gradient matrix by multiplying that error gradient by the input vector.
From this point, we calculate our gradient matrix by multiplying the input vector $(3, 5)$ with our newly found error gradient $(0.1, 0.1)$. In formulaic notation, this is found by:

$$\nabla W = \begin{pmatrix}3\\5\end{pmatrix}\begin{pmatrix}0.1 & 0.1\end{pmatrix} = \begin{pmatrix}0.3 & 0.3\\0.5 & 0.5\end{pmatrix}$$

c) Calculate updated weight values after performing backpropagation.
Now we use the formula we found in the previous section to perform backpropagation using our old weight matrix, our known learning rate, and the gradient matrix we've just found:

$$W_{\text{new}} = W_{\text{old}} - \eta \cdot \nabla W \tag{7.13}$$

so

$$W_{new} = \begin{pmatrix} 0.6 & 1.2 \\ 0.8 & 0.4 \end{pmatrix} - 0.02 \begin{pmatrix} 0.3 & 0.3 \\ 0.5 & 0.5 \end{pmatrix} = \begin{pmatrix} 0.594 & 1.194 \\ 0.79 & 0.39 \end{pmatrix}$$

and we've arrived at the result of our entire process, with our new updated weight matrix of

$$\begin{pmatrix} 0.594 & 1.194 \\ 0.79 & 0.39 \end{pmatrix}.$$

This represents our model learning and coming closer to being more accurate, and many more iterations of this process will further refine this weight matrix.

Here we've explored how matrix operations like forward propagation and backpropagation are used in adjusting the weights within a neural network to dramatically improve its accuracy. By calculating the output through matrix multiplication, determining the error using the Mean Squared Error (MSE), and then updating the weights using backpropagation, we can iteratively refine the model until we run out of data, compute, or the model is good enough.

In our example, the original weight matrix of

$$\begin{pmatrix} 0.6 & 1.2 \\ 0.8 & 0.4 \end{pmatrix}$$

becoming

$$\begin{pmatrix} 0.594 & 1.194 \\ 0.79 & 0.39 \end{pmatrix}$$

after one iteration of forward propagation, error calculation, and backpropagation demonstrates a change that represents the network adjusting its parameters (weights) to reduce the error between its predicted output and the target output. Each weight shifts slightly as the model "learns" from its mistakes. As a result, the model becomes more accurate in its predictions, and with more iterations, it will continue to improve. This process demonstrates how neural networks iteratively refine their weights to minimize prediction errors and improve overall performance. This doesn't necessarily need to come closer to 0 (unlike what we'll do in our next section) but instead become more finely tuned to the task and data the network is being fed. The actual implementations of this are vastly more numerous, but understanding this is key to understanding how a machine can learn, using linear algebra.

With the gradients calculated, the next logical step is understanding how these weight updates can be controlled, leading us into the concept of regularization, which plays a crucial role in preventing overfitting.

7.6 Mathematical Foundations of gradient descent

In this section we take a more detailed look at the role of gradient descent in backpropogation. We first recall some important results from calculus.

Chain Rule and Partial Derivatives

To begin, we will conceptualize what is referred to as the **partial derivative** [16].

Definition 7.1: Partial Derivative

Given a multivariate function $f(x_1, x_2, ..., x_n)$. For each $i \in [1, n]$, we define the *partial derivative of f with respect to x_i* by:

$$\frac{\partial J}{\partial x_i} = \lim_{h \to 0} \left[\frac{f(x_1, ..., x_{i-1}, x_i + h, x_{i+1}, ..., x_n) - f(x_1, ..., x_i, ..., x_n)}{h} \right]$$

In other words, the partial derivative of f with respect to x_i is computed by taking the *first derivative* of f with respect to x_i, while remaining variables are taken as constants.

However, it is very difficult to calculate the limit according to the above definition. Instead, we use a number of well-established differentiation rules along with the **chain rule** to compute the partial derivative.

Definition 7.2: Chain Rule

Given 3 variables a, b and c. The chain rule is expressed:

$$\frac{db}{da} = \frac{db}{dc} \cdot \frac{dc}{da}$$

where $\frac{db}{da}$ is the derivative of b with respect to a, and similarly for $\frac{db}{dc}$ and $\frac{dc}{da}$.

An important note is that this notation $\frac{dy}{dx}$ is a total derivative, not partial derivative.
To understand these concepts, we will look at the following example:

Example 7.3: Partial derivative calculations

Given the function:

$$f(x, y) = x^2 y + 3xy^2 \tag{7.20}$$

a) Find $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ at point $(x, y) = (3, 5)$

b) Suppose $x(t) = t^2$ and $y(t) = t + 2$, find $\frac{\partial f}{\partial t}$ at $t = 2$

Solution.
a) From equation (7.20), we have:

$$\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial (x^2 y)}{\partial x} + \frac{\partial (3xy^2)}{\partial x} \\
&= y \frac{\partial x^2}{\partial x} + 3y^2 \frac{\partial x}{\partial x} \ (y \text{ is considered as constant}) \\
&= 2xy + 3y^2
\end{aligned}$$

and

$$\begin{aligned}\frac{\partial f}{\partial y} &= \frac{\partial (x^2 y)}{\partial y} + \frac{\partial (3xy^2)}{\partial y} \\ &= x^2 \frac{\partial y}{\partial y} + 3x \frac{\partial y^2}{\partial y} \ (x \text{ is considered as constant}) \\ &= x^2 + 6xy\end{aligned}$$

Therefore, at point $(x, y) = (3, 5)$

$$\frac{\partial f}{\partial x} = 2xy + 3y^2 = 2 \times 3 \times 5 + 3 \times 5^2 = 105$$
$$\frac{\partial f}{\partial y} = x^2 + 6xy = 3^2 + 6 \times 3 \times 5 = 99$$

b) Note that f is a multivariate function with 2 variables x and y. Thus, applying the intuition that the total change in f equals the sum of total changes due to each of the variables, we have:

$$df = \frac{\partial f}{\partial x} \cdot dx + \frac{\partial f}{\partial y} \cdot dy$$

Now, applying Definition 7.2, we get:

$$dx = \frac{dx}{dt} \cdot dt \text{ and } dy = \frac{dy}{dt} \cdot dt$$

Therefore, we have the expression:

$$\begin{aligned}df &= \frac{\partial f}{\partial x} \cdot \left(\frac{dx}{dt} \cdot dt \right) + \frac{\partial f}{\partial y} \cdot \left(\frac{dy}{dt} \cdot dt \right) \\ &= \left(\frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt} \right) \cdot dt\end{aligned}$$

Continuing to apply the chain rule, we get the equation:

$$\frac{df}{dt} = \frac{\partial f}{\partial x} \cdot \frac{dx}{dt} + \frac{\partial f}{\partial y} \cdot \frac{dy}{dt} \tag{7.21}$$

Now, plugging $x(t) = t^2$, $y(t) = t + 2$ and $t = 2$ into Equation (7.21), we have:

$$\begin{aligned}\frac{df}{dt} &= (2xy + 3y^2) \cdot \frac{dt^2}{dt} + (x^2 + 6xy) \cdot \frac{d(t+2)}{dt} \\ &= 2t(2xy + 3y^2) + x^2 + 6xy \\ &= 2 \times 2(2 \times 2^2 \times (2+2) + 3 \times (2+2)^2) + (2^2)^2 + 6 \times 2^2 \times (2+2) \\ &= 320 + 16 + 96 = 432\end{aligned}$$

Gradient Descent

The idea of gradient descent is to move in the direction that minimizes the approximation of the objective by moving a scale of η in the opposite direction with the descent $f'(x_i)$ [8]:

$$x_{i+1} = x_i - \eta f'(x_i),\ \forall i \tag{7.22}$$

where x_0 is the initial point and η is called the **learning rate**. But in reality, we deal with a dataset of points, not just a single point. Therefore, we need to express equation (11.1) in matrix form.

Suppose we have an $m \times n$ matrix X with m records, each which has n columns whose values denote its features. Let W be the column vector of parameters (weights) and $\mathbf{y}$ be the output column vector consists of target values.

For a linear regression model, often we will use the **Mean Squared Error** objective/loss function, that is of the form:

$$J(W) = \frac{1}{2m}(XW - \mathbf{y})^T(XW - \mathbf{y})$$

Now, we will calculate the gradient $\dfrac{\partial J}{\partial W}$.

Firstly, we need to know that $g(f(W))$ is a single variable function (the only variable here is W). This means the partial derivative of $g(f(W))$ with respect to W is the the same as the derivative of $g(f(W))$ with respect to W. Therefore, we can apply the chain rule in Definition 7.2 in calculating the partial derivative $\dfrac{\partial g(f(W))}{\partial W}$.

We let $\mathbf{z} = f(W) = XW - \mathbf{y}$ and $g(\mathbf{z}) = \mathbf{z}^T\mathbf{z}$. Applying the chain rule for $b = g(f(W)) = (f \circ g)(W)$, $a = W$ and $c = \mathbf{z}$ (consider W as a variable), we will have:

$$\begin{aligned}
\frac{\partial g(f(W))}{\partial W} &= \frac{\partial g(f(W))}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial a} \\
&= \frac{\partial g(f(W))}{\partial f(W)} \cdot \frac{\partial f(W)}{\partial W} \\
&= \frac{\partial g}{\partial \mathbf{z}} \cdot \frac{\partial f}{\partial W}
\end{aligned}$$

Since $\dfrac{\partial g}{\partial \mathbf{z}} = \dfrac{\partial(\mathbf{z}^T\mathbf{z})}{\partial \mathbf{z}} = 2\mathbf{z}$ and $\dfrac{\partial f}{\partial W} = \dfrac{\partial(XW - \mathbf{y})}{\partial W} = X$, we have the gradient for $J(W)$ as follows:

$$\frac{\partial J(W)}{\partial W} = \frac{1}{2m}\frac{\partial g(f(W))}{\partial W} = \frac{1}{2m}X^T 2\mathbf{z} = \frac{1}{m}X^T(XW - \mathbf{y})$$

Applying equation (1), we obtain the gradient descent algorithm in matrix form:

$$W \longleftarrow W - \eta\frac{\partial J(W)}{\partial W} = W - \frac{\eta}{m}X^T(XW - \mathbf{y})$$

Summarising what we have above, we have the following process, which we refer to as the *Gradient Descent Algorithm* [6].

Definition 7.3: Gradient Descent Algorithm

Given a dataset X of size $m \times n$ consists of n features and m records, let W denote the $n \times 1$ matrix of parameters and $\mathbf{y}$ the output column vector of target values. Assuming that we use Mean Square Error as the objective/loss function, then for each iteration, the parameter vector is updated by:

$$W \longleftarrow W - \frac{\eta}{m} X^T (XW - \mathbf{y})$$

where η is the learning rate.

In the update rule above, the term $X^T(XW - \mathbf{y})$ is the gradient of the MSE loss function, and this term is easily calculated using matrix multiplication and transposition. Furthermore, we know that the MSE function is quadratic, so its gradient is linear in terms of the parameters W. Thus, for each update iteration, we only need to evaluate using a predetermined formula, making Gradient Descent straightforward and useful especially in linear regression problems.

Stochastic Gradient Descent

Sometimes we have to solve the problems involving very large datasets. This can require large calculations that may be difficult with limited computation power, however in some cases a data sample can be used to estimate the gradient well for the entire dataset. This leads to a concept called *stochastic gradient descent*.

In stochastic gradient descent, we will take out a sample of records S consisting of k records X_1, X_2, ..., X_k where S contains the indices of the chosen records. We set up our objective/loss function as a MSE function dealing with only the records in S instead of the whole dataset:

$$J(W) = \frac{1}{2|S|}(X_0 W - \mathbf{y_0})^T (X_0 W - \mathbf{y_0})$$

where $|S|$ is the number of records in the sample, X_0 is the $k \times n$ matrix consisting of the sample records and $\mathbf{y_0}$ is the $k \times 1$ output column vector.
Thus, the update rule for the weight vector W will be:

$$W \longleftarrow W - \frac{\eta}{|S|} X_0^T (X_0 W - \mathbf{y_0})$$

So the only difference between Stochastic Gradient Descent from Gradient Descent is that we perform the calculation using a sample of the dataset instead of the whole dataset. The parameter update will now rely on a subset of the dataset for each iteration.

The below example will help illustrate the Gradient Descent process.

Example 7.4: Gradient descent calculations

Consider a simple 2-D data set with only 4 data points of the form (x_1, x_2), and each data point has a label value y assigned. We employ a linear regression model: $y = w_1 x_1 + w_2 x_2 + b$ where w_1, w_2 are the weights and b is a real constant.

The dataset of $[x_1,\ x_2,\ y]$ is given by:

x_1	x_2	y
1	1	6
2	0	7
1	3	8
2	2	9

Now, we will perform the following steps:

Parameter Initialization

- Parameter matrix $W = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$
- Dataset $X = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 1 & 3 & 1 \\ 2 & 2 & 1 \end{bmatrix}$
- Desired output $\mathbf{y} = \begin{bmatrix} 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}$
- Learning rate $\eta = 0.1$.

Updating parameters

Since we have a total of $m = 4$ records in the sample dataset, the MSE function is calculated by:

$$XW - \mathbf{y} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 1 \\ 1 & 3 & 1 \\ 2 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} 6 \\ 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} -3 \\ -4 \\ -3 \\ -4 \end{bmatrix}$$

$$J(W) = \frac{1}{2m}(XW - \mathbf{y})^T(XW - \mathbf{y})$$

$$= \frac{1}{8}\begin{bmatrix} -3 & -4 & -3 & -4 \end{bmatrix} \begin{bmatrix} -3 \\ -4 \\ -3 \\ -4 \end{bmatrix} = \frac{25}{4}$$

Hence, we update the parameters:

$$W = W - \frac{\eta}{m} X^T(XW - \mathbf{y})$$

$$= \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \frac{0.1}{4} \begin{bmatrix} 1 & 2 & 1 & 2 \\ 1 & 0 & 3 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ -4 \\ -3 \\ -4 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} -0.55 \\ -0.5 \\ -0.35 \end{bmatrix} = \begin{bmatrix} 1.55 \\ 1.5 \\ 1.35 \end{bmatrix}$$

Therefore, after the first update, $w_1 = 1.55$, $w_2 = 1.5$ and $b = 1.35$.

This is just a simple implementation of the Gradient Descent process (in fact, the real solution to the above example is $w_1 = 2$, $w_2 = 1$ and $b = 3$). In real scenarios, we have to run more epochs/iterations and handle a larger dataset to achieve the best learning results on the training dataset.

7.7 Conclusions

This article has demonstrated the foundational role linear algebra and matrix operations play in creating and training neural networks. Core processes such as forward propagation, error correction, backpropagation, and weight updates rely on matrix operations to drive the iterative learning process. By mastering efficient matrix calculations, neural networks can refine predictions, adjust weights, and improve accuracy over many iterations.

Non-linear activation functions and other techniques reinforce the flexibility and power of neural networks, preventing them from collapsing into simple linear systems and enabling the approximation of complex functions.

The concept of partial derivatives is fundamental in multivariate calculus. A partial derivative measures how a function $f(x_1, x_2, ..., x_n)$ changes with respect to one variable (e.g. x) while keeping all other variables constant. This is useful for analyzing functions of multiple variables, especially in machine learning and deep learning. The chain rule builds on this concept and explains how to calculate the total derivative of a function when its variables themselves depend on another variable, such as time t.

Gradient descent, though foundational, is often outpaced by stochastic gradient descent in terms of efficiency and speed, particularly under conditions involving large datasets. Stochastic gradient descent's improved performance, facilitated by linear algebraic optimizations, offers a more dynamic approach to weight adjustments, which is crucial in real-time data processing.

By deeply understanding these advanced techniques, designers can unlock the full potential of neural networks, making these mathematical frameworks indispensable in AI and machine learning. These foundations enable modern applications like image recognition and natural language processing, and their use will only expand in scope as the technology advances. The future of this field is thought to be both vast and world-changing, depending on the speed and success of its implementations.

Future research should continue to refine these algorithms, exploring more hybrid techniques that could offer even greater efficiency and flexibility in handling diverse and evolving data landscapes.

Context

This article was adapted from reports submitted for SIT292.

About the Authors

Chris is a first-year student at Deakin University pursuing Bachelor of Computer Science. He is interested in many fields of research such as optimization, image processing and speech recognition. Chris is very excited with the given opportunities to explore and connect with different researchers to have more understanding about these vast fields.

Ari Robin is a Computer Science student at Deakin University in Melbourne, Australia, with interest in the development and applications of Quantum Computing.

Acknowledgements

(Chris) I would like to express my special thanks and gratitude to Simon James, who taught me in Linear Algebra for Data Analysis, has assisted me and gave me really detailed feedback on my report. I know being a part of a joint article is a rare event so I really appreciate this opportunity you have given to me.

(Ari) I would like to thank Dr Simon James for his inspiration, assistance and tireless efforts both in this paper, the furthering of my interests and beyond.

References

All diagrams were created by the authors using Tikz [17], while graphical plots were created using Python [11].

[1] Backpropagation in neural network. *GeeksforGeeks*, 2024. https://www.geeksforgeeks.org/backpropagation-in-neural-network/.

[2] Linear algebra operations for machine learning. *GeeksforGeeks*, 2024. https://www.geeksforgeeks.org/ml-linear-algebra-operations/.

[3] Partial derivatives in machine learning. *GeeksforGeeks*, 2024. https://www.geeksforgeeks.org/partial-derivatives-in-machine-learning/.

[4] What is forward propagation in neural networks? *GeeksforGeeks*, 2024. https://www.geeksforgeeks.org/what-is-forward-propagation-in-neural-networks/.

[5] Great Learning Editorial Team [Online Article]. Understanding learning rate in machine learning. *Great Learning*, 2024. https://www.mygreatlearning.com/blog/understanding-learning-rate-in-machine-learning/.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2006.

[7] V. Divakar. Understanding backpropagation (Online Article), 2018. https://blog.quantinsti.com/backpropagation/.

[8] G. Farina. Gradient descent [Lecture Notes]. *Massachusetts Institute of Technology (MIT)*, 2024.

[9] IBM. What is a neural network? *IBM Think - Tech news, education and events*, 2021. https://www.ibm.com/topics/neural-networks.

[10] J. Osajima. The math behind neural networks - forward propagation (Online Article). 2018. https://www.jasonosajima.com/forwardprop.

[11] Python Core Developers. *Python: A dynamic, open source programming language*. Python Software Foundation, 2022. https://www.python.org/downloads/release/python-3110/.

[12] S. Raja. A derivation of backpropagation in matrix form (Online Article). 2016. https://sudeepraja.github.io/Neural/.

[13] W. Rowe. Mean square error & R2 score clearly explained (Online Article). *BMC Blogs*, 2018. https://www.bmc.com/blogs/mean-squared-error-r2-and-variance-in-regression-analysis/.

[14] B. Scarff. Understanding backpropagation algorithm. *Towards Data Science*, 2019. https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd.

[15] S. Siddant. The mathematics of neural networks — a complete example. *Medium*, 2024. https://medium.com/@SSiddhant/the-mathematics-of-neural-networks-a-complete-example-65f2b12cdea2.

[16] G. Strang and E. J. Herman. Partial derivatives. In *Calculus Volume 3*. OpenStax, 2016. https://openstax.org/books/calculus-volume-3/pages/4-3-partial-derivatives.

[17] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[18] N. Topper. Sigmoid activation function: An introduction (Online Article). *BuiltIn*, 2023. https://builtin.com/machine-learning/sigmoid-activation-function.

[19] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. Forward propagation, backward propagation, and computational graphs. In *Dive into Deep Learning*. Cambridge University Press, 2023.

8

Optimization in machine learning and deep learning

Tuan Thanh (Chris) Lu

Abstract

This study investigates the application of linear algebra to optimization algorithms such as AdaGrad, RMSProp and Adam. Results reveal that these adaptive methods offer significant improvements in dynamic environments by optimizing learning rates and mitigating issues such as the vanishing gradient problem. The findings underscore the critical role of sophisticated mathematical operations (including linear algebra) in advancing algorithmic performance and suggest directions for future research in hybrid optimization techniques.

8.1 Introduction

We know that in Machine Learning and Deep Learning, especially in Deep Neural Network training, implementing only linear activation functions will make our model sensitive to strange patterns/records. Furthermore, the output of this network will be linear with respect to the inputs, which is not realistic when we deal with real-world datasets. It may also provide outputs that are totally different from the expected values, leading to ineffective learning. Therefore, we need optimization algorithms that deal with all aspects of the model to minimize error to the training dataset. This document will give a detailed look at different optimization algorithms and how these can be used in different circumstances. In Section 1.2, we will introduce the notations we will use in this document. In Sections 1.3, 1.4 and 1.5, we will respectively dive into 3 of the most popular optimization algorithms: AdaGrad, RMSProp and Adam. In Section 1.6, we will compare and visualize the results of the 3 algorithms. Finally in Section 1.7, we will summarize the key findings and the role of linear algebra in these algorithms. For a detailed foundation of neural networks, including their definition and forward propagation algorithm, readers are encouraged to refer to the joint article *Linear Algebra in Neural Network Training*, which provides essential background material to better follow this document.

8.2 Preliminaries

This section presents the required notation and background. We will use capital letters, X, W to denote matrices, and bold lower case $\mathbf{v}$, $\mathbf{y}$ to denote vectors or $\mathbf{u}_i$ for column vectors associated with the i-th column of a matrix. Moreover, we use $\mathbf{u} \odot \mathbf{v}$ (or $u \odot v$) to define the Hadamard

product of 2 vectors/column matrices of the same size. It is calculated as follows:

$$\mathbf{x} \odot \mathbf{y} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \odot \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 \\ x_2 y_2 \\ \vdots \\ x_n y_n \end{bmatrix}$$

Below is a list of some the common notation we employ for easy reference:

η: learning rate

n: number of parameters that need to be optimized

$iter$: number of iterations/epochs

$J(\mathbf{x})$: stochastic objective function with parameters $\mathbf{x}$. An objective function is the real-valued function whose value is to be either minimized or maximized over the set of feasible alternatives

w_i : i-th parameter

$\frac{\partial J}{\partial w_i}$: partial derivative of J with respect to the parameter w_i

G_i : sum of squared gradients for the i-th parameter

F_i : exponentially smoothed values for the i-th parameter

λ: decay factor

λ_1, λ_2: decay factor for F_i and G_i, respectively

8.3 AdaGrad

AdaGrad, or the Adaptive Gradient algorithm, is a family of sub-gradient algorithms for stochastic optimization [4, 5]. The purpose of AdaGrad is to minimize the value of a stochastic objective function with respect to a set of parameters. The AdaGrad method involves vector and matrix operations such as scalar multiplication and addition to adjust the model parameters iteratively in order to minimize an objective function. Additionally, the computation of parameter updates involves gradients, the Hadamard product and diagonal matrices accumulatiing the squared gradients, which are linear algebra constructs that allow efficient scaling of the learning rate for each parameter.
Now, we will explore the steps involved in the AdaGrad algorithm:
+ Step 1: **Initialization**

- η : learning rate
- $J(\mathbf{x})$: stochastic objective function
- $W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$: initial parameter vector
- $G_i = 0$, $\forall i$: initial sum of squared gradients for the i-th parameter.

For each iteration, we do the following for the i-th parameter:

+ Step 2: **Accumulate square gradients**

$$G_i \longleftarrow G_i + \left(\frac{\partial J}{\partial w_i}\right)^2, \forall i$$

+ Step 3: **Update parameters**

$$w_i \longleftarrow w_i - \frac{\eta}{\sqrt{G_i}} \left(\frac{\partial J}{\partial w_i} \right), \forall i$$

+ Step 4: Repeat Step 2 and Step 3 until one of the following conditions is satisfied:

- Maximum number of iterations reached.
- The difference between the values of $i+1$-th iteration and i-th iteration is smaller than threshold ε.

This is summarised formally as Algorithm 2.

Data: $\eta > 0$, $J(\mathbf{x})$, $n \geqslant 1$, $iter \geqslant 1$
Variable: W, G_i, t
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$;
$G_i \leftarrow 0$, $\forall i$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
  Accumulate square gradients (2);

$$G_i \leftarrow G_i + \left(\frac{\partial J}{\partial w_i} \right)^2, \forall i$$

  Update parameters (3);

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{G_i}} \left(\frac{\partial J}{\partial w_i} \right), \forall i$$

  $t \leftarrow t + 1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\left| w_i^{(t+1)} - w_i^{(t)} \right| < \varepsilon, \forall i$

Algorithm 2: AdaGrad algorithm in simplest form

Remark.

In the third step, G_i may equal 0; therefore, we can replace expression $\sqrt{G_i}$ by $\sqrt{G_i + \epsilon}$ where ϵ is a relatively small positive real number (often, $\epsilon = 10^{-8}$).

It is convenient to express the equations in matrix form. First, recall the definitions for η, $J(\mathbf{x})$ as in the above algorithm. We define:

- $W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$ as the $n \times 1$ matrix of parameters (weights) we need to update.

- $G = \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T$ as the $n \times 1$ matrix of sum of squared gradients. Initially, G will be equal to a $n \times 1$ zero matrix.
- $\dfrac{\partial J}{\partial W} = \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T$ as the partial derivative of J with respect to W.

Initially, G will be equal to a $n \times 1$ zero matrix. We will rewrite Step 2 in matrix form as follows:

$$G = \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T + \begin{bmatrix} \left(\dfrac{\partial J}{\partial w_1}\right)^2 & \left(\dfrac{\partial J}{\partial w_2}\right)^2 & ... & \left(\dfrac{\partial J}{\partial w_n}\right)^2 \end{bmatrix}^T$$

$$= G + \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T \odot \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T = G + \left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right)$$

Now, we will rewrite Step 3 in matrix form, but before this, we will introduce a new matrix:

$$S = diag(G) = \begin{bmatrix} G_1 & 0 & ... & 0 \\ 0 & G_2 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & G_n \end{bmatrix}$$

where S is a diagonal matrix consisting of elements in column matrix G. It is evident that:

$$D \times D = \begin{bmatrix} \sqrt{G_1} & 0 & ... & 0 \\ 0 & \sqrt{G_2} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & \sqrt{G_n} \end{bmatrix} \begin{bmatrix} \sqrt{G_1} & 0 & ... & 0 \\ 0 & \sqrt{G_2} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & \sqrt{G_n} \end{bmatrix} = \begin{bmatrix} G_1 & 0 & ... & 0 \\ 0 & G_2 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & G_n \end{bmatrix} = S$$

Therefore,

$$S^{-\frac{1}{2}} = \left(S^{\frac{1}{2}}\right)^{-1} = D^{-1} = \begin{bmatrix} \sqrt{G_1} & 0 & ... & 0 \\ 0 & \sqrt{G_2} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & \sqrt{G_n} \end{bmatrix}^{-1} = \begin{bmatrix} \dfrac{1}{\sqrt{G_1}} & 0 & ... & 0 \\ 0 & \dfrac{1}{\sqrt{G_2}} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & \dfrac{1}{\sqrt{G_n}} \end{bmatrix}$$

Now, we can rewrite Step 3:

$$W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T - \begin{bmatrix} \dfrac{\eta}{\sqrt{G_1}}\left(\dfrac{\partial J}{\partial w_1}\right) & \dfrac{\eta}{\sqrt{G_2}}\left(\dfrac{\partial J}{\partial w_2}\right) & ... & \dfrac{\eta}{\sqrt{G_n}}\left(\dfrac{\partial J}{\partial w_n}\right) \end{bmatrix}^T$$

$$= \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \dfrac{1}{\sqrt{G_1}} & 0 & ... & 0 \\ 0 & \dfrac{1}{\sqrt{G_2}} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & \dfrac{1}{\sqrt{G_n}} \end{bmatrix} \begin{bmatrix} \dfrac{\partial J}{\partial w_1} \\ \dfrac{\partial J}{\partial w_2} \\ \vdots \\ \dfrac{\partial J}{\partial w_n} \end{bmatrix} = W - \eta S^{-\frac{1}{2}} \left(\frac{\partial J}{\partial W}\right)$$

Summarising all above expressions, Algorithm 3 gives the AdaGrad algorithm in matrix form.

Data: $\eta > 0$, $J(x)$, $n \geqslant 1$, $iter \geqslant 1$
Variable: W, G, t, $\dfrac{\partial J}{\partial W}$, S
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$;
$G \leftarrow \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T = \mathbf{0}$;
$\dfrac{\partial J}{\partial W} \leftarrow \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
  Accumulate square gradients (2);

$$G \longleftarrow G + \left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W} \right)$$

  $S \leftarrow diag(G)$;
  Update parameters (3);

$$W \longleftarrow W - \eta S^{-\frac{1}{2}} \left(\frac{\partial J}{\partial W} \right)$$

  $t \leftarrow t + 1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\max\limits_i \left| w_i^{(t+1)} - w_i^{(t)} \right| < \varepsilon$

Algorithm 3: AdaGrad algorithm in matrix form

Remark.

To avoid ill-conditioning in Step 3 (i.e, to avoid one of the G_i values being equal to 0, we can replace $S = diag(G)$ by $S = diag(G + \epsilon)$ where ϵ is a relatively small positive number (default value of 10^{-8}) and:

$$S = diag(G + \epsilon) = \begin{bmatrix} G_1 + \epsilon & 0 & ... & 0 \\ 0 & G_2 + \epsilon & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & G_n + \epsilon \end{bmatrix}$$

The below linear regression Example 8.1 will help consolidate our understanding of the AdaGrad algorithm [4]:

Example 8.1

Suppose we are given a dataset consisting of random generated observations x, y that follow the linear relationship:

$$y = wx + b$$

where w is the weight and b is the bias we need to find. The first 4 observations are displayed in the below table:

x	y
1	3
2	5
3	7
4	9

We will use the mean squared error (MSE) loss function:

$$J(w, b) = \frac{1}{n}\sum_{i=1}^{n}(wx_i + b - y_i)^2$$

Now, we will follow the steps as shown in the algorithm:

Parameters Initialization

- Weight $W = \begin{bmatrix} w \\ b \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$
- Learning rate $\eta = 0.1$
- Sum of squared gradients for parameters w and b: $G = \begin{bmatrix} G_w \\ G_b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Updating parameters

Firstly, note that $n = 4$ since we have 4 records in the table. We compute the gradients:

- For w:

$$\begin{aligned}\frac{\partial J}{\partial w} &= \frac{2}{4}\sum_{i=1}^{4}(wx_i + b - y_i)x_i \\ &= \frac{2}{4}[(1+1-3)\times 1 + (2+1-5)\times 2 + (3+1-7)\times 3 + (4+1-9)\times 4] \\ &= -15\end{aligned}$$

- For b:

$$\begin{aligned}\frac{\partial J}{\partial b} &= \frac{2}{4}\sum_{i=1}^{4}(wx_i + b - y_i) \\ &= \frac{2}{4}[(1+1-3) + (2+1-5) + (3+1-7) + (4+1-9)] \\ &= -5\end{aligned}$$

- $\dfrac{\partial J}{\partial W} = \begin{bmatrix} \dfrac{\partial J}{\partial w} & \dfrac{\partial J}{\partial b} \end{bmatrix}^T = \begin{bmatrix} -15 & -5 \end{bmatrix}^T$

Now, we calculate the accumulated sum:

$$\begin{aligned} G &= G + \left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right) \\ &= \begin{bmatrix} 0 & 0 \end{bmatrix}^T + \begin{bmatrix} -15 & -5 \end{bmatrix}^T \odot \begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} 225 & 25 \end{bmatrix}^T \end{aligned}$$

Finally, we update the weight w and bias b after the first iteration:

$$\begin{aligned} W &= W - \eta S^{-\frac{1}{2}} \left(\frac{\partial J}{\partial W}\right) \\ &= W - \eta \left[diag(G)\right]^{-\frac{1}{2}} \left(\frac{\partial J}{\partial W}\right) \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix}^T - 0.1 \begin{bmatrix} 225 & 0 \\ 0 & 25 \end{bmatrix}^{-\frac{1}{2}} \begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} \frac{1}{15} & 0 \\ 0 & \frac{1}{5} \end{bmatrix} \begin{bmatrix} -15 \\ -5 \end{bmatrix} \\ &= \begin{bmatrix} 1.1 \\ 1.1 \end{bmatrix} \end{aligned}$$

Therefore, after the first iteration, the updated weight and bias will be respectively $w = 1.1$ and $b = 1.1$.

Below, we will explore the code for further iterations and the convergence of w and b respectively to 2 and 1 (because $w = 2$, $b = 1$ is indeed a solution to the given sample table).

Python Implementation

This code was adapted from examples in TensorFlow's official documentation [2] and customized for demonstration purposes, reproduced under the Apache 2.0 License and implemented in Python [10].

Firstly, we import the libraries and initialize parameters:

```
import tensorflow as tf
import numpy as np
```

```
# Define the dataset
x_data = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
y_data = np.array([3.0, 5.0, 7.0, 9.0], dtype=np.float32)

# Model parameters
w = tf.Variable([1], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)

# Learning rate
learning_rate = 0.1
```

Then, we will define the linear model and the loss function (here, tensorflow library supports reduce_mean and square method to calculate the MSE function):

```
# Define the linear model
def linear_model(x):
    return w * x + b

# Loss function (Mean Squared Error):
def loss_function(y_pred, y_true):
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

Move to the most important step. In this step, we will use the built-in AdaGrad method from tf.optimizers to define our AdaGrad optimizer. Then, we define a method called train_step to compute the loss function for each iteration (epoch). Finally, we iterate through each epoch and return the loss accuracy at the epoch divisible by 100 (this is to avoid displaying too much accuracy information loss):

```
# Optimizer
optimizer = tf.optimizers.Adagrad(learning_rate=learning_rate)

# Training loop
def train_step(x, y):
    with tf.GradientTape() as tape:
        predictions = linear_model(x)
        loss = loss_function(predictions, y)
    gradients = tape.gradient(loss, [w, b])
    optimizer.apply_gradients(zip(gradients, [w, b]))
    return loss

# Arrays to store parameter values and losses for plotting
w_values, b_values, losses = [], [], []

# Run training for a number of epochs
epochs = 1000
for epoch in range(epochs):
    current_loss = train_step(x_data, y_data)
    w_values.append(w.numpy()[0])
    b_values.append(b.numpy()[0])
    losses.append(current_loss.numpy())
    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss: {current_loss.numpy()}")
```

The output of this cell is:

```
Epoch 0: Loss: 7.5
Epoch 100: Loss: 0.06729909032583237
Epoch 200: Loss: 0.04501043260097504
Epoch 300: Loss: 0.030178122222423553
Epoch 400: Loss: 0.020255565643310547
Epoch 500: Loss: 0.013605390675365925
Epoch 600: Loss: 0.009142904542386532
Epoch 700: Loss: 0.006146133877336979
Epoch 800: Loss: 0.004132492002099752
Epoch 900: Loss: 0.002778972964733839
```

Remark.

AdaGrad is widely used in these scenarios:

- **Natural Language Processing (NLP)**: We know that the AdaGrad algorithm works really well with sparse data and in NLP, data sparsity is common due to the huge vocabulary but inconsistent distribution of word occurrences. Therefore, AdaGrad is ideal for NLP tasks such as text classification [5].
- **Large-scale Machine Learning/Deep Learning**: AdaGrad is effective in scenarios where training data is not only large but includes features of varying importance and frequency. By adapting the learning rates based on the actual data each parameter deals with, it helps in quickly converging to an optimal solution [9].

Although AdaGrad has some advantages such as the *adaptive learning rate*, *handling sparse data* and *little tuning*, it still has some limitations [3]:

- **Accumulation of squared gradients**: Because the sum of squared gradients G_i can be very large over time, this causes the learning rate $\frac{\eta}{G_i}$ to be infinitely small as η is unchanged through the iterations. This leads to the learning rate soon converging to 0 and the model no longer updating parameters, resulting in ineffective learning.
- **Inappropriate for Non-Convex Problems**: Due to its rapid reduction in learning rate, in the context of training deep neural networks, which often involve non-convex optimization, AdaGrad can perform poorly compared to other optimizers like Adam or RMSprop.

8.4 RMSProp

RMSProp, or Root Mean Square Propagation, is an optimization algorithm/method designed for Artificial Neural Network (ANN) training. As in the previous section on AdaGrad, we estimate G_i by accumulating the sum of squared gradients. However, AdaGrad prematurely converges in many scenarios. To resolve this issue, RMSProp uses *exponential averaging* to estimate G_i. This adjustment relies on linear algebra operations involving addition, scalar multiplication and the Hadamard product of column matrices of gradients. Similar to the AdaGrad algorithm, RMSProp also uses matrix multiplication and a diagonal matrix for updating parameters.

What is meant by exponential averaging? The idea is that RMSProp uses a *decay factor* $\lambda \in (0,\ 1)$ and scales the current squared aggregate G_i by λ and adds the result to $1 - \lambda$ times the current squared partial derivative $\frac{\partial J}{\partial w_i}$ [3]. The RMSProp steps are as follows:

+ Step 1: **Initialization**

- η : learning rate
- $J(\mathbf{x})$: stochastic objective function
- $\lambda \in (0,\ 1)$: decay factor
- $W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$: Initial parameter vector
- $G_i = 0,\ \forall i$: Initial sum of squared gradients for the i-th parameter.

For each step, we do the following for the i-th parameter:

+ Step 2: **Exponentially averaged gradients**

$$G_i \longleftarrow \lambda G_i + (1-\lambda)\left(\frac{\partial J}{\partial w_i}\right)^2, \forall i$$

+ Step 3: **Update parameters**

$$w_i \longleftarrow w_i - \frac{\eta}{\sqrt{G_i}}\left(\frac{\partial J}{\partial w_i}\right), \forall i$$

+ Step 4: Repeat Step 2 and Step 3 until one of the following conditions is satisfied:

- Maximum number of iterations reached.
- The difference between the values of $i+1$-th iteration and i-th iteration is smaller than threshold ε.

Formally we can summarise this as Algorithm 4.

Data: $\eta > 0$, $\lambda \in (0,\ 1)$, $J(\mathbf{x})$, $n \geqslant 1$, $iter \geqslant 1$
Variable: W, G_i, t
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix}^T$;
$G_i \leftarrow 0,\ \forall i$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
  Exponentially averaged gradients (2);

$$G_i \longleftarrow \lambda G_i + (1-\lambda)\left(\frac{\partial J}{\partial w_i}\right)^2, \forall i$$

  Update parameters (3);

$$w_i \longleftarrow w_i - \frac{\eta}{\sqrt{G_i}}\left(\frac{\partial J}{\partial w_i}\right), \forall i$$

  $t \leftarrow t+1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\left|w_i^{(t+1)} - w_i^{(t)}\right| < \varepsilon, \forall i$

Algorithm 4: RMSProp algorithm in simplest form

Remark.

Again, in the third step, we can replace $\sqrt{G_i}$ by $\sqrt{G_i + \epsilon}$ where ϵ is relatively small positive number (by default, $\epsilon = 10^{-8}$) to prevent ill-conditioning for G_i.

Data: $\eta > 0$, $\lambda \in (0,\, 1)$, $J(\mathbf{x})$, $n \geqslant 1$, $iter \geqslant 1$
Variable: W, G, t, $\dfrac{\partial J}{\partial W}$, S
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$;
$G \leftarrow \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T = \mathbf{0}$;
$\dfrac{\partial J}{\partial W} \leftarrow \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
Exponentially averaged gradients (2);

$$G \longleftarrow \lambda G + (1-\lambda)\left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right)$$

$S \leftarrow diag(G)$;
Update parameters (3);

$$W \longleftarrow W - \eta\, S^{-\frac{1}{2}}\left(\frac{\partial J}{\partial W}\right)$$

$t \leftarrow t + 1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\max\limits_i \left| w_i^{(t+1)} - w_i^{(t)} \right| < \varepsilon$

Algorithm 5: RMSProp algorithm in matrix form

As previously, we will rewrite the algorithm in matrix form to express it more succinctly.

First, recall the definitions η, $J(\mathbf{x})$, λ as above and *column matrices* G, W, $\dfrac{\partial J}{\partial W}$ *and matrix* S *as given in the AdaGrad section.*
We will rewrite Step 2 in matrix form:

$$\begin{aligned} G &= \begin{bmatrix} \lambda G_1 & \lambda G_2 & ... & \lambda G_n \end{bmatrix}^T + \left[(1-\lambda)\left(\frac{\partial J}{\partial w_1}\right)^2 \quad (1-\lambda)\left(\frac{\partial J}{\partial w_2}\right)^2 \quad ... \quad (1-\lambda)\left(\frac{\partial J}{\partial w_n}\right)^2 \right]^T \\ &= \lambda \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T + (1-\lambda)\left[\left(\frac{\partial J}{\partial w_1}\right)^2 \quad \left(\frac{\partial J}{\partial w_2}\right)^2 \quad ... \quad \left(\frac{\partial J}{\partial w_n}\right)^2 \right]^T \\ &= \lambda G + (1-\lambda)\left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right) \end{aligned}$$

Now, we rewrite Step 3 similarly to Step 3 in the AdaGrad algorithm:

$$W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T - \left[\frac{\eta}{\sqrt{G_1}}\left(\frac{\partial J}{\partial w_1}\right) \quad \frac{\eta}{\sqrt{G_2}}\left(\frac{\partial J}{\partial w_2}\right) \quad ... \quad \frac{\eta}{\sqrt{G_n}}\left(\frac{\partial J}{\partial w_n}\right) \right]^T$$

$$= \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} - \eta \begin{bmatrix} \frac{1}{\sqrt{G_1}} & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{G_2}} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & \frac{1}{\sqrt{G_n}} \end{bmatrix} \begin{bmatrix} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_n} \end{bmatrix} = W - \eta S^{-\frac{1}{2}} \left(\frac{\partial J}{\partial W} \right)$$

Summarising all above expressions, Algorithm 5 gives the RMSProp algorithm in matrix form.

Remark.

The same technique of introducing a small ϵ as previously can be used to avoid ill-conditioning.

To demonstrate the RMSProp algorithm, we have the below example. We will use the same data as Example 8.1 and all assumptions associated with it for a clear comparison.

Example 8.2

Parameters Initialization

- Weight $W = \begin{bmatrix} w & b \end{bmatrix}^T = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$
- Learning rate $\eta = 0.1$
- Sum of squared gradients for parameters w and b: $G = \begin{bmatrix} G_w \\ G_b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- Decay factor $\lambda = 0.9$

Updating parameters

We compute the gradients:

- For w:

$$\begin{aligned} \frac{\partial J}{\partial w} &= \frac{2}{4} \sum_{i=1}^{4} (wx_i + b - y_i)x_i \\ &= \frac{2}{4}[(1 + 1 - 3) \times 1 + (2 + 1 - 5) \times 2 + (3 + 1 - 7) \times 3 + (4 + 1 - 9) \times 4] \\ &= -15 \end{aligned}$$

- For b:

$$\begin{aligned} \frac{\partial J}{\partial b} &= \frac{2}{4} \sum_{i=1}^{4} (wx_i + b - y_i) \\ &= \frac{2}{4}[(1 + 1 - 3) + (2 + 1 - 5) + (3 + 1 - 7) + (4 + 1 - 9)] \\ &= -5 \end{aligned}$$

- $\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial w} & \frac{\partial J}{\partial b} \end{bmatrix}^T = \begin{bmatrix} -15 & -5 \end{bmatrix}^T$

Now, we calculate the exponentially averaged gradients:

$$\begin{aligned} G &= \lambda G + (1-\lambda)\left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right) \\ &= 0.9 \begin{bmatrix} 0 & 0 \end{bmatrix}^T + (1-0.9) \begin{bmatrix} -15 & -5 \end{bmatrix}^T \odot \begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} 0 & 0 \end{bmatrix}^T + 0.1 \begin{bmatrix} 225 & 25 \end{bmatrix}^T \\ &= \begin{bmatrix} 22.5 & 2.5 \end{bmatrix}^T \end{aligned}$$

Finally, we update the weight w and bias b after the first iteration:

$$\begin{aligned} W &= W - \eta S^{-\frac{1}{2}}\left(\frac{\partial J}{\partial W}\right) \\ &= W - \eta \left[diag(G)\right]^{-\frac{1}{2}}\left(\frac{\partial J}{\partial W}\right) \\ &= \begin{bmatrix} 1 & 1 \end{bmatrix}^T - 0.1 \begin{bmatrix} 22.5 & 0 \\ 0 & 2.5 \end{bmatrix}^{-\frac{1}{2}} \begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 0.1 \begin{bmatrix} \frac{1}{\sqrt{22.5}} & 0 \\ 0 & \frac{1}{\sqrt{2.5}} \end{bmatrix} \begin{bmatrix} -15 \\ -5 \end{bmatrix} \\ &\approx \begin{bmatrix} 1.42 \\ 1.42 \end{bmatrix} \end{aligned}$$

So after the first iteration, the updated weight and bias will be respectively $w = 1.42$ and $b = 1.42$.

We use the below code for performing further iterations (the implementation will be almost the same as with AdaGrad, the key differences are about final estimations of w and b as well as the convergence graph).

Python Implementation

Firstly, we import the libraries and initializing parameters (note that we need to define decay factor with default set to 0.9):

```
import tensorflow as tf
import numpy as np
```

```
# Define the dataset
x_data = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
y_data = np.array([3.0, 5.0, 7.0, 9.0], dtype=np.float32)

```

```
# Model parameters
w = tf.Variable([1], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)

# Learning rate
learning_rate = 0.1
decay_factor = 0.9
```

Then, we will define the linear model and the loss function:

```
# Define the linear model
def linear_model(x):
    return w * x + b

# Loss function (Mean Squared Error):
def loss_function(y_pred, y_true):
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

Now we move to the most important step. In this step, we will use the built-in RMSProp method from `tf.optimizers` to define our RMSProp optimizer (here, we will have an extra parameter `rho` which indicates the decay factor). Then, we define a method called `train_step` to compute the loss function for each iteration (epoch). Finally, we iterate through each epoch and return the loss accuracy at epochs divisible by 100.

```
# Optimizer
optimizer = tf.optimizers.RMSprop(learning_rate=learning_rate, rho=decay_factor)

# Training loop
def train_step(x, y):
    with tf.GradientTape() as tape:
        predictions = linear_model(x)
        loss = loss_function(predictions, y)
    gradients = tape.gradient(loss, [w, b])
    optimizer.apply_gradients(zip(gradients, [w, b]))
    return loss

# Arrays to store parameter values and losses for plotting
w_values, b_values, losses = [], [], []

# Run training for a number of epochs
epochs = 1000
for epoch in range(epochs):
    current_loss = train_step(x_data, y_data)
    w_values.append(w.numpy()[0])
    b_values.append(b.numpy()[0])
    losses.append(current_loss.numpy())
    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss: {current_loss.numpy()}")
```

The output of this cell is:

```
Epoch 0: Loss: 7.5
Epoch 100: Loss: 0.04134552553296089
Epoch 200: Loss: 0.03369515761733055
Epoch 300: Loss: 0.03375004231929779
Epoch 400: Loss: 0.033749982714653015
Epoch 500: Loss: 0.033749982714653015
Epoch 600: Loss: 0.033749982714653015
Epoch 700: Loss: 0.033749982714653015
Epoch 800: Loss: 0.033749982714653015
Epoch 900: Loss: 0.033749982714653015
```

So we can see that from epoch 400, the training is saturated – it does not reduce the value of MSE after this epoch.

Remark.

RMSProp is commonly used in these applications:

- **Adaptive Learning Rate**: RMSprop adjusts the learning rate for each parameter based on a moving average of squared gradients, which helps in managing different scales of data effectively [12].
- **Suitable for noisy dataset**: By smoothing the gradients, RMSProp can handle noisy datasets really well compared to AdaGrad or other Gradient Descent algorithms [6].

Although RMSProp has advantages and is better than AdaGrad in many circumstances, there are still limitations associated with it:

- **Sensitive to hyperparameters**: Because the RMSProp algorithm involves learning rate η and decay factor λ (these are hyperparameters), tuning these hyperparameters correctly is crucial for RMSProp's performance.

8.5 Adam

Adam is an extended version of *stochastic gradient descent* which can be implemented in machine learning and deep learning. It uses estimations of the first and second moments of the gradient to adapt the learning rate for each weight of the neural network. The Adam algorithm is the combination of two gradient descent algorithms: Momentum and RMSProp. The core of Adam's functionality involves calculating the exponentially smoothed values and exponentially averaged gradients, which are linear algebra operations (including scalar multiplication, Hadamard product and addition) applied to column matrices of parameters. Additionally, updating parameters requires matrix multiplication alongside the use of a diagonal matrix. We will explain the integration of these two algorithms to the Adam optimizer below [1].

Momentum component

This algorithm uses exponentially smoothed values to accelerate the process of gradient descent. Denote the $n \times 1$ matrix of exponentially smoothed values of the gradient for all components by F. Recall the matrix of parameters W, stochastic objective function $J(\mathbf{x})$, learning rate η and decay factor λ. For each iteration, we update W as follows:

- **Update exponentially smooth value**

$$F \longleftarrow \lambda F + (1-\lambda)\left(\frac{\partial J}{\partial W}\right)$$

- **Update parameters**

$$W \longleftarrow W - \eta F$$

RMSProp component

As mentioned when summarising the steps of RMSProp, instead of taking the accumulated sum of square gradients, it uses exponentially averaged gradients. Recall all the definitions from the *first step of the RMSProp algorithm*, we update w_i for each iteration as below.

- **Exponentially averaged gradients**

$$G \longleftarrow \lambda G + (1-\lambda)\left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right)$$

- **Update parameters**

$$W \longleftarrow W - \eta S^{-\frac{1}{2}}\left(\frac{\partial J}{\partial W}\right) \text{ where } S = diag(G) = \begin{bmatrix} G_1 & 0 & ... & 0 \\ 0 & G_2 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & ... & G_n \end{bmatrix}$$

We know that these two algorithms share common problems such as generalisation of performance. As proposed in [7], Adam combines these optimizers: Momentum and RMSProp to give a more optimized gradient descent algorithm. Integrating the above two optimizers gives us the Adam algorithm as shown below.

+ Step 1: **Initialization**

- η : learning rate
- $J(\mathbf{x})$: stochastic objective function
- $W = \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}$: initial parameter vector
- $G_i = 0$, $\forall i$: initial sum of squared gradients for the i-th parameter.
- $F_i = 0$, $\forall i$: initial exponentially smoothed values for the i-th parameter.
- λ_1, $\lambda_2 \in (0, 1)$: decay factors for F_i and G_i, respectively.

For each step k, we do the following for the i-th parameter:

+ Step 2: **Update exponentially smoothed values and exponentially average gradients**

$$F_i \longleftarrow \lambda_1 F_i + (1-\lambda_1)\left(\frac{\partial J}{\partial w_i}\right), \forall i \text{ and } G_i \longleftarrow \lambda_2 G_i + (1-\lambda_2)\left(\frac{\partial J}{\partial w_i}\right)^2, \forall i$$

+ Step 3: **Compute bias-corrected**

$$\hat{F}_i \longleftarrow \frac{F_i}{1-\lambda_1^k}, \forall i \text{ and } \hat{G}_i \longleftarrow \frac{G_i}{1-\lambda_2^k}, \forall i$$

+ Step 4: **Update parameters**

$$w_i \longleftarrow w_i - \frac{\eta}{\sqrt{\hat{G}_i}}\hat{F}_i,\ \forall i$$

+ Step 5: **Repeating Step 2 to Step 4** until one of the following conditions is satisfied:

- Maximum number of iterations reached.
- The difference between the values of $i+1$-th iteration and i-th iteration is smaller than threshold ε.

Summarising the above, Algorithm 6 gives the Adam algorithm in matrix form.

Data: $\eta > 0,\ \lambda_1,\ \lambda_2 \in (0,\ 1),\ J(\mathbf{x}),\ n \geqslant 1,\ iter \geqslant 1$
Variable: $W,\ G_i,\ F_i,\ t$
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$;
$F_i \leftarrow 0,\ \forall i$;
$G_i \leftarrow 0,\ \forall i$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
  Update exponentially smoothed values and exponentially average gradients (2);

$$F_i \longleftarrow \lambda_1 F_i + (1-\lambda_1)\left(\frac{\partial J}{\partial w_i}\right),\ \forall i \text{ and } G_i \longleftarrow \lambda_2 G_i + (1-\lambda_2)\left(\frac{\partial J}{\partial w_i}\right)^2,\ \forall i$$

  Compute bias-corrected (3);

$$\hat{F}_i \longleftarrow \frac{F_i}{1-\lambda_1^k},\ \forall i \text{ and } \hat{G}_i \longleftarrow \frac{G_i}{1-\lambda_2^k},\ \forall i$$

  Update parameters (4);

$$w_i \longleftarrow w_i - \frac{\eta}{\sqrt{\hat{G}_i}}\hat{F}_i,\ \forall i$$

  $t \leftarrow t+1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\left|w_i^{(t+1)} - w_i^{(t)}\right| < \varepsilon,\ \forall i$

Algorithm 6: Adam algorithm in simplest form

Remark.

In addition to use of ϵ with $\hat{G}_i$, the paper [8] suggested that good default settings for the hyperparameters (including decay factors, learning rate and a small constant) are $\eta = 0.001$, $\lambda_1 = 0.9$, $\lambda_2 = 0.999$ and $\epsilon = 10^{-8}$.
Adam is different to RMSProp in two key respects [3]:

- Adam incorporates momentum by replacing the gradient with an exponentially smoothed value.
- In Step 4 of the algorithm, if we write the update again in terms of F_i and G_i, it will be as follows:

$$\begin{aligned} w_i &= w_i - \frac{\eta}{\sqrt{\hat{G}_i}}\hat{F}_i \\ &= w_i - \frac{\eta}{\sqrt{\frac{G_i}{1-\lambda_2^k}}}\frac{F_i}{1-\lambda_1^k} \\ &= w_i - \eta\frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k}\left(\frac{F_i}{\sqrt{G_i}}\right) \end{aligned}$$

  Therefore, we can rewrite the new learning rate for the k-th iteration as:

$$\eta_k = \eta\frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k}$$

  So the learning rate now is based on each iteration, leading to less hyperparameter tuning than AdaGrad, RMSProp and other gradient descent methods.

Again, we will rewrite the algorithm in matrix form to have a better understanding of it. First, recall the definitions η, $J(\mathbf{x})$, λ_1, λ_2 as in the above algorithm and *column matrices* G, W, $\frac{\partial J}{\partial W}$ *and matrix* S *as in the AdaGrad section.*
We will rewrite Step 2 in matrix form as:

$$\begin{aligned} G &= \begin{bmatrix}\lambda_2 G_1 & \lambda_2 G_2 & \dots & \lambda_2 G_n\end{bmatrix}^T \\ &\quad + \left[(1-\lambda_2)\left(\frac{\partial J}{\partial w_1}\right)^2 \quad (1-\lambda_2)\left(\frac{\partial J}{\partial w_2}\right)^2 \quad \dots \quad (1-\lambda_2)\left(\frac{\partial J}{\partial w_n}\right)^2\right]^T \\ &= \lambda_2\begin{bmatrix}G_1 & G_2 & \dots & G_n\end{bmatrix}^T + (1-\lambda_2)\left[\left(\frac{\partial J}{\partial w_1}\right)^2 \quad \left(\frac{\partial J}{\partial w_2}\right)^2 \quad \dots \quad \left(\frac{\partial J}{\partial w_n}\right)^2\right]^T \\ &= \lambda_2 G + (1-\lambda_2)\left(\frac{\partial J}{\partial W}\odot\frac{\partial J}{\partial W}\right) \end{aligned}$$

and

$$\begin{aligned} F &= \begin{bmatrix}\lambda_1 F_1 & \lambda_1 F_2 & \dots & \lambda_1 F_n\end{bmatrix}^T \\ &\quad + \left[(1-\lambda_1)\left(\frac{\partial J}{\partial w_1}\right) \quad (1-\lambda_1)\left(\frac{\partial J}{\partial w_2}\right) \quad \dots \quad (1-\lambda_1)\left(\frac{\partial J}{\partial w_n}\right)\right]^T \\ &= \lambda_1 G + (1-\lambda_1)\left(\frac{\partial J}{\partial W}\right) \end{aligned}$$

Now, we rewrite Step 3 (for the k-th iteration) :

$$\hat{F} = \begin{bmatrix}\hat{F_1} & \hat{F_2} & ... & \hat{F_n}\end{bmatrix}^T = \begin{bmatrix}\frac{F_1}{1-\lambda_1^k} & \frac{F_2}{1-\lambda_1^k} & ... & \frac{F_n}{1-\lambda_1^k}\end{bmatrix}^T = \frac{1}{1-\lambda_1^k} F$$

$$\hat{G} = \begin{bmatrix}\hat{G_1} & \hat{G_2} & ... & \hat{G_n}\end{bmatrix}^T = \begin{bmatrix}\frac{G_1}{1-\lambda_2^k} & \frac{G_2}{1-\lambda_2^k} & ... & \frac{G_n}{1-\lambda_2^k}\end{bmatrix}^T = \frac{1}{1-\lambda_2^k} G$$

Finally, we rewrite Step 4 (for the k-th iteration):

$$\begin{aligned}
W &= \begin{bmatrix}w_1 & w_2 & ... & w_n\end{bmatrix}^T - \begin{bmatrix}\frac{\eta}{\sqrt{\hat{G}_1}}\hat{F}_1 & \frac{\eta}{\sqrt{\hat{G}_2}}\hat{F}_2 & ... & \frac{\eta}{\sqrt{\hat{G}_n}}\hat{F}_n\end{bmatrix}^T \\
&= \begin{bmatrix}w_1 & w_2 & ... & w_n\end{bmatrix}^T - \eta\frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k}\begin{bmatrix}\frac{1}{\sqrt{G_1}}F_1 & \frac{1}{\sqrt{G_2}}F_2 & ... & \frac{1}{\sqrt{G_n}}F_n\end{bmatrix}^T \\
&= \begin{bmatrix}w_1\\w_2\\\vdots\\w_n\end{bmatrix} - \eta\frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k}\begin{bmatrix}\frac{1}{\sqrt{G_1}} & 0 & ... & 0\\0 & \frac{1}{\sqrt{G_2}} & ... & 0\\\vdots & \vdots & \vdots & \vdots\\0 & 0 & ... & \frac{1}{\sqrt{G_n}}\end{bmatrix}\begin{bmatrix}F_1\\F_2\\\vdots\\F_n\end{bmatrix} \\
&= W - \eta\frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k} S^{-\frac{1}{2}} F
\end{aligned}$$

where $S = diag(G) = \begin{bmatrix}G_1 & 0 & ... & 0\\0 & G_2 & ... & 0\\\vdots & \vdots & \vdots & \vdots\\0 & 0 & ... & G_n\end{bmatrix}$.

Remark.

We can see that in the rewrite of Step 4, the update of W only depends on the learning rate η, decay factors λ_1, λ_2, column matrices G (S is based on elements of G) and F. Therefore, we can ignore Step 3 and directly update W using Step 4.

Finally, summarising the above expressions, Algorithm 7 gives the Adam algorithm in matrix form.

Remark.

The same note applies for Algorithm 7 as for the previous algorithms regarding use of the small positive number ϵ.

Data: $\eta > 0$, λ_1, $\lambda_2 \in (0, 1)$, $J(\mathbf{x})$, $n \geqslant 1$, $iter \geqslant 1$
Variable: W, F, G, t, $\dfrac{\partial J}{\partial W}$, S
Result: Optimized W
$W \leftarrow \begin{bmatrix} w_1 & w_2 & ... & w_n \end{bmatrix}^T$;
$F \leftarrow \begin{bmatrix} F_1 & F_2 & ... & F_n \end{bmatrix}^T = \mathbf{0}$;
$G \leftarrow \begin{bmatrix} G_1 & G_2 & ... & G_n \end{bmatrix}^T = \mathbf{0}$;
$\dfrac{\partial J}{\partial W} \leftarrow \begin{bmatrix} \dfrac{\partial J}{\partial w_1} & \dfrac{\partial J}{\partial w_2} & ... & \dfrac{\partial J}{\partial w_n} \end{bmatrix}^T$;
$t \leftarrow 1$;
while *Stopping condition not met* **do**
  Update exponentially smoothed values and exponentially averaged gradients (2);

$$F \longleftarrow \lambda_1 F + (1 - \lambda_1) \left(\frac{\partial J}{\partial W} \right)$$

$$G \longleftarrow \lambda_2 G + (1 - \lambda_2) \left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W} \right)$$

  $S \leftarrow diag(G)$;
  Update parameters (3);

$$W \longleftarrow W - \eta \frac{\sqrt{1 - \lambda_2^k}}{1 - \lambda_1^k} S^{-\frac{1}{2}} F$$

  $t \leftarrow t + 1$;
end
Stopping Conditions:

- $t = iter$ (Maximum number of iterations reached) **or**
- $\max\limits_i \left| w_i^{(t+1)} - w_i^{(t)} \right| < \varepsilon$

Algorithm 7: Adam algorithm in matrix form

Now, we will illustrate the Adam algorithm using the same example.

Example 8.3

Parameters Initialization

- Weight $W = \begin{bmatrix} w & b \end{bmatrix}^T = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$
- Learning rate $\eta = 0.1$ (although as recommended, η should be equal to 0.001, for the purpose of comparison, we will use the same value as previously).
- Exponentially averaged gradients for parameters w and b: $G = \begin{bmatrix} G_w \\ G_b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

- Exponentially smoothed values for parameters w and b: $F = \begin{bmatrix} F_w \\ F_b \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$
- Decay factors $\lambda_1 = 0.9$ and $\lambda_2 = 0.999$

Updating parameters

We compute the gradients:

- For w:

$$\begin{aligned} \frac{\partial J}{\partial w} &= \frac{2}{4}\sum_{i=1}^{4}(wx_i + b - y_i)x_i \\ &= \frac{2}{4}[(1+1-3)\times 1 + (2+1-5)\times 2 + (3+1-7)\times 3 + (4+1-9)\times 4] \\ &= -15 \end{aligned}$$

- For b:

$$\begin{aligned} \frac{\partial J}{\partial b} &= \frac{2}{4}\sum_{i=1}^{4}(wx_i + b - y_i) \\ &= \frac{2}{4}[(1+1-3) + (2+1-5) + (3+1-7) + (4+1-9)] \\ &= -5 \end{aligned}$$

- $\dfrac{\partial J}{\partial W} = \begin{bmatrix} \dfrac{\partial J}{\partial w} & \dfrac{\partial J}{\partial b} \end{bmatrix}^T = \begin{bmatrix} -15 & -5 \end{bmatrix}^T$

Note that this is the first iteration so $k = 1$. Now, we calculate the exponentially smoothed values:

$$\begin{aligned} F &= \lambda_1 F + (1-\lambda_1)\left(\frac{\partial J}{\partial W}\right) \\ &= 0.9\begin{bmatrix} 0 & 0 \end{bmatrix}^T + (1-0.9)\begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} -1.5 & -0.5 \end{bmatrix}^T \end{aligned}$$

Next, we calculate the exponentially averaged gradients:

$$\begin{aligned} G &= \lambda_2 G + (1-\lambda_2)\left(\frac{\partial J}{\partial W} \odot \frac{\partial J}{\partial W}\right) \\ &= 0.999\begin{bmatrix} 0 & 0 \end{bmatrix}^T + (1-0.999)\begin{bmatrix} -15 & -5 \end{bmatrix}^T \odot \begin{bmatrix} -15 & -5 \end{bmatrix}^T \\ &= \begin{bmatrix} 0 & 0 \end{bmatrix}^T + 0.001\begin{bmatrix} 225 & 25 \end{bmatrix}^T \\ &= \begin{bmatrix} 0.225 & 0.025 \end{bmatrix}^T \end{aligned}$$

Finally, we update the weight w and bias b after the first iteration:

$$W = W - \eta \frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k} S^{-\frac{1}{2}} F$$

$$
\begin{aligned}
&= W - \eta \frac{\sqrt{1-\lambda_2^k}}{1-\lambda_1^k} [diag(G)]^{-\frac{1}{2}} F \\
&= \begin{bmatrix} 1 & 1 \end{bmatrix}^T - 0.1 \frac{\sqrt{1-0.999^1}}{1-0.9^1} \begin{bmatrix} 0.225 & 0 \\ 0 & 0.025 \end{bmatrix}^{-\frac{1}{2}} \begin{bmatrix} -1.5 & -0.5 \end{bmatrix}^T \\
&= \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \sqrt{0.001} \begin{bmatrix} \frac{1}{\sqrt{0.225}} & 0 \\ 0 & \frac{1}{\sqrt{0.025}} \end{bmatrix} \begin{bmatrix} -1.5 \\ -0.5 \end{bmatrix} \\
&= \begin{bmatrix} 1.1 \\ 1.1 \end{bmatrix}
\end{aligned}
$$

So after the first iteration, the updated weight and bias will be respectively $w = 1.1$ and $b = 1.1$.

The below code is used for further iterations (the implementation will be almost the same as with AdaGrad and RMSProp, the key differences are about final estimations of w and b as well as the convergence graph).

Python Implementation

Firstly, we import the libraries and initializing parameters:

```
import tensorflow as tf
import numpy as np
```

```
# Define the dataset
x_data = np.array([1.0, 2.0, 3.0, 4.0], dtype=np.float32)
y_data = np.array([3.0, 5.0, 7.0, 9.0], dtype=np.float32)

# Model parameters
w = tf.Variable([1], dtype=tf.float32)
b = tf.Variable([1], dtype=tf.float32)

# Learning rate
learning_rate = 0.1
```

Then, we will define the linear model and the loss function:

```
# Define the linear model
def linear_model(x):
    return w * x + b

# Loss function (Mean Squared Error):
def loss_function(y_pred, y_true):
    return tf.reduce_mean(tf.square(y_pred - y_true))
```

In this step, we will use the built-in Adam method from `tf.optimizers` to define our Adam optimizer (here, we will have some extra hyperparameters such as `learning_rate`, `beta_1` and `beta_2` which are decay factors and constant epsilon). Then, we define a method called `train_step` to compute the loss function for each iteration (epoch). Finally, we iterate through each epoch and return the loss accuracy at epochs divisible by 100.

```
# Optimizer: Using Adam with specific hyperparameters
optimizer = tf.optimizers.Adam(
    learning_rate=0.1,     # Initial learning rate
    beta_1=0.9,           # Exponential decay rate for the 1st moment estimates
    beta_2=0.999,         # Exponential decay rate for the 2nd moment estimates
    epsilon=1e-08         # Small constant for numerical stability
)

# Training loop
def train_step(x, y):
    with tf.GradientTape() as tape:
        predictions = linear_model(x)
        loss = loss_function(predictions, y)
    gradients = tape.gradient(loss, [w, b])
    optimizer.apply_gradients(zip(gradients, [w, b]))
    return loss

# Arrays to store parameter values and losses for plotting
w_values, b_values, losses = [], [], []

# Run training for a number of epochs
epochs = 1000
for epoch in range(epochs):
    current_loss = train_step(x_data, y_data)
    w_values.append(w.numpy()[0])
    b_values.append(b.numpy()[0])
    losses.append(current_loss.numpy())
    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss: {current_loss.numpy()}")
```

The output of this cell is:

```
Epoch 0: Loss: 7.5
Epoch 100: Loss: 0.0019636775832623243
Epoch 200: Loss: 1.6936496649577748e-09
Epoch 300: Loss: 1.3642420526593924e-12
Epoch 400: Loss: 1.1368683772161603e-13
Epoch 500: Loss: 1.1368683772161603e-13
Epoch 600: Loss: 1.1368683772161603e-13
Epoch 700: Loss: 1.1368683772161603e-13
Epoch 800: Loss: 1.1368683772161603e-13
Epoch 900: Loss: 1.1368683772161603e-13
```

We can see that from epoch 400, the training is saturated; it does not reduce the value of MSE after this epoch.

Remark.

Since Adam is a combination of the other optimizers, it has some advantages that makes it extremely popular in Machine Learning/Deep Learning [8]:

- **Adaptive Learning rate**: Because the Adam algorithm adjusts the learning rate for each parameter based on the estimation of the first and second moments (respectively, F_i and G_i) of the gradients, this allows for efficient learning rate updates.
- **Robustness to Hyperparameters**: Since the Adam algorithm gives intuitive insights for hyperparameters such as the learning rate and decay factors, the Adam algorithm requires less tuning, with the hyperparameters also less sensitive to changes than the AdaGrad and RMSProp optimizers.
- **Better Convergence** Since Adam combines RMSProp and Momentum methods, it incorporates many advantages from these two algorithms, including convergence, making it reliable for practical problems.

However, it still has some limitations:

- **Convergence Issue**: There is evidence [11] suggesting that Adam can fail to converge to the optimal solution under certain conditions or may converge to a suboptimal solution. In some cases, switching to Stochastic Gradient Descent provides better performance than the Adam algorithm.
- **Computationally intensive**: Because we need to store and update the first and second moment for each parameter, Adam requires more computation than other optimizers such as Adam or RMSProp.

8.6 Discussion

For comparison of results, we will use the same Example 8.1. Moreover, we use the `Numpy` and `matplotlib` libraries for returning the parameters and visualizing the loss function over epochs.

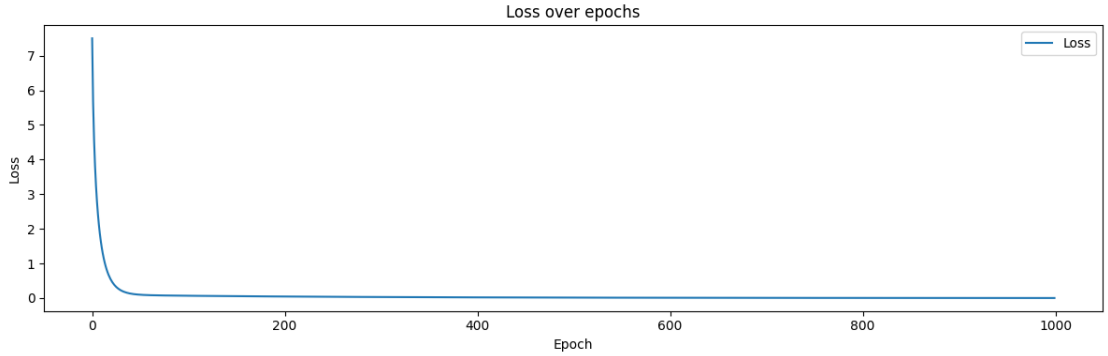
The following code generates the final paramaters. We provide results for each of the AdaGrad, RMSProp and Adam methods.

```
# Output the final parameters
print(f"Final parameters: w = {w.numpy()[0]}, b = {b.numpy()[0]}")
```

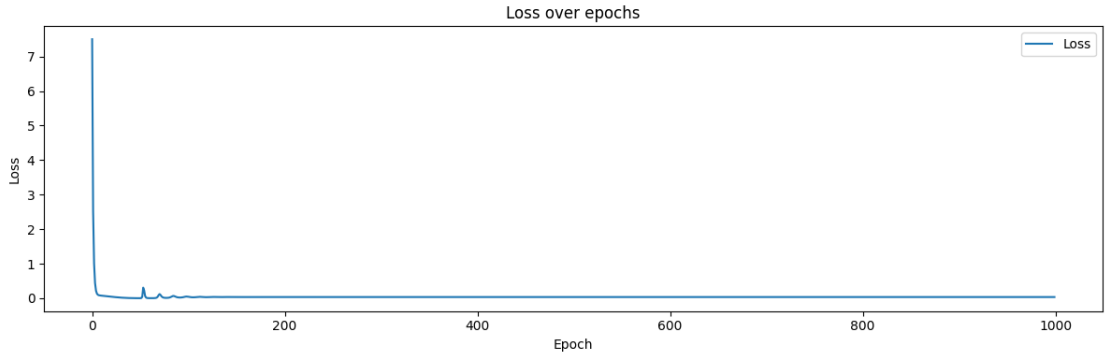
```
Final parameters (AdaGrad): w = 1.9630769491195679, b = 1.105146884918213
Final parameters (RMSProp): w = 1.9499999284744263, b = 0.9500000476837158
Final parameters (Adam)   : w = 1.9999998807907104, b = 1.000000238418579
```

All values for w are close to the exact weight of 2, while all values for the bias are close to the expected value of 1, however we can see that for this example, Adam produces the best final results.

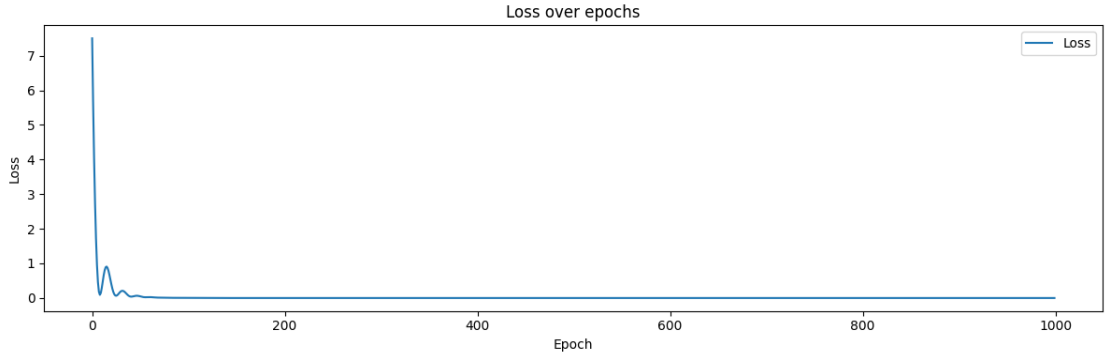
Plots showing how the loss function converges through the epochs were generated using the following code with graphical outputs shown in Figure 13.17.



(a) AdaGrad



(b) RMSProp



(c) Adam

Figure 8.1: Plots comparing convergence for each of the algorithms on the data given in Example 8.1. Created using Python [10].

```python
import matplotlib.pyplot as plt

# Plotting the results
plt.figure(figsize=(12, 4))
plt.plot(losses, label='Loss')
plt.title('Loss over epochs')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

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

We see that the Loss for all three algorithms falls steeply at first, with only small decreases after the first 10 epochs. RMSProp has small flucations roughly from epoch 30 to ephoc 50, while Adam has some between roughly 20 and 60.

8.7 Conclusion

This report has explored the integral role of linear algebra in optimizing machine learning algorithms, focusing on AdaGrad, RMSProp, and Adam. The mathematical representations highlight linear algebra concepts as fundamental in facilitating efficient computations necessary for these algorithms, especially in managing large-scale data and complex model architectures.

AdaGrad's approach to adapting the learning rate to parameters provides early rapid changes but can lead to premature slowing of learning rates. In contrast, RMSProp modifies AdaGrad's method by introducing a decay factor to avoid diminishing learning rates too quickly. Adam builds upon these improvements by combining momentum and adaptive learning rate techniques, which our example showed to provide the most consistent and robust convergence.

Future research should continue to refine these algorithms, exploring hybrid techniques that could offer even greater efficiency and flexibility in handling diverse and evolving data landscapes.

Context

This article was adapted from a report submitted for SIT292.

About the Author

Chris is a first-year student at Deakin University pursuing Bachelor of Computer Science. He is interested in many fields of research such as optimization, image processing and speech recognition. Chris is very excited with the given opportunities to explore and connect with different researchers to have more understanding about these vast fields.

Acknowledgements

I would like to express my special thanks and gratitude to Simon James, who taught me in the Linear Algebra for Data Analysis unit, has assisted me and gave me really detailed feedback on my report, especially structuring it professionally to be almost like a typical research paper.

References

[1] A. Ajagekar. Adam. *Cornell University Computational Optimization Open Textbook*, 2021. Available online: https://optimization.cbe.cornell.edu/index.php?title=Adam.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from https://www.tensorflow.org/.

[3] Charu C. Aggarwal. *Linear Algebra and Optimization for Machine Learning: A Textbook*. Springer, 2020.

[4] D. Villarraga. Adagrad. *Cornell University Computational Optimization Open Textbook*, 2021. Available online: https://optimization.cbe.cornell.edu/index.php?title=AdaGrad.

[5] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[6] G. Goh. *Why momentum really works*. Distill, 2017. Available online: https://distill.pub/2017/momentum/.

[7] W. E. L. Ilboudo, T. Kobayashi, and K. Sugimoto. Robust stochastic gradient descent with student-t distribution based first-order momentum. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–14, 2020.

[8] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015. arXiv preprint arXiv:1412.6980.

[9] H. Brendan McMahan and M. Streeter. Delay-tolerant algorithms for asynchronous distributed online learning. *Advances in Neural Information Processing Systems*, 27, 2014.

[10] Python Core Developers. *Python: A dynamic, open source programming language*. Python Software Foundation, 2022. https://www.python.org/downloads/release/python-3110/.

[11] S. J. Reddi, S. Kale, and S. Kumar. On the convergence of adam and beyond. In *6th International Conference on Learning Representations (ICLR)*, 2018.

[12] T. Tieleman and G. Hinton. *Lecture 6.5—RMSprop: Divide the gradient by a running average of its recent magnitude*. COURSERA: Neural Networks for Machine Learning, 2012.

9

Linear programming and performance enhancement through LU factorization

Van Nam Quang Nguyen

Abstract

Linear programming is a fundamental optimization technique used across various fields to solve constrained problems efficiently. This report explores the core concepts of linear programming, with a particular focus on the simplex and revised simplex methods. One major challenge in solving linear programs is the high computational cost associated with matrix inversion, especially for large-scale problems. To address this, we introduce LU factorization as an efficient alternative. We examine the sparse Bartels-Golub method, which uses sparse matrices to perform partial updates of the lower and upper triangular matrices in LU factorization. This approach significantly enhances the computational efficiency of the revised simplex method, making it more suitable for solving large-scale linear programming problems in a time-efficient manner.

9.1 Introduction

Linear programming is a powerful mathematical technique used for optimizing outcomes such as maximizing profits or minimizing costs under a set of constraints. It serves as a fundamental tool in operational research and various fields, including economics, engineering, and logistics, addressing real-world problems that require the optimal allocation of resources. The essential components of linear programming—objective functions, constraints, and feasible regions—form the backbone of this optimization method, enabling practitioners to model complex scenarios effectively. Central to linear programming is the simplex method, a widely-used algorithm that systematically navigates the feasible region to identify optimal solutions.

However, traditional implementations can be computationally intensive, especially for large-scale problems. To enhance performance, advanced linear algebra techniques, such as LU factorization and Etas factorization, are integrated into the simplex method, simplifying matrix operations and enabling quicker updates while maintaining numerical stability. The Sparse Bartels-Golub method further refines this approach by exploiting the sparse structure of many real-world linear programming problems, significantly improving computational efficiency. By examining these concepts and methodologies, this work aims to provide a comprehensive understanding of linear programming, emphasizing the crucial role of linear algebra in developing efficient optimization strategies and addressing complex decision-making challenges across various domains.

9.2 Linear optimisation

This section introduces the key concepts and notation used in linear optimsation. We begin with a classical real-world problem.

Problem Statement 9.1: Furniture Company Context

A furniture company produces two types of products: Tables and Chairs. Each Table generates $30 profit per unit, and each Chair generates $20 profit per unit.
The company has 120 hours of labor available per week. Each Table requires 4 hours of labor, and each Chair requires 2 hours. Additionally, the company has a limited supply of 100 units of wood, with each Table requiring 5 units and each Chair requiring 3 units.
Find the number of Tables(T) and Chairs (C) that gain the maximize profit.

The objective function is:

$$Z = 30T + 20C$$

The constraints are:

$$\begin{aligned} 4T + 2C &\leq 120 \\ 5T + 3C &\leq 100 \\ T, C &\geq 0 \end{aligned}$$

The given problem is an example of *Linear Programming*, where the goal is to optimize an objective function subject to specific constraints. In this paper, we will discover how to solve that problem.

Components of Linear Programming

Definition 9.1: Decision variables

Decision variables are the values that need to be determined to obtain optimal solutions [9], denoted:

$$x_j \geq 0, \quad \text{where } j = 1, 2, ..., n.$$

Since decision variables often come from real-world contexts, they are usually required to be nonnegative. For example, production level or days of production cannot be negative.

Definition 9.2: Objective function

The objective function is the linear function with decision and other variables as inputs, which needs to be maximized or minimized. The general form is given as:

$$\zeta = c_1 x_1 + c_2 x_2 + ... + c_n x_n$$

In real-world applications, the objective function is often subject to various *constraints*, which can include equalities and inequalities, i.e., conditions requiring linear combinations of variables

to be greater, less than, or equal to given quantitities. These constraints help guide the process of determining the permissible values of decision variables.

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \left\{ \begin{array}{c} \geq \\ = \\ \leq \end{array} \right\} b$$

In this case, b can be considered as a *bound* of the function, either an *upper bound* or *lower bound* that constrains the function. We can convert from inequality to equality constraints by using a non-negative *slack* variable, w [9]:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n + w = b, \quad w \geq 0$$

However, to convert from equality to inequality requires us to replace the former with two equations:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$$

Can be turn to:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq b$$
$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq b$$

The standard form for a maximization problem is typically expressed as follows:

$$\begin{array}{rl} \text{Maximize} & \zeta = c_1x_1 + c_2x_2 + \cdots + c_nx_n \\ \text{subject to} & a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \leq b_m \\ \text{and} & x_1, x_2, \ldots, x_n \geq 0. \end{array}$$

where m is the number of constraints; and n is the number of decision variables.

The standard form can be written in matrix representation as:

$$\begin{array}{rl} \text{Maximize} & \zeta = \mathbf{c^T x} \\ \text{subject to} & A\mathbf{x} \leq \mathbf{b} \\ \text{and} & \mathbf{x} \geq 0. \end{array}$$

i.e., we will have:

$$\begin{array}{rl} \text{Maximize} & \zeta = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}^T \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \\ \text{subject to} & \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \leq \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \\ \text{and} & \mathbf{x} \geq 0. \end{array}$$

If we solve such a system, the *solution* is the proposed values of the decision variables. However, if the linear program does not have any feasible solutions, then it is called *infeasible*.

Determining which problems are feasible and which are infeasible is crucial.

Definition 9.3: Solution terminology

A *feasible solution* is any solution that satisfies all constraints
A *feasible region* is the collection of all feasible solutions obtained from the objective function.

$$x \in \mathbb{R}^n, Ax \leq b, x \geq 0$$

An *optimal solution* is a solution that maximises the value of the objective function.

Since the feasible region forms a convex set, it's important to consider the concept of "extreme points" within these sets. In linear programming, extreme points provide a significant method for solving the problem.

Theorem 9.1

If a linear programming (LP) problem has an optimal solution, then there exists an optimal extreme point of the feasible set $S = \{x \in \mathbb{R}^n \mid Ax = b,\ x \geq 0\}$.

Basic solutions

Given the following linear programming example:

$$\begin{aligned} \text{Maximize} \quad & z = 3x_1 + 2x_2 \\ \text{Subject to:} \quad & x_1 + x_2 \leq 4 \\ & 2x_1 + x_2 \leq 6 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Let's convert to standard form by using slack variables. In this problem, there are two constraints and so we can use two slack variables s_1, s_2:

$$\begin{aligned} x_1 + x_2 + s_1 &= 4 \\ 2x_1 + x_2 + s_2 &= 6 \end{aligned}$$

Now if we set x_1, x_2 to 0, then s_1 and s_2 is respectively 4 and 6. We refer to the variables x_1 and x_2, which are set to zero, as *non-basic variables*, while s_1 and s_2 serve as the *basic variables*. These selections for s_1, x_2, s_1, s_2 are collectively referred to as a basic solution.

Definition 9.4: Basic solution

A *basic solution* in linear programming is a solution obtained by selecting a subset of variables (basic variables) which forms the basis for the column space of the constraint coefficient matrix and setting the remaining variables (non-basic variables) to zero.
A basic solution satisfies the non-negativity condition called *feasible basic solution*, while it's called *infeasible basic solution*.

From the above equations, we can represent the same information using matrices:

$$A = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 2 & 1 & 0 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 4 \\ 6 \end{bmatrix}$$

Now we can split matrix A into two matrices. One contains a *basis* for the column space of the original constraint coefficient matrix, while the other is the identity matrix corresponding with the slack variables:

$$N = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Now following definition is provided for basic feasible solutions. It will be fundamental in later parts of this article as it serves as core for the upcoming section on the Simplex method.

Definition 9.5: Basic feasible solution (BFS)

A basic feasible solution (BFS) [2] is a solution to the linear programming problem where $\mathbf{x} \in S = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ and $\mathbf{x} = \begin{bmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{bmatrix}$, with matrix A, which has rank m, partitioned into two submatrices, with A $= [B|N]$

- **Basis matrix** B: an $m \times m$ square, invertible matrix formed by selecting m linearly independent columns of A, corresponding with the **set of basic variables** $x_B \geq 0$.
- **Non-basis matrix** N: the remaining $m \times (n - m)$ submatrix, corresponding with the **set of non-basic variables** $x_N = 0$

From the Definition 9.2, we can obtain the following system:

$$[B|N] \begin{bmatrix} \mathbf{x}_B \\ \mathbf{x}_N \end{bmatrix} = \mathbf{b}$$

We can substitute the above equation to obtain:

$$B\mathbf{x}_B + N\mathbf{x}_N = \mathbf{b}$$

Isolating $\mathbf{x}_B$, we can rearrange the equation:

$$B\mathbf{x}_B = \mathbf{b} - N\mathbf{x}_N$$

Now let's solve for $\mathbf{x}_B$, with B **invertible** (or **non-singular**) by multiplying both sides by B^{-1}:

$$\mathbf{x}_B = B^{-1}(\mathbf{b} - N\mathbf{x}_N) = B^{-1}\mathbf{b} - B^{-1}N\mathbf{x}_N$$

Then the following system can be broken down as the following, by assuming that $\mathbf{x}_N = 0$:

$$\mathbf{x}_B = B^{-1}\mathbf{b}$$

Hence $\mathbf{x}_B$ and $\mathbf{x}_N$ are the **basic solutions** of $A\mathbf{x} = \mathbf{b}$

Theorem 9.2

Given $\mathbf{x} \in S = \{\mathbf{x}, A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ then it holds that:

$$\mathbf{x} \textit{ is an extreme point} \iff \mathbf{x} \textit{ is a feasible basic solution.}$$

From that theorem it follows that

Theorem 9.3

If there exists an optimal solution, then there is an optimal BFS.

9.3 Simplex method

In this section we give an overview of solving systems of linear equations using the Simplex method.

Degeneracy

Although each BFS uniquely represents an extreme point, the reverse is not true; a single extreme point can correspond with multiple BFSs. This nonuniqueness arises because some variables in the basic set $\mathbf{x}_B$ may be zero, leading to indistinguishable representations of the same extreme point, so that we come up with a new definition.

Definition 9.6: Degenerate

A feasible basic solution $\mathbf{x} \in P = \{\mathbf{x} \in \mathbb{R}^2 \mid A\mathbf{x} = \mathbf{b}, \mathbf{x} \geq \mathbf{0}\}$ is *degenerate* if at least one of the variables in the basic set $\mathbf{x}_B$ is zero. In contrast, $\mathbf{x} \in P$ is said to be **nondegenerate** if all m of the basic variables are positive.

Basic Feasible Solutions in the Simplex Method

Let us talk about the general Simplex algorithm. We have mentioned that "the optimal solution must be a BFS", therefore, in all BFS, we can find the optimal solution that can improve our objective function. Having found a BFS as an extreme point of the feasible region, we will then try to find an *adjacent* BFS that can improve the objective function. By 'adjacent', we mean that we will swap basic variables with non-basic variables.

Definition 9.7: Adjacent

Two different basic feasible solutions $\mathbf{x}_m$ and $\mathbf{x}_n$ are *adjacent* if they have $m - 1$ basic variables in common.

The primary goal of the simplex method is to navigate between adjacent basic feasible solutions (BFS) that correspond with extreme points. However, if multiple BFSs correspond with the same extreme point or if that extreme point is not unique, we may find ourselves unable to improve our objective function. In this situation, the movement between BFSs can enter a cycle, where we might move to a different BFS only to return to the previous extreme point. Therefore, the BFS selected needs to satisfy the *non-degeneracy* condition.

Simplex method

Now let us take look at the linear programming problem. We will have $\mathbf{x} = (\mathbf{x}_B, \mathbf{x}_N)$ which is the basic feasible solution and the partitioned cost vector $\mathbf{c} = [\mathbf{c}_B|\mathbf{c}_N]^T$, then the objective function can be written as:

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T\mathbf{x}_B + \mathbf{c}_N^T\mathbf{x}_N$$

Since we assume that the basic solution x_B will be:

$$\mathbf{x}_B = B^{-1}\mathbf{b} - B^{-1}N\mathbf{x}_N$$

then the objective function will be written as:

$$\mathbf{c}^T\mathbf{x} = \mathbf{c}_B^T(B^{-1}\mathbf{b} - B^{-1}N\mathbf{x}_N) + \mathbf{c}_N^T\mathbf{x}_N = \mathbf{c}^TB^{-1}\mathbf{b} + (\mathbf{c}_N^T - \mathbf{c}_B^TB^{-1}N)\mathbf{x}_N$$

Now let's say the set of non-basic variables will have k elements, then we can rewrite the above equation as:

$$\zeta = z(x_1, ..., x_n) = \mathbf{c}^TB^{-1}\mathbf{b} + \sum_{j=0}^{k}(c_j - \mathbf{c}_B^TB^{-1}a_j)x_j$$

Since $\mathbf{c}_N^T$ is a row vector, each element c_j represents the value at the j-th position in that row. This means that every entry in $\mathbf{c}_N^T$ corresponds to a specific column index. Similarly, $\mathbf{x}_N$ is a column vector, where each element x_j represents the value at the j-th position in that column. Additionally, the column vector a_j is obtained by selecting the j-th column from the non-basis matrix $\mathbf{N}$.

We have already mentioned that main concept of the simplex method is to move between adjacent BFSs, which will affect the values of the decision variables. More precisely, a non-basic variable will increase from 0 to be a positive number and a basic variable will decrease from a positive number to 0. Hence we can denote the change in the non-basic variables by the derivative:

$$\frac{\partial z}{\partial x_j} = c_j - \mathbf{c}_B^TB^{-1}a_j$$

This equation corresponds with the increase to x_j. If $\mathbf{c}_j - \mathbf{c}_B^TB^{-1}a_j > 0$, the non-basic variable x_j will increase from zero to positive. From that we will have the *reduced cost* which is denoted as:

$$-\frac{\partial z}{\partial x_j} = \mathbf{c}_B^TB^{-1}a_j - c_j = z_j - c_j$$

This means if the reduced cost is negative, the value of the derivative $\frac{\partial \zeta}{\partial x_j}$ will be *positive* (when maximising). In other words, when the derivative is as large as possible and positive, x_j will gain the largest increase [1].

Now we will consider the feasible basic solution with the current variable x_j:

$$\mathbf{x}_B = B^{-1}\mathbf{b} - B^{-1}A_jx_j$$

We will denote $\bar{\mathbf{b}} = B^{-1}\mathbf{b}$ and $\bar{\mathbf{a}_j} = B^{-1}A_j$, then we can write

$$\mathbf{x}_B = \bar{\mathbf{b}} - \bar{\mathbf{a}_j}x_j$$

Since $\bar{\mathbf{b}}, \bar{\mathbf{a}_j}$ are column vectors, then we will have:

$$\begin{bmatrix} x_{B1} \\ x_{B2} \\ \vdots \\ x_{B1} \end{bmatrix} = \begin{bmatrix} \bar{b_1} \\ \bar{b_2} \\ \vdots \\ \bar{b_m} \end{bmatrix} - \begin{bmatrix} \bar{a_{j1}} \\ \bar{a_{j2}} \\ \vdots \\ \bar{a_{jm}} \end{bmatrix} x_j$$

In this case, we know that $\bar{b_i} \geq 0$ and $\bar{a_{ji}} > 0$, then if we increase x_j, the variable x_{Bi} will be decreased until it reaches 0, so that we will have:

$$0 = \bar{b_i} - \bar{a_{ji}} x_j \implies x_j = \frac{\bar{b_i}}{\bar{a_{ji}}}$$

In this case the largest possible value of x_j will satisfy the *minimum ratio test*:

$$min \left\{ \frac{\bar{b_i}}{\bar{a_{ji}}} : i = 1, ..., m \quad \text{and } a_{ij} > 0 \right\}$$

Example 9.1 is presented below to illustrate these ideas.

Example 9.1: Solving with Simplex Method

$$\begin{aligned} \text{Maximize} \quad & z = 3x_1 + 2x_2 + 4x_3 \\ \text{s.t.} \quad & 2x_1 + x_2 + x_3 \leq 4 \\ & x_1 + 3x_2 + 2x_3 \leq 6 \\ & x_1 + 2x_2 + 3x_3 \leq 5 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Step 1: Convert the problem to standard form
First, we will turn the linear problem into standard form. To do this, we will introduce slack variables s_1, s_2, and s_3 to convert the inequalities into equalities:

$$\begin{aligned} 2x_1 + x_2 + x_3 + s_1 &= 4 \\ x_1 + 3x_2 + 2x_3 + s_2 &= 6 \\ x_1 + 2x_2 + 3x_3 + s_3 &= 5 \\ x_1, x_2, x_3, s_1, s_2, s_3 &\geq 0 \end{aligned}$$

From this we have:

$$\mathbf{c} = \begin{bmatrix} 3 \\ 2 \\ 4 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}, \quad A = \begin{bmatrix} 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & 3 & 2 & 0 & 1 & 0 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ 6 \\ 5 \end{bmatrix}$$

Step 2: Identify variables and matrices, basis and non-basis
Now we will partition matrix A into basis and non-basis matrices:

$$B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad N = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 3 & 2 \\ 1 & 2 & 3 \end{bmatrix}$$

So that we now have:

$$\mathbf{x}_B = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}, \quad \mathbf{x}_N = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{c}_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{c}_N = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix}$$

Since B is an identity matrix, its inverse will be the same:

$$B^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Step 3: Choose Entering and Leaving Variables
Now all the needed elements are gathered, we will calculate the *reduced cost*:

$$\mathbf{c}_B^T B^{-1} N - \mathbf{c}_N^T = \begin{bmatrix} -3 & -2 & -4 \end{bmatrix}$$

From the reduced cost we can find the following:

$$\frac{\partial z}{\partial x_1} = 3, \quad \frac{\partial z}{\partial x_2} = 2, \quad \frac{\partial z}{\partial x_3} = 4$$

We will choose x_3 as the entering variable, but we still need to determine which variable will leave the basis. To do this, we will look at the column of A that corresponds with x_3. For the minimum ratio test we have:

$$\min \left\{ \frac{4}{1}, \frac{6}{2}, \frac{5}{3} \right\}$$

Since the smallest ratio occurs for the element s_3, we will have that x_3 will enter and s_3 will leave.

Step 4: Repeat step 2, 3 until all reduced costs are positive
The new non-basic and basic variables will be:

$$\mathbf{x}_B = \begin{bmatrix} s_1 \\ s_2 \\ x_3 \end{bmatrix}, \quad \mathbf{x}_N = \begin{bmatrix} x_1 \\ x_2 \\ s_3 \end{bmatrix}, \quad \mathbf{c}_B = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix}, \quad \mathbf{c}_N = \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix}$$

Then we will have the following matrices:

$$B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 3 \end{bmatrix}, \quad N = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Now let us make the following calculations:

$$B^{-1}\mathbf{b} = \begin{bmatrix} 7/3 \\ 8/3 \\ 5/3 \end{bmatrix}, \quad B^{-1}N = \begin{bmatrix} 5/3 & 1/3 & -1/3 \\ 1/3 & 5/3 & -2/3 \\ 1/3 & 2/3 & 1/3 \end{bmatrix}$$

We can easily calculate the reduced cost as:

$$\mathbf{c}_B^T B^{-1} N - \mathbf{c}_N^T = \begin{bmatrix} 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} 5/3 & 1/3 & -1/3 \\ 1/3 & 5/3 & -2/3 \\ 1/3 & 2/3 & 1/3 \end{bmatrix} - \begin{bmatrix} 3 & 2 & 0 \end{bmatrix} = \begin{bmatrix} -5/3 & 2/3 & 4/3 \end{bmatrix}$$

In this case, we might find that the first element in reduced cost is negative so that we will continue the process. Now we will choose x_1 to enter and determine the minimum ratio test:

$$\min \left\{ \frac{7}{5}, 8, 5 \right\}$$

We will choose s_1 to enter and x_1 to leave. The solution will be found following the repeated steps above, leading to:

$$\mathbf{x}_B = \begin{bmatrix} x_1 \\ s_2 \\ x_3 \end{bmatrix}, \quad \mathbf{x}_N = \begin{bmatrix} s_1 \\ x_2 \\ s_3 \end{bmatrix}, \quad \mathbf{c}_B = \begin{bmatrix} 3 \\ 0 \\ 4 \end{bmatrix}$$

$$\mathbf{c}_N = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix}, \quad B^{-1}\mathbf{b} = \begin{bmatrix} 7/5 \\ 11/5 \\ 6/5 \end{bmatrix}, \quad B^{-1}N = \begin{bmatrix} 3/5 & 1/5 & -1/5 \\ -1/5 & 8/5 & -3/5 \\ -1/5 & 3/5 & 2/5 \end{bmatrix}$$

Then the reduced cost will be:

$$\mathbf{c}_B^T B^{-1} N - \mathbf{c}_N^T = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

Now all of the reduced costs are positive, so we assume:

$$\mathbf{x}_B^* = \begin{bmatrix} x_1 \\ s_2 \\ x_3 \end{bmatrix} = B^{-1}\mathbf{b} = \begin{bmatrix} 7/5 \\ 11/5 \\ 6/5 \end{bmatrix}$$

Step 5: Compute the optimal value
Then $x_1 = \frac{7}{5}, x_2 = 0, x_3 = \frac{6}{5}$ with x_2 as the non-basic variable. Passing these variables to the objective function we can easily find the max:

$$z = 3 \times \frac{7}{5} + 2 \times 0 + 4 \times \frac{6}{5} = 9$$

9.4 Updated Simplex method

From the knowledge above we will introduce an updated version, which will shorten the steps required to achieve the optimal solution. We can see that the Simplex method as given above is heavily dependent on B^-1 since it appears in several computations including $B^-1\mathbf{b}, B^-1A_j$..., so that it will be convenient if we can avoid computing it so often and only compute when finding the reduced cost and the **direction** $\bar{\mathbf{a}_j}$.
This method will be referred to as the *Revised Simplex method*. It is usually presented in what is known as tableau form, however we will focus on matrix representation to highlight how the improvements are made.

We will introduce a new vector $\mathbf{y} = \mathbf{c}_B^T B^{-1}$, which will help shorten our notation.

Revised Simplex method

1. Solve for the simplex multiplier $\mathbf{y}B = \mathbf{c}_B$
2. Select the entering basic variables using reduced cost.
3. Solve $\bar{\mathbf{a}_j} = B^{-1}A_j$
4. Find the leaving basic variables using minimum ratio test.
5. Update the basis matrix B, $\mathbf{c}_N, \mathbf{c}_B$ and $\mathbf{x}_N, \mathbf{x}_B$

Computational expense of the inverse

Normally, we will find B^{-1} several times to solve our problem. However, finding the inverse of a matrix computationally is quite costly, which will lead to a large amount of computing time. Some methods make use of the relative ease in finding inverses for sparse matrices.

Definition 9.8: Sparse matrix

A *sparse matrix* is a matrix where most of the elements are zero. While there isn't a strict threshold for the number of zero-valued elements in sparse matrix, a common criterion is that the number of non-zero elements should be approximately equal to or less than the number of rows or columns.

$$A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

Some common sparse matrices that we encounter are triangular matrices, elementary matrices or diagonal matrices. In contrast to sparse matrices, we can also have dense matrices:

Definition 9.9: Dense matrix

A *dense matrix* is one where most of the elements are non-zero. For example

$$B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Dense matrices require more memory and computational power because each element, regardless of its size, must be processed. In the context of the Simplex method, the matrix B can be considered dense, making it challenging to invert efficiently. Conversely, sparse matrices, which contain a significant number of zero elements, are easier to handle and calculate. The computation of the inverse of a sparse matrix is typically faster and more efficient than that of a dense matrix because algorithms can ignore zero entries, reducing both memory usage and computational time. Moreover, techniques like LU updates and ETAS (elementary transformation of a sparse matrix) factorisation further enhance the computational efficiency by allowing for updates to the factorization of a matrix without recalculating it entirely. This is particularly advantageous in large-scale optimization problems, where maintaining and updating sparse structures enables quicker solutions without the overhead associated with dense matrix operations.

Etas factorization

This method makes use of elementary matrices (in this case we refer to them as *etas matrices*). Let B_k denote the basis matrix after k iterations [7].

Recall that the basis matrix B is an invertible $m \times m$ square matrix formed by selecting m linearly independent columns of A, corresponding with the *set of basic variables* $\mathbf{x}_B \geq 0$.

Now let us consider the basis matrix B after k iterations, $\mathbf{B_k}$ and the new entering column A_j from $\bar{\mathbf{a}_j} = B^{-1}A_j$. We will have:

$$B_k = B_{k-1}E_k$$

where E_k is the elementary matrix that performs the replacement of A_j. If we extend to change from B_0, we will have the following:

$$B_k = B_0 E_1 E_2 \dots E_k$$

Since $B_0 = I$ at the beginning this can be shortened to:

$$B_k = E_1 E_2 \dots E_k$$

If we pass this to $\mathbf{y}B = \mathbf{c}_B^T$ and $B\bar{\mathbf{a}_j} = A_j$, we will find that:

$$(((\mathbf{y}E_1)E_2)\dots)E_k = \mathbf{c}_B^T, \quad (((E_1\bar{\mathbf{a}_j})E_2)\dots)E_k = A_j$$

One efficient way to solve this is to use the the temporary vectors $\mathbf{u}, \mathbf{w}, \mathbf{v}$... to denote each step, for example, we will have:

$$\mathbf{u} = \mathbf{y}E_1, \quad \mathbf{w} = \mathbf{u}E_2, \quad \mathbf{v} = \mathbf{w}E_3$$

To prove the efficiency of the method, we will continue Example 9.1 above.

Example 9.2: Solving with Etas Factorization

$$\text{Maximize} \quad z = 3x_1 + 2x_2 + 4x_3$$

$$\begin{aligned} \text{s.t.} \qquad 2x_1 + x_2 + x_3 &\leq 4 \\ x_1 + 3x_2 + 2x_3 &\leq 6 \\ x_1 + 2x_2 + 3x_3 &\leq 5 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

First we will convert to standard form:

$$A = \begin{bmatrix} 2 & 1 & 1 & 1 & 0 & 0 \\ 1 & 3 & 2 & 0 & 1 & 0 \\ 1 & 2 & 3 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 4 \\ 6 \\ 5 \end{bmatrix}$$

$$\mathbf{x}_B = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix}, \quad \mathbf{x}_N = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \mathbf{c}_B = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \quad \mathbf{c}_N = \begin{bmatrix} 3 \\ 2 \\ 4 \end{bmatrix}$$

Now let us calculate the system:

$$\mathbf{y} = \mathbf{c}_B^T B_0^{-1} = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}$$

then x_3 will leave and s_3 will enter, as with the example above.
Updating some components:

$$\mathbf{x}_B = \begin{bmatrix} s_1 \\ s_2 \\ x_3 \end{bmatrix}, \quad \mathbf{x}_N = \begin{bmatrix} x_1 \\ x_2 \\ s_3 \end{bmatrix}, \quad \mathbf{c}_B = \begin{bmatrix} 0 \\ 0 \\ 4 \end{bmatrix}, \quad \mathbf{c}_N = \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix}$$

$$B_1 = B_0 E_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

In this, $\mathbf{E_1}$ is the matrix that performs the replacement of columns in $\mathbf{B}$. **Iteration 2.**

We will continue to find $\mathbf{y}$, but in this case instead of solving directly, we will use

$$\mathbf{y}B_1 = \mathbf{c}_B^T \implies \mathbf{y} = \mathbf{x}_B^T E_1^{-1}$$

By solving the inverse elementary matrix, we will reduce the computing time:

$$\mathbf{y} = \begin{bmatrix} 0 & 0 & 4 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1/3 \\ 0 & 1 & -2/3 \\ 0 & 0 & 1/3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \frac{4}{3} \end{bmatrix}$$

Now we can calculate the reduced cost as:

$$\mathbf{c}_B^T B^{-1} N - \mathbf{c}_N = \begin{bmatrix} 0 & 0 & \frac{4}{3} \end{bmatrix} \begin{bmatrix} 2 & 1 & 0 \\ 1 & 3 & 0 \\ 1 & 2 & 1 \end{bmatrix} - \begin{bmatrix} 3 & 2 & 0 \end{bmatrix} = \begin{bmatrix} -\frac{5}{3} & \frac{2}{3} & \frac{4}{3} \end{bmatrix}$$

Note that the solution is the same as we calculated previously. Now we have to find $\bar{\mathbf{a}_j}$, from $B\bar{\mathbf{a}_j} = A_j$:

$$B_1\bar{a_1} = A_1 \implies \bar{a_1} = E_1^{-1}A_1$$

we can find $\bar{a_1} = \begin{bmatrix} 5/3 \\ 1/3 \\ 1/3 \end{bmatrix}$ From that we can do the minimum ratio test and know that the x_1 will enter and s_1 leave. The rest of the example will reproduce the same solution as Example 9.1.

The *etas factorisation* approach will decompose the basis matrix into a sequence of elementary matrices, which are sparse matrices from which we can easily find the inverse and the needed variables.

LU factorization

Another way to improve the efficiency of the simplex method is to use the LU decomposition approach [5]. This will improve the computation time since in LU we introduce *lower triangular* and *upper triangular* matrices that are also considered **sparse matrices**.

Theorem 9.4: LU factorization

If A can be reduced to a row-echelon matrix U, then

$$A = LU \tag{9.1}$$

where:

- $L = E_1^{-1}E_2^{-1}...E_{k-1}^{-1}E_k^{-1}$ is a lower triangular and invertible matrix
- $U = E_k E_{k-1}...E_2 E_1 A$ is an upper triangular row-echelon matrix.

The reason for using LU decomposition is that we can apply *back substitution* and *forward substitution* to solve for each variable instead of using an inverse matrix. With these methods, we can solve $A\mathbf{x} = \mathbf{b}$ by applying these steps:

- Replace A with LU to obtain $LU\mathbf{x} = \mathbf{b}$

- Forward substitution: solve $L\mathbf{y} = \mathbf{b}$ by replacing $U\mathbf{x} = \mathbf{y}$.
- Back substitution: solve $U\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$.

LU factorisation in the simplex method

We can use LU factorisation to factorize the inverse B^{-1} in $\mathbf{y}B = \mathbf{c}_B^T$ and $B\bar{\mathbf{a}_j} = A_j$. Let's consider B_k at the k-th iteration and the new entering A_j from the previous iteration. Now, instead of solving B_k^{-1} directly, we can use LU factorisation to factorize $\mathbf{y}B = \mathbf{c}_B^T$, then we have:

$$\mathbf{y}LU = \mathbf{c}_B^T$$

$$\begin{aligned} \text{Forward substitution:} \quad & \mathbf{w}U = \mathbf{c}_B^T \\ \text{Back substitution:} \quad & \mathbf{y}L = \mathbf{w} \end{aligned}$$

This will work with $B\bar{\mathbf{a}_j} = A_j$, however there will be a slight difference:

$$LU\bar{\mathbf{a}_j} = A_j$$

$$\begin{aligned} \text{Forward substitution:} \quad & L\mathbf{w} = A_j \\ \text{Back substitution:} \quad & U\bar{\mathbf{a}_j} = \mathbf{w} \end{aligned}$$

After these calculations, we can continue the normal steps in the Revised Simplex Method from finding the leaving variables, then taking the feasible basic solution and updating the basis matrix B. However, instead of updating the whole matrix B, we will be updating the matrices in the LU factorisation.

The sparse Bartels-Golub Method

Another way to use LU decomposition is to apply it using the sparse Bartel-Golub method [8]. In this particular method, instead of recalculating the whole of matrices L and U, we will focus on updating them. This means we will avoid re-calculation, and in turn will reduce the complexity of the simplex method. Before discussing this method we will provide a definition for the *eta matrix*:

Definition 9.10: Eta matrix

The eta matrix is an elementary matrix that differs from the identity matrix in a single column. It helps adjust the basis inverse after introducing a new entering variable and removing a leaving variable. We can have a lower triangular eta matrix that modifies the p-th column with a non-zero entry η_{qp} at row q. The eta matrix is given by:

$$H_{qp} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & \cdots & \eta_{qp} & 1 \end{bmatrix}$$

We can construct the eta matrix with the leaving and entering variables as follows:

$$H_k = I + (\mathbf{v} - \mathbf{e}_q)\mathbf{e}_q^T$$

Where:

- I is the identity matrix,
- $\mathbf{v}$ is the update column vector,
- $\mathbf{e}_q$ is the unit vector with 1 in the q-th position (corresponding to the row being updated) and zero else where.

In the **sparse Bartels-Golub Method**, instead of decomposing the matrix B at each iteration, the lower-triangular matrix L is expressed as a product of eta matrices [4]. Since there is a sequence of eta matrices, the inverse of $\mathbf{L}$ is the inverse of these eta matrices. Suppose we have the following lower triangular matrix:

$$\mathbf{L} = \begin{bmatrix} 1 & \eta_{12} & \eta_{13} \\ 0 & 1 & \eta_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix can be expressed as:

$$L = H_{13} \cdot H_{23} \cdot H_{12} = \begin{bmatrix} 1 & 0 & \eta_{13} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & \eta_{23} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & \eta_{12} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Now, let us consider the basis matrix B in the update of upper triangular matrix U, which is expressed as [6]:

$$PUQ = M_r M_{r-1} \cdots M_1 B$$

where P and Q are permutation matrices, and U is the upper triangular matrix. To simplify this, we will omit the permutation matrices and perform the basis change. Recall that the basis change involves one column leaving while another column enters at the pivot position p. Let $\mathbf{a}_q$ denote the entering column and $\mathbf{e}_p$ denote the unit column. We then have:

$$\bar{B} = B + (\mathbf{a}_q - B\mathbf{e}_p)\mathbf{e}_p^T$$

If we apply the the LU decomposition to B, with $B = LU$, and multiplying both sides by the static matrix L^{-1}, we have:

$$L^{-1}\bar{B} = U + (L^{-1}\mathbf{a}_q - U\mathbf{e}_p)\mathbf{e}_p^T$$

Equation (9.4) can be understood as having a new basis matrix $\bar{B}$. The column p of $\mathbf{U}$ will be replaced by a vector $g = L^{-1}\mathbf{a}_q$ which is called the *spike* [3]. If we consider the upper triangular matrix:

$$\bar{U} = U + (L^{-1}\mathbf{a}_q - U\mathbf{e}_p)\mathbf{e}_p{}' = U + (\mathbf{g} - U\mathbf{e}_p)\mathbf{e}_p^T$$

then the new basis matrix can be expressed as:

$$\bar{B} = L\bar{U}$$

Example 9.3: Updates using the spike

Consider a 5×5 upper triangular matrix U with the entering vector $\mathbf{a}$ as shown below.

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ 0 & u_{22} & u_{23} & u_{24} & u_{25} \\ 0 & 0 & u_{33} & u_{34} & u_{35} \\ 0 & 0 & 0 & u_{44} & u_{45} \\ 0 & 0 & 0 & 0 & u_{55} \end{bmatrix}, \quad \mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ 0 \end{bmatrix}$$

The spike vector $\mathbf{a}$ will enter at the column 2 denoted as p, then the update will be performed as:

$$\begin{bmatrix} u_{11} & \mathbf{a_1} & u_{13} & u_{14} & u_{15} \\ 0 & \mathbf{a_2} & u_{23} & u_{24} & u_{25} \\ 0 & \mathbf{a_3} & u_{33} & u_{34} & u_{35} \\ 0 & \mathbf{a_4} & 0 & u_{44} & u_{45} \\ 0 & \mathbf{0} & 0 & 0 & u_{55} \end{bmatrix}$$

Now let us consider the column permutation matrix Q, which rearranges the columns so that the spike is in the correct position (at r position). By moving the spike to position r while shifting the other columns to the left by one to fill the gap.

$$\begin{bmatrix} u_{11} & u_{13} & u_{14} & \mathbf{a_1} & u_{15} \\ 0 & u_{23} & u_{24} & \mathbf{a_2} & u_{25} \\ 0 & u_{33} & u_{34} & \mathbf{a_3} & u_{35} \\ 0 & 0 & u_{44} & \mathbf{a_4} & u_{45} \\ 0 & 0 & 0 & \mathbf{0} & u_{55} \end{bmatrix}$$

After this we remove the redundant row p to obtain Upper Hessenberg matrix.

$$\begin{bmatrix} u_{11} & u_{13} & u_{14} & \mathbf{a_1} & u_{15} \\ - & - & - & \mathbf{a_2} & u_{25} \\ 0 & u_{33} & u_{34} & \mathbf{a_3} & u_{35} \\ 0 & 0 & u_{44} & \mathbf{a_4} & u_{45} \\ 0 & 0 & 0 & \mathbf{0} & u_{55} \end{bmatrix}$$

We then apply the row permutation matrix P to form a new matrix U.

$$\begin{bmatrix} u_{11} & u_{13} & u_{14} & \mathbf{a_1} & u_{15} \\ 0 & u_{33} & u_{34} & \mathbf{a_2} & u_{25} \\ 0 & 0 & u_{44} & \mathbf{a_3} & u_{35} \\ 0 & 0 & 0 & \mathbf{a_4} & u_{45} \\ 0 & 0 & 0 & \mathbf{0} & u_{55} \end{bmatrix}$$

Recall that the basis matrix can be written in terms of the new upper triangular matrix:

$$\bar{B} = L\bar{U}$$

Since the upper triangular matrix has been updated, we might have to update L with the suitable eta matrix, which corresponds with the operation work on rows and columns of upper triangular matrix U.

The procedure of updating upper triangular matrix can be summarised as follows.

LU Update Algorithm

1. A new spike $L^{-1}\mathbf{a}_q$ will be inserted into matrix U at position p. Then the old column at position p is removed from the matrix.
2. Identify the position r, which is the last non-zero element of the spike. Perform permutation matrix Q to shift the columns from p to r, where the old column is on the left.

3. Eliminate the value from p to $r-1$ of row p and store the multiplier factor in the matrix L.
4. The permuation matrix P used to perform the row permuation. This will shift the row $p+1...r$ upward to fill the gap after eliminating row p and restores the triangular structure of U.
5. Save the modified row (which was originally row p and is now in position r) back into the sparse representation of U and create the eta matrices for L
6. Update the basis matrix:
$$\bar{B} = L\bar{U}$$
where $\bar{U}$ is the updated upper triangular matrix.
7. Return the modified lower triangular matrix $\bar{L}$, upper triangular matrix $\bar{U}$, and updated basis matrix $\bar{B}$.

Revised Simplex method using LU update

Now it's time to gather all of the knowledge above to revise the simplex method. In the Revised Simplex method, we will first initialize the basis matrix by using LU decomposition to obtain:

$$PBQ = LU$$

With the basis matrix, we can easily find the current basis solution as:

$$\mathbf{x}_B = U^{-1}L^{-1}\mathbf{b}$$

And we can calculate the weight $\mathbf{y}$ which is the BTRAN using LU decomposition:

$$\mathbf{y}LU = \mathbf{c}_B^T$$

From that we can find the reduced cost of all non-basis variables and choose the most negative one for the entering variable:

$$\mathbf{c}_B^T B^{-1}N - \mathbf{c}_N$$

In this stage if the reduced costs are all positive then it is the optimal solution, otherwise we will find the leaving variables by finding $\bar{a}$. This step can be called FTRAN:

$$LU\bar{\mathbf{a}_j} = A_j$$

After that we can use the minimum ratio test to find the leaving variable. Finally after we complete and find the feasible basic solution, we will update U^{-1} and repeat the steps.

Note: LU Updates are crucial for enhancing the accuracy and efficiency of the Simplex method, particularly in large-scale linear programming problems. By focusing on sparse matrices, LU Updates enable faster iterative computations, allowing the algorithm to maintain a triangular structure without recalculating the entire factorization after each iteration. This not only reduces computational overhead but also enhances numerical stability, minimizing the risk of errors during calculations. Furthermore, the use of sparse matrix representations improves memory efficiency, making LU Updates an essential tool for optimizing the Simplex method in complex problem-solving scenarios.

A comparison of the revised simplex method with the revised simplex method using the Bartels-Golub method is summarised in Table 9.1.

Table 9.1: Comparison of Revised Simplex Method and Sparse Bartels-Golub Method [4]

Aspect	Revised Simplex Method	Sparse Bartels-Golub Method
Basis Matrix Updates	Recomputes the inverse of the basis matrix B^{-1} at each iteration.	Updates the LU factorization especially the matrix U^{-1}.
Handling Sparsity	Doesn't explicitly handle sparse matrices efficiently.	Explicitly exploits the sparse structure, updating only non-zero elements.
Computational Efficiency	$O(m^3)$ due to recomputing the inverse.	Much lower complexity, $O(m^2)$, with sparse matrices, focusing only on LU updates.
Memory Usage	High memory usage for storing dense inverse matrices.	Efficient memory usage, especially for large, sparse systems.
Numerical Stability	May suffer from numerical instability when recalculating basis matrix.	LU factorization is numerically more stable, particularly for large systems.

9.5 Summary

In summary, while traditional implementations of the simplex method can be computationally expensive, particularly for large-scale problems, the integration of linear algebra techniques significantly enhances its performance. Specifically, LU factorization decomposes the matrix of constraints into simpler triangular forms, allowing for quicker updates of the basis matrix while also improving numerical stability. The Sparse Bartels-Golub method further refines this process by leveraging the sparse structure of many real-world problems, which reduces unnecessary computations and accelerates the optimization process. By incorporating LU updates within the framework of the Sparse Bartels-Golub method, the simplex method becomes more efficient and scalable, enabling it to tackle complex linear programming challenges effectively. This highlights the critical role of linear algebra in optimizing solutions.

Context

This article was adapted from a report submitted from SIT292 - Linear Algebra for Data Analysis

About the Author

I am currently pursuing a Bachelor's degree in Artificial Intelligence with Honors. My passion lies in understanding AI concepts, which requires a strong foundation in mathematics, particularly Linear Algebra and Optimization. This interest has driven me to explore these mathematical fields in depth, allowing me to develop a deeper understanding of their principles and build a solid foundation for comprehending AI at a fundamental level.

Acknowledgements

I would like to express my gratitude to Simon James for providing me with the opportunity to independently explore mathematical concepts, which has greatly enhanced my problem-solving abilities and self-study skills. I am also deeply thankful to Gleb Beliakov for his insightful feedback, which has helped me gain a deeper understanding of linear algebra and linear programming.

References

[1] Christopher Griffin. Linear Programming: Penn State Math 484 Lecture Notes, version 1.8.3.1. *Penn State University*, 2014.

[2] Roy H Kwon. *Introduction to Linear Optimization and Extensions with MATLAB*. CRC Press, New York, 2014.

[3] István Maros. *Computational Techniques of the Simplex Method, International Series in Operations Research & Management Science: 61*. Springer, 2003.

[4] Steven S. Morgan. *A Comparison of Simplex Method Algorithms [Master Thesis]*. University of Florida, 1997.

[5] W. K. Nicholson. *Linear Algebra with Applications*. Lyryx, Revised A edition, 2021.

[6] J. K. Reid. A sparsity-exploiting variant of the Bartels—Golub decomposition for linear programming bases. *Mathematical Programming*, 24:55–69, 1982.

[7] Vašek Chvátal. *Linear programming*. Freeman, New York, 1983.

[8] Leena M. Suhl and Uwe H. Suhl. A fast LU update for linear programming. *Annals of Operations Research*, 43:33–47, 1993.

[9] Robert J Vanderbei. *Linear programming : foundations and extensions*. Springer Nature, Switzerland, 2020.

10

An application of linear algebra to robotics

Vinh Nguyen

Abstract

The report provides an insight into an application of linear algebra in solving the forward kinematics problem in robotics, where a robotic system's position is computed given the system's pose and characteristics. This is introduced in depth with preliminary concepts related to three-dimensional point and coordinate frame transformations using matrix algebra, from which approaches to solving the forward kinematics problem are provided. This includes a simple and inefficient 7-parameter approach, as well as the more efficient and commonly used Denavit-Hartenberg approach.

10.1 Introduction

Mechatronical automations, especially robotic systems, operate in a *sense-think-act* loop, where the robotic system continually senses its surroundings and circumstances, plans its next action, then performs the action. In many cases, this requires the robot to know the physical locations of its own parts; for example, a robotic manipulator needs to know where the tool mounted to its end (i.e. its *end effector*) is and how it is angled with respect to its workspace such as an assembly line.

The calculations to result in this data from information related to the robot's actuators are called *forward kinematics*, which is rooted from point and coordinate frame transformations in three-dimensional (3D) space. In this report, the basis of 3D point and frame transformations will be discussed, and from which forward kinematics will be described.

This report aims to be an introduction to robotics from a mathematical standpoint through uncovering the connections between linear algebra, mechanical engineering, and computing, as well as an introduction to an application of linear algebra outside the traditionally-associated fields of data science and machine learning.

10.2 The three-dimensional space

It is impossible to describe forward kinematics without first describing the mathematical representation of the universe we live in. As we may know, the physical space in our universe is three-dimensional; this concept is defined mathematically in Definition 10.1 below.

Definition 10.1: Three-dimensional space

A *three-dimensional* (*3D*) space is a mathematical space in which a point's position is defined by a 3-tuple

$$(x, y, z)$$

(the point's *coordinates*), with the values corresponding to the point's projections onto three perpendicular axes x, y and z respectively.

These axes form a *coordinate frame* (or simply *frame*) based on which the point's position is given [9].

There are two fundamentally incompatible ways of arranging these axes, which are referred to as *left-handed* and *right-handed* coordinate frames. However, the latter is more common in robotics and will therefore be used throughout the report. The right-hand coordinate frame, shown in Figure 10.1, can be easily determined using the *right-hand rule for cross product*: using the right hand, one can point their index finger away from them, their middle finger to their left, and their thumb upward; these fingers point in the directions of the x, y and z axes respectively.

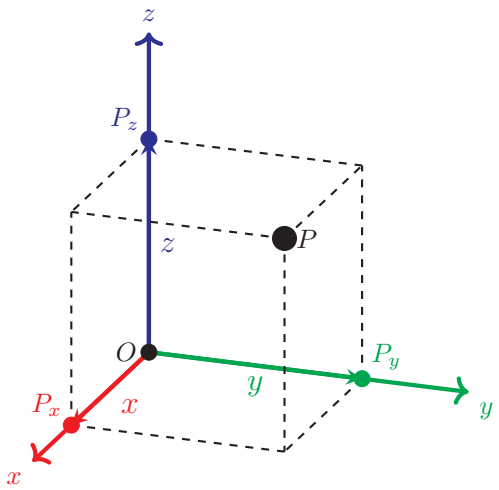


Figure 10.1: The right-hand 3D coordinate frame, with a point P and its projections P_x, P_y and P_z on the x, y and z axes respectively.

Definition 10.2 describes a point coordinates representation commonly used in matrix algebra.

Definition 10.2: Homogeneous coordinate vectors

The coordinates (x, y, z) can be rewritten in the *homogeneous coordinate vector* form as the 4-vector [8]

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

The 1 entry at the end of the homogeneous coordinate vector provides a constant that can be multiplied with to transform the vector. This makes use of the matrix multiplication rule where,

for instance,

$$\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \end{bmatrix}$$

If $b_n = 1$, as with the case of homogeneous coordinate vectors, a_n will become an offset that is added to the result; in mathematical terms:

$$\begin{bmatrix} a_1 & a_2 & \cdots & a_{n-1} & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 b_1 + a_2 b_2 + \cdots + a_{n-1} b_{n-1} + a_n \end{bmatrix}$$

This will be useful in some transformations, as will be seen later in the report (particularly Proof 10.1 and Example 10.1).

10.3 3D rigid transformations

At its core, forward kinematics are merely a series of transformations, particularly *rigid transformations*, which are defined below.

Definition 10.3: Rigid transformations

A *rigid transformation* is a transformation (movement) of one or more points that preserves distances between any two points transformed by it [15].

Proposition 10.1: Rigid transformations as matrices

With the homogeneous coordinate vector representation defined in Definition 10.2, any rigid transformation can be defined as a 4×4 transformation matrix on such vectors in the form

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The last row of the transformation matrix is always $\begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$ in order to preserve the 1 entry at the end of homogeneous coordinate vectors; the last entry in the transformed vector will be computed as $0 \times x + 0 \times y + 0 \times z + 1 \times 1 = 1$.

There are two basic types of rigid transformations, *translation* and *rotation* [19], as well as *reflection*. These transformations are described below.

Translation

Definition 10.4: Translation by (d_x, d_y, d_z)

The *translation* by (d_x, d_y, d_z) linearly moves a point in 3D space by d_x, d_y and d_z along the x, y and z axes respectively.

As mentioned above, it is possible to define this translation as a transformation matrix; this is given in the below theorem.

Theorem 10.1: Translation as a matrix transformation

A translation by (d_x, d_y, d_z) can be performed using the transformation matrix

$$T(d_x, d_y, d_z) = \begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Proof for Theorem 10.1

Suppose that a point (x, y, z) is to be translated by (d_x, d_y, d_z). The original point's homogeneous coordinate vector is

$$A = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and the transformation matrix for this vector is

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The result of this transformation is

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \times x + 0 \times y + 0 \times z + d_x \times 1 \\ 0 \times x + 1 \times y + 0 \times z + d_y \times 1 \\ 0 \times x + 0 \times y + 1 \times z + d_z \times 1 \\ 0 \times x + 0 \times y + 0 \times z + 1 \times 1 \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \\ z + d_z \\ 1 \end{bmatrix}$$

This is the mathematical form of the statement given in Definition 10.4, and the proof is thus complete.

Later on in this report, we will also use the following short forms for translations along the coordinate frame axes:

$$\begin{cases} T_x(x) = T(x, 0, 0) \\ T_y(y) = T(0, y, 0) \\ T_z(z) = T(0, 0, z) \end{cases}$$

To determine the inverse transformation for translation, note that by definition, an inverse transformation induced by the matrix M^{-1} reverts the forward transformation M (as $MM^{-1} = I$). We thus have the following proposition:

Proposition 10.2: Inverse transformation for translations

The inverse of the translation $T(d_x, d_y, d_z)$ is $T(-d_x, -d_y, -d_z)$.

Example 10.1 below demonstrates the translation transformation.

Example 10.1: Translation of a point

The point $A(2, 2, 2)$ is written in homogeneous coordinate vector representation as:

$$A = \begin{bmatrix} 2 \\ 2 \\ 2 \\ 1 \end{bmatrix}$$

To translate A by $(-1, 1, 0)$, the translation matrix is first formed:

$$T(-1, 1, 0) = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

They can then be multiplied together to compute the coordinates for the transformed point B:

$$\begin{aligned} B &= T(-1, 1, 0)A \\ &= \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \times 2 + 0 \times 2 + 0 \times 2 + (-1) \times 1 \\ 0 \times 2 + 1 \times 2 + 0 \times 2 + 1 \times 1 \\ 0 \times 2 + 0 \times 2 + 1 \times 2 + 0 \times 1 \\ 0 \times 2 + 0 \times 2 + 0 \times 2 + 1 \times 1 \end{bmatrix} = \begin{bmatrix} 2 - 1 \\ 2 + 1 \\ 2 + 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} \end{aligned}$$

B thus has the coordinates $(1, 3, 2)$. Points A and B are shown in Figure 10.2.

Rotation

Unlike in 2D planes (where a rotation is defined only by its angle and a centre to rotate by [11]), rotations in 3D space are defined not only by the angle of rotation, but also the axis. Thus there are three *principal rotation axes* and their corresponding transformation matrices, which are given in Theorem 10.2.

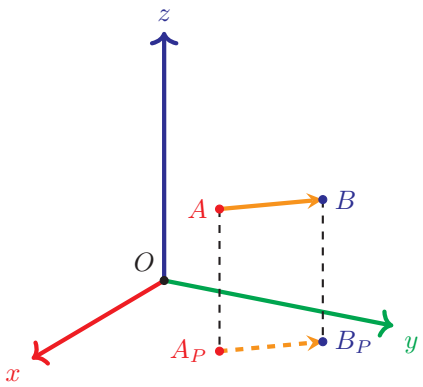


Figure 10.2: Points A (the original point) in red, and B (the translated point) in blue. An orange vector is also shown to illustrate the translation, as well as the projections A_P and B_P of the points onto the xy plane.

Theorem 10.2: Extrinsic rotations along principal axes

A point can be rotated along the following *principal axes*: [8]

- x axis by the angle φ with the transformation matrix

$$R_x(\varphi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- y axis by the angle θ with the transformation matrix

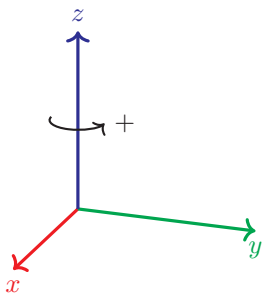
$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- z axis by the angle ψ with the transformation matrix

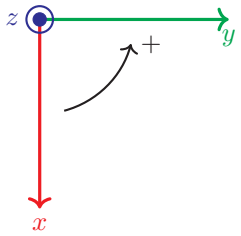
$$R_z(\psi) = \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The angles' positive direction is determined using the *right-hand convention* (not to be confused with the right-hand rule introduced in Section 10.2). When making a "thumbs-up" gesture with the right hand, the thumb points in the direction of an axis, while the other fingers curl in the positive rotation direction. This is similar to rotations in the 2D plane, where the positive direction is counterclockwise; the 2D plane is merely an xy plane, with the z axis pointing out of the page (Figure 10.3).

Note the term *extrinsic* used in the title of Theorem 10.2. There are two forms of rotations that are used: *extrinsic*, where the given rotations are with respect to axes in a fixed frame, so that these rotation axes do not change as the point or coordinate frame is transformed, and *intrinsic*, where the rotation axes are bound to the coordinate frame being transformed and thus change



(a) Positive direction of rotation in 3D space.



(b) Positive direction of rotation in 2D space.

Figure 10.3: Comparison between rotations in 2D and 3D.

after the transformation. Due to its simplicity, however, extrinsic rotations are more frequently used, and are usually assumed; this report will cover extrinsic rotations exclusively.

These 3D rotations are merely 2D rotations performed on planes parallel to the non-rotational axes. This is shown in Proof 10.2, which acknowledges the trivial 2D rotation matrix for rotations by α about the origin shown below:

$$R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

Proof for Theorem 10.2

Suppose the point to be rotated is $P(x, y, z)$. Let $P_x(0, y, z)$, $P_y(x, 0, z)$ and $P_z(x, y, 0)$ be P's projections onto the yz, xz and xy planes respectively (Figure 10.4).

It is obvious that as P rotates about the x, y and the z axes, P_x, P_y and P_z also rotate with the same angle and direction, and P's x, y and z coordinates remain unaffected respectively. We can therefore consider the 2D rotations on the yz, xz and xy planes respectively for these rotations.

Let us generalise the 2D rotation matrix for an arbitrary (i.e. non-xy) Cartesian coordinates plane. In the standard 2D plane, the x and y axes are arranged in such a way that x axis rotated by $\frac{\pi}{2}$ aligns with the y axis; we can generalise this into axes A and B such that A rotated by $\frac{\pi}{2}$ aligns with B.

Given the (non-homogeneous) coordinate vector $\begin{bmatrix} a \\ b \end{bmatrix}$ for a point, where a and b are the point's coordinates with respect to A and B respectively, we have the following rotation matrix for rotating the point about the origin by α:

$$\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$

For the rotation of P about the x axis by φ, consider the rotation of P_x in the yz plane. By our definition above, the y and z axes match with A and B respectively, and thus we have the following transformation:

$$\begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} y\cos\varphi - z\sin\varphi \\ y\sin\varphi + z\cos\varphi \end{bmatrix}$$

According to the matrix multiplication rule, along with Proposition 10.1, we can assemble the following transformation matrix for the corresponding 3D rotation:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R_x(\varphi)$$

Note that the row corresponding to x is $\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$; this "copies" the original x coordinate to the transformed x coordinate unchanged, as stated above. Similarly, for the rotation about the y axis by θ, consider the rotation of P_y in the xz plane, where the z and x axes match with A and B respectively. We thus have the following transformation:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} z \\ x \end{bmatrix} = \begin{bmatrix} z\cos\theta - x\sin\theta \\ z\sin\theta + x\cos\theta \end{bmatrix} = \begin{bmatrix} -x\sin\theta + z\cos\theta \\ x\cos\theta + z\sin\theta \end{bmatrix}$$

The corresponding 3D rotation matrix can then be assembled as follows:

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R_y(\theta)$$

Finally, for the rotation about the z axis by ψ, consider the rotation of P_z in the xy plane. This time, x and y matches with the A and B axes, and we have the following transformation:

$$\begin{bmatrix} \cos\psi & -\sin\psi \\ \sin\psi & \cos\psi \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

This can be adapted into the following 3D transformation matrix:

$$\begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = R_z(\psi)$$

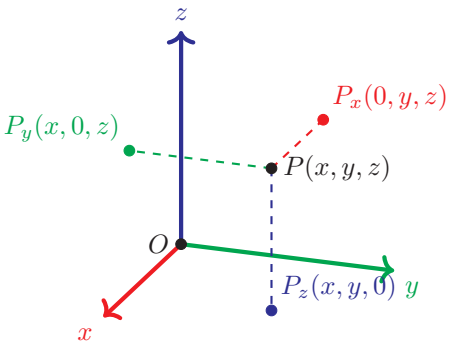


Figure 10.4: Projections of the point P on different planes in the 3D coordinate frame.

These principal rotations are usually combined together; the rotation angles define the point's *orientation*, and are referred to by terms described in the following definition, which are borrowed from aeronautics:

Definition 10.5: Principal axes of rotation

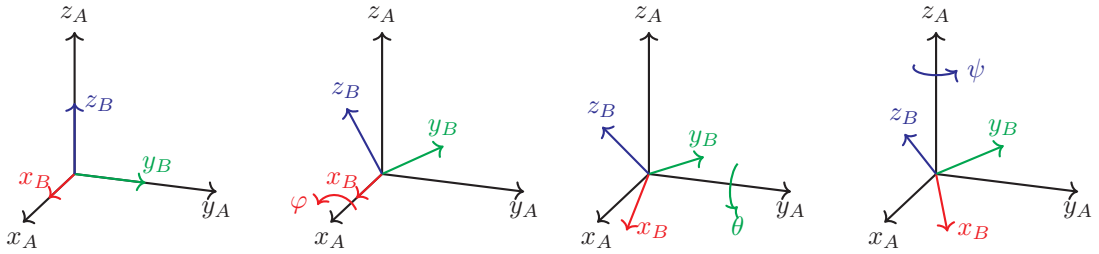
The angles of rotation φ (about the x axis), θ (about the y axis), and ψ (about the z axis) are also referred to as *roll*, *pitch*, and *yaw* respectively [1].

There are multiple ways to combine these rotations; however, the most common way is to perform the x-rotation, followed by the y-rotation, and finally the z-rotation, which is also called the xyz order (Figure 10.5).

Proposition 10.3: Rotation in the xyz order

The transformation matrix that performs a rotation according to the *orientation tuple* (φ, θ, ψ) (corresponding to the roll, pitch and yaw respectively) is

$$
\begin{aligned}
R(\varphi,\theta,\psi) &= R_Z(\psi)R_Y(\theta)R_X(\varphi) \\
&= \begin{bmatrix}
\cos\psi\cos\theta & \cos\psi\sin\theta\sin\varphi - \sin\psi\cos\varphi & \cos\psi\sin\theta\cos\varphi + \sin\psi\sin\varphi & 0 \\
\sin\psi\cos\theta & \sin\psi\sin\theta\sin\varphi + \cos\psi\cos\varphi & \sin\psi\sin\theta\cos\varphi - \cos\psi\sin\varphi & 0 \\
-\sin\theta & \cos\theta\sin\varphi & \cos\theta\cos\varphi & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\end{aligned}
$$



(a) B is initially coincident with A.

(b) B is first rotated about A's x axis.

(c) It is then rotated about A's y axis.

(d) Finally, it is rotated about A's z axis.

Figure 10.5: A step-by-step rotation process following the xyz order. In this process, frame A is rotated to form frame B.

Using the notion that a transformation is reverted by its inverse, we also arrive at the following proposition.

Proposition 10.4: Inverse transformation for principal axis rotations

The inverse of a rotation about a principal (i.e. x, y or z) axis by α is the rotation about the same axis by $-\alpha$.

However, for the composite rotation given in Proposition 10.3, the rules of matrix inversion result in a different matrix altogether, as shown in the following theorem:

Theorem 10.3: Inverse transformation for xyz-order rotations

The inverse of a rotation according to the orientation (φ, θ, ψ) is

$$R^{-1}(\varphi,\theta,\psi) = R_X(-\varphi)R_Y(-\theta)R_Z(-\psi)$$
$$= \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta & 0 \\ -\cos\varphi\sin\psi - \sin\varphi\sin\theta\sin\psi & \cos\varphi\cos\psi + \sin\varphi\sin\theta\sin\psi & -\sin\varphi\sin\theta & 0 \\ \sin\varphi\sin\psi + \cos\varphi\sin\theta\cos\psi & \sin\varphi\sin\psi + \cos\varphi\sin\theta\sin\psi & \cos\varphi\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Proof for Theorem 10.3

As stated in Proposition 10.3, the rotation matrix for the orientation (φ, θ, ψ) is

$$R(\varphi,\theta,\psi) = R_Z(\psi)R_Y(\theta)R_X(\varphi)$$

Therefore, the inverse of this matrix is

$$R^{-1}(\varphi,\theta,\psi) = R_X^{-1}(\varphi)R_Y^{-1}(\theta)R_Z^{-1}(\psi)$$

Substituting the inverses as described in Proposition 10.4, we have [2]

$$R^{-1}(\varphi,\theta,\psi) = R_X(-\varphi)R_Y(-\theta)R_Z(-\psi)$$
$$= \begin{bmatrix} c_\theta c_\psi & -c_\theta s_\psi & s_\theta & 0 \\ c_\varphi s_\psi + s_\varphi s_\theta s_\psi & c_\varphi c_\psi - s_\varphi s_\theta s_\psi & -s_\varphi s_\theta & 0 \\ s_\varphi s_\psi - c_\varphi s_\theta c_\psi & s_\varphi s_\psi + c_\varphi s_\theta s_\psi & c_\varphi c_\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $s_\alpha = \sin(-\alpha)$ and $c_\alpha = \cos(-\alpha)$ for an angle α.

According to trigonometric identities, we have

$$\begin{cases} s_\alpha = \sin(-\alpha) = -\sin\alpha \\ c_\alpha = \cos(-\alpha) = \cos\alpha \end{cases}$$

which we can then substitute to obtain

$$R^{-1}(\varphi,\theta,\psi)$$
$$= \begin{bmatrix} \cos\theta\cos\psi & \cos\theta\sin\psi & -\sin\theta & 0 \\ -\cos\varphi\sin\psi - \sin\varphi\sin\theta\sin\psi & \cos\varphi\cos\psi + \sin\varphi\sin\theta\sin\psi & -\sin\varphi\sin\theta & 0 \\ \sin\varphi\sin\psi + \cos\varphi\sin\theta\cos\psi & \sin\varphi\sin\psi + \cos\varphi\sin\theta\sin\psi & \cos\varphi\cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note that the order of rotation is reversed to zyx for the inverse transformation. This results in a different form for the inverse rotation matrix compared to its forward counterpart in Proposition 10.3.

Reflection

Definition 10.6: Reflection in 3D space

In 3D space, a point can be *reflected* through a plane; this results in the *image* of the point on the other side of the plane.

With this definition, the following transformation matrices for reflecting through the xy, yz and xz planes can be defined by simply negating the coordinates on the other axes.

Proposition 10.5: Transformation matrices for reflections

The transformation matrices for reflection of a point through planes formed by coordinate frame axes are: [10]

- yz plane: $R_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- xz plane: $R_{xz} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- xy plane: $R_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Another consequence of this statement is:

Proposition 10.6: Inverse transformation for reflections

The inverse matrix for a reflection transformation matrix is itself.

Reflections are generally not used in robotics, and are only introduced in this section for mathematical completeness.

10.4 Coordinate frames

As stated in Definition 10.1, a *coordinate frame* (or simple *frame*) is the combination of the x, y and z axes, which forms the frame of reference for defining a point's position. In robotics, multiple coordinate frames are involved; usually, there is a *world frame* which is stationary (and is thus often set to a stationary point; e.g. the robotic manipulator's mounting point on the workspace) [12], and one or more frames that are displaced relative to the world frame. The relationship between one frame and another is quantified as the frame's *pose*, which is defined below.

Definition 10.7: Pose of a coordinate frame

A coordinate frame can be formed from another *parent* frame by rotating the parent frame using the orientation (φ, θ, ψ) and then translating it by the origin offset (x, y, z).
In this case, the newly-created frame has a *pose* consisting of the origin coordinates (x, y, z) and the orientation (φ, θ, ψ) relative to its parent frame.

Following this definition, the transformation matrix for aligning the parent frame to its child can be written as follows:

Proposition 10.7: Coordinate frame transformation matrix

Given a frame B whose pose consists of the origin coordinates (x, y, z) and the orientation (φ, θ, ψ) relative to its parent frame A, the transformation matrix aligning A to B is

$$\begin{aligned} F_A^B &= T(x,y,z)R(\varphi,\theta,\psi) \\ &= \begin{bmatrix} \cos\psi\cos\theta & \cos\psi\sin\theta\sin\varphi - \sin\psi\cos\varphi & \cos\psi\sin\theta\cos\varphi + \sin\psi\sin\varphi & x \\ \sin\psi\cos\theta & \sin\psi\sin\theta\sin\varphi + \cos\psi\cos\varphi & \sin\psi\sin\theta\cos\varphi - \cos\psi\sin\varphi & y \\ -\sin\theta & \cos\theta\sin\varphi & \cos\theta\cos\varphi & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

The transformation matrix aligning B to A is simply the above matrix's inverse:

$$F_B^A = [F_A^B]^{-1} = R^{-1}(\varphi,\theta,\psi)T(-x,-y,-z)$$

Another way of interpreting the frame transformation matrix, which can be useful in certain scenarios (such as plotting the coordinate frames onto a 3D plot), is shown in the following theorem.

Theorem 10.4: Alternate interpretation of the coordinate frame transformation matrix

The transformation matrix aligning a parent frame A to its child B has the form

$$F_A^B = \begin{bmatrix} x_1 - x_0 & x_2 - x_0 & x_3 - x_0 & x_0 \\ y_1 - y_0 & y_2 - y_0 & y_3 - y_0 & y_0 \\ z_1 - z_0 & z_2 - z_0 & z_3 - z_0 & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where (x_0, y_0, z_0), (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) are coordinates of B's origin and x, y and z-axis unit vector (i.e. vector aligning with axis and with length of 1) end points in the A frame respectively.

Proof for Theorem 10.4

It is obvious that the last column corresponds with B's origin coordinates; this is explicitly shown in Proposition 10.7 above. The x, y and z-axis unit vector end points have the coordinates $(1, 0, 0)$, $(0, 1, 0)$ and $(0, 0, 1)$ respectively. As the F_A^B matrix defines the transformation of frame A's axes to those of frame B, we can also use it to find the locations of those points as they are moved from A to B.

With the matrix in the form

$$F_A^B = \begin{bmatrix} r_{11} & r_{12} & r_{13} & x_0 \\ r_{21} & r_{22} & r_{23} & y_0 \\ r_{31} & r_{32} & r_{33} & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

we have the following transformations for:

- x axis unit vector end point: $\begin{bmatrix} r_{11} & r_{12} & r_{13} & x_0 \\ r_{21} & r_{22} & r_{23} & y_0 \\ r_{31} & r_{32} & r_{33} & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} + x_0 \\ r_{21} + y_0 \\ r_{31} + z_0 \\ 1 \end{bmatrix}$
- y axis unit vector end point: $\begin{bmatrix} r_{11} & r_{12} & r_{13} & x_0 \\ r_{21} & r_{22} & r_{23} & y_0 \\ r_{31} & r_{32} & r_{33} & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} r_{12} + x_0 \\ r_{22} + y_0 \\ r_{32} + z_0 \\ 1 \end{bmatrix}$
- z axis unit vector end point: $\begin{bmatrix} r_{11} & r_{12} & r_{13} & x_0 \\ r_{21} & r_{22} & r_{23} & y_0 \\ r_{31} & r_{32} & r_{33} & z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} r_{13} + x_0 \\ r_{23} + y_0 \\ r_{33} + z_0 \\ 1 \end{bmatrix}$

The coordinates for those points of frame B in frame A are therefore:

$$\begin{cases} (x_1, y_1, z_1) = (r_{11} + x_0, r_{21} + y_0, r_{31} + z_0) \\ (x_2, y_2, z_2) = (r_{12} + x_0, r_{22} + y_0, r_{32} + z_0) \\ (x_3, y_3, z_3) = (r_{13} + x_0, r_{23} + y_0, r_{33} + z_0) \end{cases}$$

We can then move x_0, y_0 and z_0 in the above statements to the left-hand side to retrieve the expressions for each of the r_{ji} entries:

$$\begin{cases} r_{1i} = x_i - x_0 \\ r_{2i} = y_i - y_0 \\ r_{3i} = z_i - z_0 \end{cases} \quad \text{for } i \in \{1, 2, 3\}$$

This results in the matrix in Theorem 10.4, and the proof is therefore complete.

In the proof above, it can be seen that it is possible to find the coordinates of a point in frame A given its coordinates in another frame B by first "mounting" the point in frame A and then transforming it to frame B. The following statement formalises this method.

Proposition 10.8: Changing reference frame of a point

The F_A^B matrix aligning frame A with frame B also transforms a point's coordinates in frame B to frame A.

This proposition provides the method for switching between different coordinate frames and is extremely important in robotics, where it is used for determining the position of an object relative to different parts of the robot. Additionally, Theorem 10.4 provides us with the following interpretation.

Proposition 10.9: Axis unit vectors of transformed coordinate frame

Given the transformation matrix

$$F_A^B = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

aligning coordinate frame A to B, the vectors

$$\mathbf{x}' = \begin{bmatrix} r_{11} \\ r_{21} \\ r_{31} \end{bmatrix}, \ \mathbf{y}' = \begin{bmatrix} r_{12} \\ r_{22} \\ r_{32} \end{bmatrix}, \ \mathbf{z}' = \begin{bmatrix} r_{13} \\ r_{23} \\ r_{33} \end{bmatrix}$$

are the x, y and z-axis unit vectors of frame B in frame A respectively.

The following theorem introduces an important tool for transforming coordinate frames and points.

Theorem 10.5: Performing transformations in an arbitrary frame

Suppose that a point (or coordinate frame) is given in a frame A. To transform the point/frame using the matrix M given in an arbitrary frame B that can be aligned from A using the matrix F_A^B, the transformation matrix

$$F_A^B M [F_A^B]^{-1}$$

can be applied [14].

Proof for Theorem 10.5

The first transformation using the matrix $[F_A^B]^{-1} = F_B^A$ transforms the given point (or frame) from frame A to frame B, as stated in Proposition 10.8. With the point/frame now in frame B, the transformation given by M is then performed, giving the transformed matrix in frame B. Finally, this is brought back to frame A using the matrix F_A^B, also per Proposition 10.8.

The point/frame is correctly transformed and the final result is given in frame A, and the above statement is thus correct.

Finally, we have the following statement for chaining frame transformations, which is an essential part of forward kinematics:

Proposition 10.10: Chaining coordinate frame transformations

Given transformation matrices F_A^B and F_B^C that align frame A to frame B and frame B to frame C respectively, the transformation matrix

$$F_A^C = F_A^B F_B^C$$

aligns frame A to frame C.

10.5 Kinematic chains and forward kinematics

Another way to define robotic actuation systems is to refer to them as *kinematic chains*, which are chains of rigid *links* connected by *joints* that can be actuated.

A joint can be either *revolute* (where it spins to rotate a link relative to another), or *prismatic* (where it slides to linearly move a link relative to another). A link is referred to as the *child* to a *parent link*, which is located towards the robot's mounting point. Figure 10.6 illustrates these two parts.

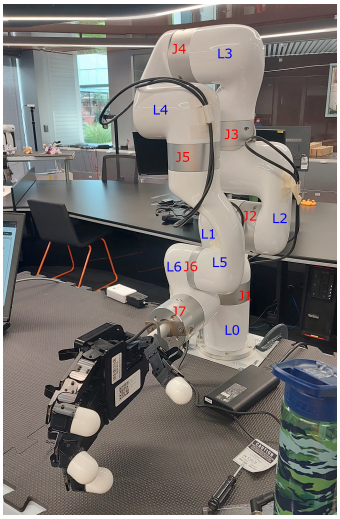


Figure 10.6: Revolute joints and links (labelled in red and blue text respectively) in a robotic manipulator. Note that $L0$ is a fixed link connecting the workbench with the first joint. *Photo by Vinh Nguyen.*

Mathematically, a kinematic chain can be considered as a chain of coordinate frames, with each frame defined relative to the preceding one. This representation is often used to compute the pose of joints, and especially of the end effector, given the actuators' positions, which is known as *forward kinematics*.

There also exists the inverse of this problem called *inverse kinematics*, where the actuators' positions are solved in order for the system to achieve a certain pose. However, the solution to this problem is significantly more complicated and is outside the scope of this report.

Elementary transform sequence

Using the concepts mentioned in the previous section, we can develop a simple approach to determine the forward kinematics of a robotic system through assigning coordinate frames to joints and working with transformations between these frames. This has been named the *elementary transform sequence* approach by Dr Peter Corke [7], and is described below.

Definition 10.8: Elementary transform sequence

Each joint i can be assigned a coordinate frame with an initial (*zero*) pose consisting of origin coordinates (x_i, y_i, z_i) and orientation $(\varphi_i, \theta_i, \psi_i)$ with respect to joint $i-1$. For the first joint, a fixed zero pose relative to the world frame can be chosen. The frames should be chosen such that the rotation (for revolute joints) or translation (for prismatic joints) axis is aligned to either the x, y or z axis. We thus have the following transformation matrix aligning joint i with the preceding joint, in accordance with Proposition 10.7:

$$Z_{i-1}^{i} = T(x_i, y_i, z_i)R(\varphi_i, \theta_i, \psi_i)$$

This matrix is then transformed with a rotation or translation transformation (depending on joint type) along the selected actuation axis; assuming the z axis is chosen, we have the following resulting alignment matrix from the frames of joint i to $i+1$ (which corresponds to link $i-1$):

$$F_i^{i+1} = \begin{cases} R_z(\theta_i) Z_i^{i+1} & \text{for revolute joint rotating by } \theta_i \\ T_z(d_i) Z_i^{i+1} & \text{for prismatic joint translating by } d_i \end{cases}$$

In the case of the world frame to first joint frame transformation, the alignment matrix F_0^1 is equal to Z_0^1, as the zero-pose frame of the first joint does not move relative to the world frame. Similarly, the tool frame can be treated as a joint.
The transformation matrix from the world frame to the zero frame of joint k is thus

$$F_0^k = \prod_{i=0}^{k-1} F_i^{i+1}$$

The matrix can be more generally expressed as

$$F_0^k = \prod_{i=1}^{m} \mathbf{E}_i(\eta_i)$$

where

$$\mathbf{E}_i \in \{T_x, T_y, T_z, R_x, R_y, R_z\}$$

and η_i is either a constant, or the joint position value

$$\begin{cases} \theta_i & \text{for a revolute joint, or} \\ d_i & \text{for a prismatic joint.} \end{cases}$$

Let us demonstrate this method with an example. The SCARA robot [16] (Figure 10.7) is a simple yet popular industrial robotic arm capable of moving in the 3D space using two revolute joints and a prismatic joint; the former makes it similar to human arms. A simplified diagram of the robot is shown in Figure 10.8.

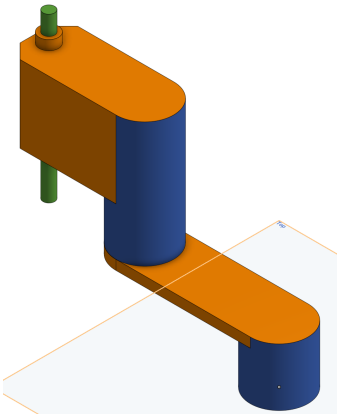


Figure 10.7: A simplified 3D model of the SCARA robot, illustrated using Onshape [13] according to [16]. Blue and green sections represent revolute and prismatic joints respectively, while orange sections represent rigid links.
Note that the end effector joint on the actual SCARA robot is a combined revolute-prismatic joint; the revolute component is omitted for simplicity.

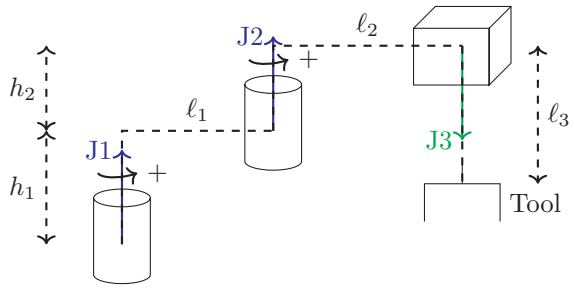
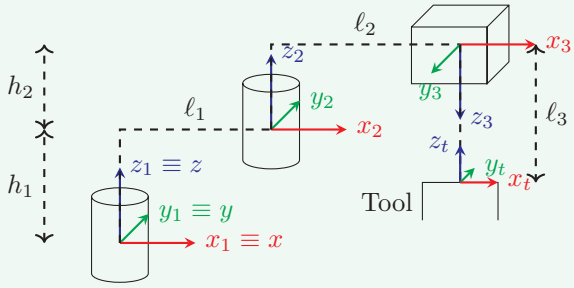


Figure 10.8: Kinematic chain diagram of the SCARA robot in Figure 10.7. Blue and green arrows illustrate the actuation axes of revolute and prismatic joints respectively.

Example 10.2: Demonstration of the elementary transform sequence

First, we will define the coordinate frames for each joint, as well as the world frame that we will work with in the example. For each joint's frame, the z axis is set to the actuation axis, and the x and y axes are then defined accordingly. This results in the world frame xyz and joint frames $x_1y_1z_1$, $x_2y_2z_2$ and $x_3y_3z_3$ corresponding to $J1$, $J2$ and $J3$ respectively, which are illustrated in the following drawing. We also define the tool frame $x_ty_tz_t$ for the end effector.



With these frames geometrically specified, we can write the zero-pose frame transformation matrices. As can be seen from the diagram, we have set up the world frame to be the same as that of $J1$, so the matrix transforming the world frame into $J1$'s zero frame is simply an identity matrix:

$$Z_0^1 = I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The $J1$-$J2$ frame transformation matrix is simply a translation along the x and z axes, since their frames are already aligned:

$$Z_1^2 = T(\ell_1, 0, h_1) = \begin{bmatrix} 1 & 0 & 0 & \ell_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The $J2$-$J3$ frame transformation also features a similar translation; however, a rotation is also needed to align the z axes. It can be seen that z_2 and z_3 are in opposite directions, but x_2 and x_3 are aligned, meaning that an 180° (or $\frac{\pi}{2}$ rad) rotation along the x axis is needed to align the frames:

$$Z_2^3 = T(\ell_2, 0, h_2)R\left(\frac{\pi}{2}, 0, 0\right) = \begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There is a shortcut we can take to obtain the rotation matrix. Using the interpretation provided in Proposition 10.9, we can specify the unit vectors of $J3$'s frame with respect to $J2$. These unit vectors are:

$$\begin{cases} \mathbf{x}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} & \text{as } x_3 \text{ points in the same direction as } x_2 \\ \mathbf{y}_3 = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} & \text{as } y_3 \text{ points in the opposite direction to } y_2 \\ \mathbf{z}_3 = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} & \text{as } z_3 \text{ points in the opposite direction to } z_2 \end{cases}$$

Similarly, we have the transformation matrix from the zero frame of $J3$ to the tool frame:

$$Z_3^t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From these zero-frame transformation matrices, we can construct the following transformations with actuator positions as parameters:

$$F_0^1 = Z_0^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F_1^2(\theta_1) = R_z(\theta_1)Z_1^2 = R_z(\theta_1)\begin{bmatrix} 1 & 0 & 0 & \ell_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F_2^3(\theta_2) = R_z(\theta_2)Z_2^3 = R_z(\theta_2)\begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F_3^t(d_3) = T_z(d_3)Z_3^t = T_z(d_3)\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, we have the following matrix for transforming the world frame to the tool frame:

$$
\begin{aligned}
F_0^t(\theta_1, \theta_2, d_3) &= F_0^1 F_1^2 F_2^3 F_3^t \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} R_z(\theta_1) \begin{bmatrix} 1 & 0 & 0 & \ell_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\quad R_z(\theta_2) \begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} T_z(d_3) \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

With $\theta_1 = \theta_2 = d_3 = 0$, the matrix becomes

$$
\begin{aligned}
Z_0^t = F_0^t(0,0,0) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \ell_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 & 0 & \ell_1 + \ell_2 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & h_1 + h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \ell_1 + \ell_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 + h_2 - \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
$$

which can be verified against the diagram at the beginning.

While this method is straightforward, it is clearly not the best, as each joint's properties must be described using 6 parameters: (x, y, z) for origin offset, and (φ, θ, ψ) for frame orientation. The next section describes a method for significantly reducing this count and optimising this process.

Denavit-Hartenberg forward kinematics

The *Denavit-Hartenberg convention*, proposed by Jacques Denavit and Richard Hartenberg in 1955 [5], standardises the definition and description of coordinate frames in a kinematic chain. The convention is described as follows [18].

Definition 10.9: Denavit-Hartenberg convention

In the *Denavit-Hartenberg convention*, each joint's coordinate frame's z axis is assigned to its actuation (i.e. rotation or translation) axis. Similarly to Definition 10.8, the end effector can be treated as a joint.

For the first joint, the x and y axes can be freely chosen, as long as they and the joint's z axis form a right-hand frame. For each subsequent joint, determine a line that intersects with and is perpendicular to both the joint's z axis and that of the preceding joint; this line is called the *common normal*. If the z axes are parallel or on the same line, there will be infinitely many common normals, and any one can be chosen.

The common normal's intersection with this joint's z axis is the origin of the joint's coordinate frame, and the joint's x axis is part of the common normal. The joint's y axis can then be constrained accordingly.

For each link i from joints i to $i+1$, the following parameters are specified:

- d_i (the *link offset*) is the offset along joint i's z axis from its origin to that of joint $i+1$. This parameter can be a negative value if joint $i+1$'s origin is on the negative half of joint i's z axis.
- θ_i (the *joint angle*) is the angle of rotation along joint i's z axis to align its x axis to that of joint $i+1$.
- r_i (the *link length*) is the common normal's length.
- α_i (the *link twist*) is the angle of rotation along joint $i+1$'s x axis to align joint i's z axis to that of joint $i+1$.

Proposition 10.11: Denavit-Hartenberg parameters

All the aforementioned parameters are constant as the joints move, except for:

- d_i for prismatic joints, or
- θ_i for revolute joints.

These parameters define their corresponding joint's actuation position.

In reality, an offset (denoted as d_{0i} and θ_{0i} for prismatic and revolute joints respectively) is specified for each joint to specify their zero positions; in which case, the following applies in Denavit-Hartenberg forward kinematics calculations:

- For prismatic joints, the link offset is $d_i + d_{0i}$.
- For revolute joints, the joint angle is $\theta_i + \theta_{0i}$.

These parameters are used to determine a kinematic chain's pose through the following theorem.

Theorem 10.6: Denavit-Hartenberg forward kinematics

Given the Denavit-Hartenberg parameters $(d_i, \theta_i, r_i, \alpha_i)$ of the link connect joints i with $i+1$, the transformation matrix aligning joint i's coordinate frame to that of joint $i+1$ is [4]:

$$F_i^{i+1} = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & r_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & r_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Proof of the Denavit-Hartenberg frame transformation matrix (10.6)

dh/fk Consider the frame alignment from joints 1 to 2, with joint 1's frame consisting of axes x_1, y_1 and z_1, and joint 2's frame consisting of axes x_2, y_2 and z_2. Let (d, θ, r, α) be the parameters of the link from joints 1 to 2.

According to Definition 10.9, joint 1's frame is rotated by θ about z_1 to align x_1 with x_2, and the frame is then rotated by α about x_2 to align z_1 with z_2. While the first rotation can be done in a simple transformation ($R_z(\theta)$) as its rotation axis is in joint 1's frame, the second rotation requires additional alignment transformations as stated in Theorem 10.5, and consists of three transformations:

1. Alignment of x_2 with x_1: $R_z(-\theta)$ (as x_1 is aligned with x_2 by a rotation about z_1 by θ)
2. Rotation about x_1 (which is now aligned with x_2): $R_x(\alpha)$
3. Re-alignment of x_1 with the original x_2: $R_z(\theta)$

The rotation part of the transformation is thus:

$$\begin{aligned} F_R &= R_z(\theta)R_x(\alpha)R_z(-\theta)R_z(\theta) \\ &= R_z(\theta)R_x(\alpha)R_z^{-1}(\theta)R_z(\theta) \\ &= R_z(\theta)R_x(\alpha) \quad \text{(the last two matrices cancelled out)} \\ &= R(\alpha, 0, \theta) \end{aligned}$$

From Definition 10.9, it can also be seen that the origin of joint 2 is formed by translating joint 1 frame's origin by d along z_1, and also by r along x_2. The former is a simple $T(0, 0, d)$ translation, while the latter can be thought of as a translation of joint 1's frame origin by r along x_1, followed by a rotation by θ along the z_1 axis to align it with the origin of joint 2's frame. Overall, the transformation of joint 1 frame's origin is:

$$O_2 = R_z(\theta)T(r, 0, 0)T(0, 0, d)\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}r\\0\\d\\1\end{bmatrix} = \begin{bmatrix}r\cos\theta\\ r\sin\theta\\ d\\ 1\end{bmatrix}$$

Joint 2 frame's origin therefore has the coordinates $(r\cos\theta, r\sin\theta, d)$, which is also the translation offset in the transformation. In other words, the translation part of the transformation is

$$F_T = T(r\cos\theta, r\sin\theta, d)$$

We thus have the following composite transformation matrix:

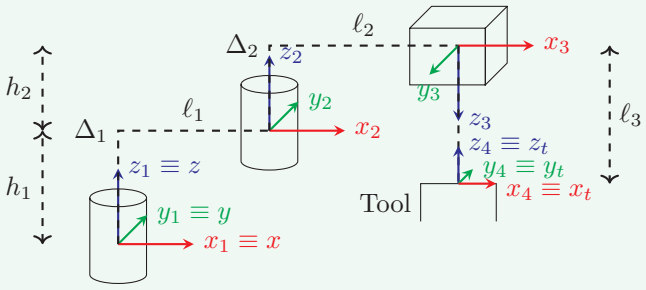
$$\begin{aligned} F &= F_T F_R \\ &= \begin{bmatrix}1 & 0 & 0 & r\cos\theta\\ 0 & 1 & 0 & r\sin\theta\\ 0 & 0 & 1 & d\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos\theta & -\sin\theta\cos\alpha & \sin\theta\sin\alpha & 0\\ \sin\theta & \cos\theta\cos\alpha & -\cos\theta\sin\alpha & 0\\ 0 & \sin\alpha & \cos\alpha & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \\ &= \begin{bmatrix}\cos\theta & -\sin\theta\cos\alpha & \sin\theta\sin\alpha & r\cos\theta\\ \sin\theta & \cos\theta\cos\alpha & -\cos\theta\sin\alpha & r\sin\theta\\ 0 & \sin\alpha & \cos\alpha & d\\ 0 & 0 & 0 & 1\end{bmatrix} \end{aligned}$$

The matrix matches that of Theorem 10.6, and the theorem is therefore proven.

The following example demonstrates this method on the aforementioned SCARA robot.

Example 10.3: Example of Denavit-Hartenberg forward kinematics

The world frame xyz, joint frames $x_1y_1z_1$, $x_2y_2z_2$ and $x_3y_3z_3$ corresponding to $J1$, $J2$ and $J3$ determined using the Denavit-Hartenberg convention is shown in the following diagram. The tool frame $x_4y_4z_4$ determined using the Denavit-Hartenberg convention, as well as the desired tool frame $x_ty_tz_t$ are also shown. Note that Δ_1 and Δ_2 are common normals of z_1–z_2 and z_2–z_3 respectively.



Note that the common normal of z_3–z_4 is not illustrated; since we choose z_4 to be on the same line as z_3, any line perpendicular to them can be considered to be on the common normal, and the common normal length is zero.

Since we have chosen z_4 to be the same as z_t, we can also choose our x_4 to be the same as x_t, and thus make $x_4y_4z_4$ naturally aligned to $x_ty_tz_t$.

Using the robot's dimension parameters given in the diagram above, we have the following table of Denavit-Hartenberg parameters for the robot's links:

i	**From**	**To**	**Joint type**	d_i/d_{0i}	θ_i/θ_{0i} **(rad)**	r_i	α_i **(rad)**
1	$J1$	$J2$	Revolute	h_1	0	ℓ_1	0
2	$J2$	$J3$	Revolute	h_2	0	ℓ_2	π
3	$J3$	$J4$ (tool)	Prismatic	ℓ_3	0	0	π

Note that since we are working with the robot's zero position, for revolute joints, the θ_i column value corresponds to θ_{0i}, and for prismatic joints, the d_i column value corresponds to d_{0i}.

We thus have the following transformation matrices for each link:

$$F_1^2(\theta_1) = \begin{bmatrix} \cos\theta_1 & -\sin\theta_1\cos 0 & \sin\theta_1\sin 0 & \ell_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1\cos 0 & -\cos\theta_1\sin 0 & \ell_1\sin\theta_1 \\ 0 & \sin 0 & \cos 0 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & \ell_1\cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & \ell_1\sin\theta_1 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F_2^3(\theta_2) = \begin{bmatrix} \cos\theta_2 & -\sin\theta_2\cos\pi & \sin\theta_2\sin\pi & \ell_2\cos\theta_2 \\ \sin\theta_2 & \cos\theta_2\cos\pi & -\cos\theta_2\sin\pi & \ell_2\sin\theta_2 \\ 0 & \sin\pi & \cos\pi & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta_2 & \sin\theta_2 & 0 & \ell_2\cos\theta_2 \\ \sin\theta_2 & -\cos\theta_2 & 0 & \ell_2\sin\theta_2 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$F_3^4(d_3) = \begin{bmatrix} \cos 0 & -\sin 0 \cos \pi & \sin 0 \sin \pi & 0 \cos 0 \\ \sin 0 & \cos 0 \cos \pi & -\cos 0 \sin \pi & 0 \sin 0 \\ 0 & \sin \pi & \cos \pi & \ell_3 + d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 + d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Since we have set up $x_1y_1z_1$ to align with xyz and $x_4y_4z_4$ to align with $x_ty_tz_t$, the transformation matrices from world frame to $J1$ and from $J4$ to tool frame are

$$F_0^1 = F_4^t = I_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From these matrices, we can finally assemble the world to tool frame transformation matrix in the same manner as in Example 10.2:

$$\begin{aligned} F_0^t(\theta_1, \theta_2, d_3) &= F_0^1 F_1^2 F_2^3 F_3^4 F_4^t \\ &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & \ell_1 \cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & \ell_1 \sin\theta_1 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\quad \begin{bmatrix} \cos\theta_2 & \sin\theta_2 & 0 & \ell_2 \cos\theta_2 \\ \sin\theta_2 & -\cos\theta_2 & 0 & \ell_2 \sin\theta_2 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 + d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & \ell_1 \cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & \ell_1 \sin\theta_1 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & \sin\theta_2 & 0 & \ell_2 \cos\theta_2 \\ \sin\theta_2 & -\cos\theta_2 & 0 & \ell_2 \sin\theta_2 \\ 0 & 0 & -1 & h_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &\quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & \ell_3 + d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 & \ell_1 \cos\theta_1 \\ \sin\theta_1 & \cos\theta_1 & 0 & \ell_1 \sin\theta_1 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_2 & -\sin\theta_2 & 0 & \ell_2 \cos\theta_2 \\ \sin\theta_2 & \cos\theta_2 & 0 & \ell_2 \sin\theta_2 \\ 0 & 0 & 1 & h_2 - \ell_3 - d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

With $\theta_1 = \theta_2 = d_3 = 0$, the matrix becomes

$$Z_0^t = F_0^t(0,0,0) = \begin{bmatrix} \cos 0 & -\sin 0 & 0 & \ell_1 \cos 0 \\ \sin 0 & \cos 0 & 0 & \ell_1 \sin 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos 0 & -\sin 0 & 0 & \ell_2 \cos 0 \\ \sin 0 & \cos 0 & 0 & \ell_2 \sin 0 \\ 0 & 0 & 1 & h_2 - \ell_3 - 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \ell_1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & \ell_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_2 - \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \ell_1 + \ell_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_1 + h_2 - \ell_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which is similar to that of Example 10.2.

Figure 10.9 shows the world and joint coordinate frames retrieved in Examples 10.2 and 10.3.

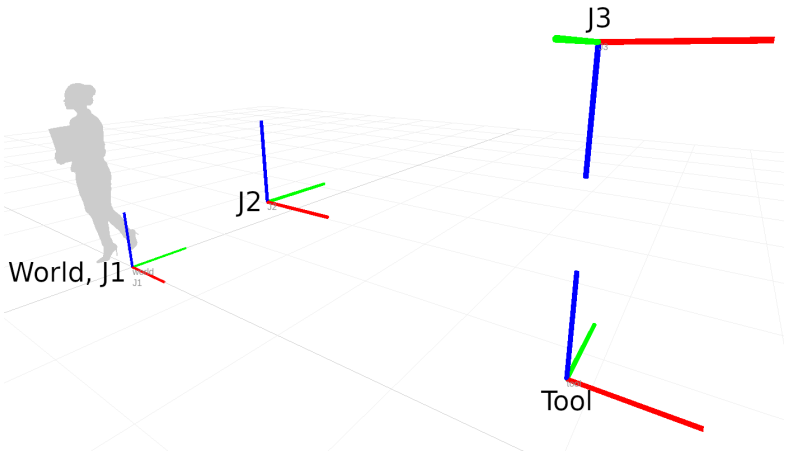


Figure 10.9: World and joint coordinate frames of the SCARA robot in Figure 10.7 (assuming $h_1 = h_2 = 1$, $\ell_1 = \ell_2 = 2$, and $\ell_3 = 1.5$). The x, y and z axes of each frame are shown in red, green and blue respectively. *This visualisation is created using Daniel Dugas' 3D transform visualisation tool, https://dugas.ch/transform_viewer/index.html used with permission [6].*

As can be seen in Example 10.3, the Denavit-Hartenberg forward kinematics approach only requires 4 parameters per joint. This makes this approach suitable for computing the forward kinematics for complex robotic systems with more joints and links, and the approach is thus more commonly used in robotics.

However, the Denavit-Hartenberg convention introduces constraints to coordinate frame placement that often cause them to not match the joint's location. These constraints make the approach less intuitive and harder to understand. A possible approach is to use elementary transform sequences when determining the robot's kinematic parameters, then converting them to Denavit-Hartenberg parameters for computation; interested readers may refer to [3] for details on the conversion process.

10.6 Summary

This report provided an insight into the application of linear algebra in robotics, particularly in forward kinematics, where a robotic system's position relative to a reference frame is solved given the system's intrinsic configuration and pose data. The report covered point and coordinate frame transformations in three-dimensional space as the foundation of this application. From this concept, approaches to solving the forward kinematics problem were introduced, including a simple but relatively inefficient approach directly derived from frame transformations, as well as the more commonly used Denavit-Hartenberg approach.

Context

This report was adapted from the author's High Distinction report regarding the application of linear algebra in robotics for the SIT292 Linear Algebra for Data Analysis unit.

About the Author

Vinh Nguyen is a second-year student pursuing the Bachelor of Computer Science course at Deakin University, majoring in Robotics. He is passionate about robotics and automation, especially their underlying algorithms, and has been volunteering at the university's Robotics and Internet of Things (RIOT) Lab to create robotic demonstrations during Trimester 2 of 2024.

Acknowledgements

The author would like to thank **Paul Phan** from the RIOT Lab at Deakin University for providing suggestions on this report's topic.

References

Unless otherwise stated, all figures were created using the TikZ package [17] in LaTeX.

[1] National Aeronautics and Space Administration. The axes of flight. *NASA Learning Resources*, 2023. Available online: https://www.nasa.gov/stem-content/the-axes-of-flight/.

[2] Michael Cheng, Yuan Chen, and Michael Humphreys. Rotation matrix and tilt about X/Y/Z in OpticStudio. Technical report, Zemax, 2021.

[3] Peter Corke. A simple and systematic approach to assigning Denavit-Hartenberg parameters. *IEEE Transactions on Robotics*, 23:590–594, 6 2007.

[4] J. Denavit and R. S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, 22(2):215–221, 1955.

[5] Jacques Denavit and Richard S. Hartenberg. A kinematic notation for lower-pair mechanisms based on matrices. *Journal of Applied Mechanics*, 22(2):215–221, 1955.

[6] Daniel Dugas. 3D transform visualizer tool. https://dugas.ch/transform_viewer/multi.html.

[7] Jesse Haviland and Peter Corke. Manipulator differential kinematics: Part I: Kinematics, velocity, and applications. *IEEE Robotics & Automation Magazine*, 31:149–158, 12 2024.

[8] K. Lehn, M. Gotzes, and F. Klawonn. Geometry processing. In *Introduction to Computer Graphics. Undergraduate Topics in Computer Science*. Springer, Cham., 2023.

[9] W. Meyer. Transformation geometry II: Isometries and matrices. In *Geometry and Its Applications (Second Edition)*, pages 257–310. Academic Press, Burlington, 2006.

[10] Madhav Mohan. Computer graphics - Reflection transformation in 3D. *GeeksforGeeks*, 2022.

[11] W. K. Nicholson. *Linear Algebra with Applications*. Lyryx, Revised A edition, 2021.

[12] Nikhil Niphadkar. Coordinate system frames in industrial robots. Technical report, Association for Advancing Automation, 2017.

[13] Onshape. Onshape product development platform, 2025. https://www.onshape.com.

[14] G. Scott Owen. 3D rotation (excerpt from HyperGraph). University of Helsinki, 1996.

[15] Frederick Roberts and Donna Roberts. Rigid transformations. *MathBitsNotebook.com*, 2025.

[16] Yuta Sato and Shingo Hoshino. SCARA robot, U.S. Patent 9,168,660, 10 2015.

[17] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

[18] Ethan Tira-Thompson. Denavit-Hartenberg reference frame layout. YouTube, 2009. https://www.youtube.com/watch?v=rA9tm0gTln8.

[19] Eric W. Weisstein. Rigid motion. *MathWorld–A Wolfram Web Resource*, 2025. https://mathworld.wolfram.com/RigidMotion.html.

11

Hash functions in digital security

Casey Wilfling

Abstract

This article explores the essential role of cryptographic hash functions in digital security, highlighting their use in ensuring data integrity, verifying authenticity, and securing digital signatures. It delves into their relationship with the CIA Triad—confidentiality, integrity, and availability—by explaining how hash functions contribute to these security principles. The discussion focuses on two well-known algorithms, MD5 and SHA-1, detailing their original applications, subsequent vulnerabilities to collision attacks, and their eventual deprecation for critical security tasks. Despite their limitations, simplified versions like Mini-MD5 and Mini-SHA-1 offer a valuable learning opportunity, providing insights into core cryptographic processes such as message padding, buffer initialization, and bitwise operations. This article aims to demystify these concepts by examining the mathematical foundations, step-by-step computations making the mechanics of cryptographic hash functions accessible to those new to the field.

11.1 Introduction

Cryptographic hash functions play a crucial role in modern digital security. These mathematical tools ensure data integrity, verify message authenticity, and secure digital signatures. A cryptographic hash function takes an input (or message) and produces a fixed-size string of characters which typically looks like a random sequence of letters and numbers. This output, known as the **hash value** or **message digest**, is unique to the input. Even a minor change in the input results in a drastically different hash, a property called the avalanche effect.

Hash functions are indispensable for:

- **Data Integrity:** Hash functions help confirm that the data sent over a network has not been altered. For instance, during file downloads, the hash of the original file is compared to the hash of the downloaded file to ensure that they are identical.
- **Digital Signatures:** Hash functions ensure the authenticity and integrity of messages. A message is hashed, and the resulting digest is encrypted with a private key to create the signature.
- **Password Storage:** Instead of directly saving passwords, the systems store the hashed version. When users log in, the system hashes the input and compares it with the stored hash.

Hash functions also support the CIA Triad, which is a foundational principle in information security.

- **Confidentiality:** Although hash functions themselves do not encrypt data, they often work in tandem with encryption algorithms to secure sensitive information. For example, hashed passwords ensure that, even if the storage system is compromised, the actual passwords remain confidential.
- **Integrity:** Hash functions are primarily used to guarantee that data has not been tampered with during storage or transmission. The unique hash value of the original data ensures its integrity, as even a tiny modification will produce a completely different hash value.
- **Availability:** By ensuring data integrity and authenticity, hash functions help maintain the reliability of systems, making data accessible and trustworthy for users.

This article explains how cryptographic hash functions help maintain the integrity and authenticity of data in modern computing.

Two well-known hash functions, MD5 and SHA-1, were developed to produce fixed-length outputs from variable-length inputs. They have been widely used for tasks such as verifying file integrity, hashing passwords, and enabling secure communications. However, advances in cryptanalysis revealed vulnerabilities in these algorithms, particularly their susceptibility to collision attacks, rendering them unsuitable for critical security applications [5].

A **collision attack** occurs when two different inputs generate the same hash value, undermining the reliability of the hash function. This vulnerability allows attackers to alter data while maintaining the same hash value, potentially leading to significant security breaches. For example, an attacker could craft a malicious document with the same hash value as a legitimate one, bypassing integrity checks [7]. Consequently, both MD5 and SHA-1 are now considered insecure for modern cryptographic use [5].

Despite these vulnerabilities, simplified versions of these algorithms—Mini-MD5 and Mini-SHA-1—offer valuable educational insights. These streamlined versions retain key cryptographic processes, such as message padding, buffer initialization, and bitwise operations, but with reduced complexity. By focusing on simplified operations like XOR and using smaller buffer and block sizes, these mini-algorithms make it easier to understand core concepts like message digest computation and circular shifts [7].

The goal of this article is to explore the mathematical foundations and practical implementations of the Mini-MD5 and Mini-SHA-1 systems. These simplified models provide a powerful framework for understanding the mechanics of hash functions, making them ideal for introductory cryptography studies. We will delve into their theoretical principles, step-by-step processes, and implementations in SageMath, highlighting their connection to the full MD5 and SHA-1 algorithms [6, 1]. Those interested in exploring cryptography in further detail are directed to [1, 2, 3, 4, 8].

11.2 Preliminaries

Notations and Terminology

Hash Function: A mathematical function that transforms input data of arbitrary size into a fixed-size output (hash value or digest).

XOR ($\oplus$): A bitwise operation where the result is 1 if the bits differ and 0 if they are the same.

Modular Arithmetic: Arithmetic system that confines numbers to a fixed range by wrapping values around upon reaching a modulus.

Circular Shift: A bitwise operation where bits are rotated left or right, with overflow bits reintroduced at the opposite end.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Table 11.1: Truth table for XOR operation

We set out mathematics using either numbered equations, as in

$$\cos\theta = \sum_{i=1}^{n} \frac{z_1}{|A||B|} (xK) \tag{11.1}$$

which can then be referred to as Eqn. (11.1) or as unnumbered equations either inline with text $(x-y)\frac{1}{t}$ or separately,

$$AB = \begin{bmatrix} a & b \\ 1 & c \end{bmatrix}.$$

11.3 Learning Objectives

In this section, I will outline the Learning Objectives for understanding and implementing simplified cryptographic hash functions. These objectives focus on the core principles of hash functions and their application through the Mini-MD5 and Mini-SHA-1 systems.

1. Explain the Purpose and Functionality of Cryptographic Hash Functions.
Understand what cryptographic hash functions are and their role in ensuring data integrity. Learn how they are applied in real-world security protocols and data verification systems.

The purpose and functionality of cryptographic hash functions were covered in the Introduction section of this report.

2. Implement a Simplified Version of the MD5 Hash Function.
We implement the Mini-MD5 algorithm with reduced buffer sizes and rounds, demonstrating how bitwise operations, message padding, and modular arithmetic work together to generate a hash output.

The MD5 algorithm is a widely known cryptographic hash function that generates a 128-bit hash value. However, we will focus on a simplified version, Mini-MD5, for educational purposes. In this implementation:

- **Buffer Sizes:** We reduce the buffer sizes from 32-bit to 16-bit.
- **Rounds:** Instead of the original MD5's 64 operations across 4 rounds, Mini-MD5 consists of 2 rounds, each with 4 operations.
- **Bitwise Operations:** XOR operations are central to this system, used to combine buffer values and parts of the message block.
- **Message Padding:** To ensure that the input message fits the required block size (256 bits), we apply padding to the message if necessary.
- **Modular Arithmetic:** Hash functions involve performing operations modulo a certain value (such as 2^{16} for 16-bit buffers). This prevents overflow and ensures the result fits within the specified number of bits.

This objective emphasizes the key components of Mini-MD5, allowing learners to follow each step in generating a hash output. Table 11.2 compares MD5 and Mini-Md5 (see [6] for details relating to the full algorithm).

Feature	Full MD5 Algorithm	Mini-MD5 Algorithm
Message Block Size	512 bits	256 bits
Buffer Size	4 x 32-bit Buffers (A, B, C, D)	4 x 16-bit Buffers (A, B, C, D)
Hash Output Size	128-bit (32 hex characters)	64-bit (16 hex characters)
Number of Rounds	4 Rounds, each with 16 operations (64 operations total)	2 Rounds, each with 4 operations (8 operations total)
Mathematical Functions	Logical operations like AND, OR, XOR, and modular addition	Simplified XOR
Padding Scheme	Message padded to a multiple of 512 bits	Message padded to a multiple of 256 bits
Circular Shifts	32-bit circular shifts applied during processing	16-bit circular shifts

Table 11.2: Comparison of Full MD5 vs. Mini-MD5 System

3. Implement a Simplified Version of the SHA-1 Hash Function.
We implement the Mini-SHA-1 algorithm with fewer steps and simplified logic, demonstrating how the round process works, involving bitwise operations, modular arithmetic, and function chaining, to produce a hash output.

SHA-1 was once widely used to ensure the integrity of messages and data. In this learning objective, we will explore the Mini-SHA-1 system, which simplifies the full SHA-1 algorithm while retaining its core structure. The implementation involves:

- **Reduced Message Size:** Mini-SHA-1 works with a reduced message length of 32 bits.
- **Simplified Buffer Size:** The original SHA-1 uses five 32-bit buffers, whereas Mini-SHA-1 uses five 16-bit buffers (A, B, C, D, E).
- **Round Process:** Mini-SHA-1 reduces the 80 rounds of SHA-1 into 8 rounds, where bitwise XOR operations and function chaining are performed to transform the message block. The process involves updating buffer values in each round to produce the final hash output.
- **Function Chaining:** The values in the five buffers are updated in each round based on logical operations applied to one another, creating dependencies across rounds.
- **Modular Arithmetic and Circular Shifts:** Each round performs left circular shifts and modular arithmetic to keep the buffer values within the 16-bit size limit.

By implementing Mini-SHA-1, learners will gain a deeper understanding of how bitwise logic and function chaining contribute to secure hash generation. Table 11.3 compares SHA-1 and Mini SHA-1 (see [1] for details regarding the full algorithm).

Feature	Full SHA-1 Algorithm	Mini-SHA-1 System
Message Block Size	512 bits	32 bits
Buffer Size	5 buffers (A, B, C, D, E), each 32 bits	5 buffers (A, B, C, D, E), each 16 bits
Hash Output Size	160-bit (40 hex characters)	64-bit (16 hex characters)
Number of Rounds	4 rounds, each with 20 operations (80 operations total)	8 rounds, each with 4 operations (32 operations total)
Mathematical Functions	Logical operations like XOR, AND, OR, and modular addition	Simplified XOR
Padding Scheme	Message padded to be a multiple of 512 bits	Message padded to be a multiple of 32 bits
Circular Shifts	32-bit circular shifts applied during processing	16-bit circular shifts

Table 11.3: Comparison of Full SHA-1 vs. Mini-SHA-1 System

11.4 Theoretical and Practical Explanations

This section explores the key concepts and practical aspects of two simplified cryptographic systems: Mini-MD5 and Mini-SHA-1. These streamlined versions of the widely used MD5 and

SHA-1 hash functions are designed to make the principles of cryptographic hashing more accessible for educational purposes by showcasing the underlying mathematical operations and logic.

The discussion begins with an explanation of how the Mini-MD5 and Mini-SHA-1 systems operate, highlighting essential processes such as message padding, XOR functions, modular arithmetic, and circular shifts. These components work together to produce valid hash outputs while adhering to foundational cryptographic principles, even in their simplified forms.

Next, each system is explored in depth through step-by-step mathematical calculations. The detailed breakdown includes buffer updates, transformations, and the application of cryptographic functions, culminating in the final hash output for a given input message. Clear examples ensure that the explanation is easy to follow and grounded in practical application.

Since hash values in cryptographic systems are typically represented as hexadecimal values, all arithmetic in this section is conducted using hexadecimal notation. To maintain consistency and clarity, uppercase letters (A-F) are used throughout to represent hexadecimal digits.

The objectives of this section are twofold:
1. To demonstrate the theoretical soundness of the Mini-MD5 and Mini-SHA-1 algorithms by validating their ability to generate accurate hash outputs.

2. To provide a practical understanding of these algorithms through detailed, step-by-step calculations, enabling readers to grasp how hash values are computed.

By integrating theoretical explanations with practical demonstrations, this section offers a comprehensive view of how cryptographic hash functions work in a simplified and approachable educational context.

General Steps of the Mini-MD5 Algorithm

Before we delve into a detailed explanation of Mini-MD5, it is essential to understand the general steps of the algorithm and why each is necessary. The Mini-MD5 algorithm is a simplified version of the MD5 cryptographic hash function, designed to transform an input message into a fixed-size 64-bit hash. This section provides a high-level overview of the Mini-MD5 algorithm, laying the groundwork for the formal mathematical proofs and detailed calculations that follow.

Step 1: Message Padding The first step of the Mini-MD5 algorithm ensures that the input message, denoted as M, fits the required block size of 256 bits (32 bytes). Since messages usually come in arbitrary lengths, padding is applied to make the message a fixed size. The padding process involves:

1. **Appending a 1-byte value:** A 1-byte value (represented as 10000000 in binary) is appended to the message to signify the end of the message. In hexadecimal, this value is represented as **0x80**. For example, the message "abc", in hexadecimal, would be represented as:

$$a = 61, \quad b = 62, \quad c = 63, \quad \text{appended byte} = 80$$

$$\text{Final result} = 61626380$$

2. **Appending 0-bits:** Enough 0-bits (00000000 in binary, 00 in hex) are appended to the message to make the total length 224 bits.

3. **Appending the original message length:** The last 32 bits of the 256-bit message contain the length of the original message, represented in hexadecimal. For example, if the original message is 32 bits long, its hexadecimal equivalent is **20**, which is appended to the end of the padded message.

The padded message M is then divided into sixteen 16-bit words, represented as $W[0], W[1], \ldots, W[15]$. In the Mini-MD5 system, only $W[0]$ to $W[7]$ are used in calculations for Round 1 and Round 2, while the full MD5 system uses all 16 words.

Why is message padding needed?

- *Fixed block size:* Cryptographic hash functions operate on fixed-size blocks. Padding ensures that the message is the correct size for block-wise processing.
- *Message integrity:* The appended 1-bit marks the end of the actual message, and the final 32 bits preserve the original message length for accurate hashing.

Step 2: Initialization of Buffers The Mini-MD5 algorithm begins with the initialization of four 16-bit buffers, each set to specific starting values:

These values are used as follows:

$$A = \texttt{1234}, \quad B = \texttt{5678}, \quad C = \texttt{9ABC}, \quad D = \texttt{DEF0}$$

Why are these buffers needed?

- **Starting state:** These buffers represent the state of the hash computation at any given point. The initial values are fixed to provide a consistent and uniform starting point for all messages, ensuring that the hashing process begins from a predictable and secure baseline.

Step 3 and 4: Processing the Message in Rounds (Step 3 = Round 1, Step 4 = Round 2) After padding the message in Step 1, the message is divided into 16-bit blocks. The Mini-MD5 algorithm processes the message through two rounds of operations. Each round consists of sixteen operations (four operations on each 16-bit word) where the buffers are updated at the end of the operations based on specific XOR functions and message blocks.

The four operations that we perform on each 16-bit word are [6, 7]:

1. **XOR Function**
2. **Modular Arithmetic**
3. **Circular Shifts**
4. **Buffer Rotation**

These updated buffer values will then be carried over to the next 16-bit words calculations, where the four operations will be repeated.

1. **XOR Function**

Note: Throughout the explanation and calculations of the Mini-MD5 system, the XOR operation will be denoted using its traditional symbol, $\oplus$.

1. XOR Function

The XOR function is computed on the buffers B, C, D for each word in the message block.

- **Round 1:** The result of the function

$$F(B, C, D) = B \oplus C \oplus D$$

  is used in further calculations in conjunction with words $W[0]$ to $W[3]$ during the following *Modular Arithmetic* step.

- **Round 2:** The function

$$G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$$

  is applied to the words $W[4]$ to $W[7]$ in the same manner as Round 1.

We begin calculations in sequence, starting with $W[0]$ and proceeding through the remaining words in the block. For each word $W[i]$, the following steps are performed:

1. Apply the appropriate XOR function (F or G) based on the round.
2. Update the buffer values A, B, C, D after the XOR computation for the current word $W[i]$.

2. Modular Arithmetic

Updating buffer A, denoted as A_U, involves performing arithmetic addition on the current value of A, the result of the XOR function, and the current word $W[i]$. This is followed by a modular arithmetic operation using mod 2^{16} to ensure the result fits within 16 bits. The formula to calculate A_U is:

$$A_U = (A + F(B, C, D) + W[i]) \mod 2^{16}, \quad \text{for Round 1}$$

$$A_U = (A + G(B, C, D) + W[i]) \mod 2^{16}, \quad \text{for Round 2}$$

Where:

- A is the current value of buffer A.
- $F(B, C, D) = B \oplus C \oplus D$, the XOR function used in Round 1.
- $G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$, the XOR function used in Round 2.
- $W[i]$ represents the current 16-bit word being processed.

To compute A_U:

1. Convert A, $F(B, C, D)$ or $G(B, C, D)$, and $W[i]$ from their hexadecimal values to their base-10 equivalents.

2. Perform arithmetic addition as $A+F(B,C,D)+W[i]$ (for Round 1) or $A+G(B,C,D)+W[i]$ (for Round 2).
3. Apply $\mod 2^{16}$ to ensure the result remains within 16 bits.
4. Convert the modulo value back into hexadecimal, represented as A_U, to proceed with subsequent operations.

3. Circular Shift Once we obtain the value of A_U, we perform a circular shift on A_U to compute A_S. A circular shift is a bitwise operation that shifts the bits of a binary number to the left or right, with the bits that are shifted out on one end reinserted, or wrapped around, on the other end. This operation can be performed using the following formula in SageMath:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Where:

- $\ll$ denotes a left shift of the bits.
- $\gg$ denotes a right shift of the bits.
- shift_amount is either 3 (for Round 1) or 5 (for Round 2).
- $\wedge 0xFFFF$ ensures that the result remains constrained to 16 bits.

The steps involved in this formula are as follows:

1. $\mathbf{A_U} \ll$ shift_amount
   - This shifts the bits of A_U to the left by shift_amount positions.
   - Any bits that move beyond the 16th bit are initially lost (since we are working with 16-bit numbers).
2. $(A_U \ll \text{shift\_amount}) \wedge 0xFFFF$
   - After the left shift, a bitwise AND operation with $0xFFFF$ (1111 1111 1111 1111 in binary) ensures the result remains within 16 bits.
   - This operation discards any bits beyond the lower 16 bits.
3. $\mathbf{A_U} \gg (\mathbf{16} - \text{shift\_amount})$
   - This shifts the bits of A_U to the right by $16 - \text{shift\_amount}$ positions.
   - The operation captures the bits shifted out on the left during the left shift and brings them back to the rightmost positions.
4. $\vee$ (Bitwise OR)
   - The result of the left shift and the result of the right shift are combined using a bitwise OR ($\vee$).
   - This ensures that the bits shifted out on the left are reinserted on the right, completing the circular shift.

Why is this needed?

- **Data Diffusion:** Circular shifts play a critical role in ensuring that all bits in the buffer contribute to the final hash value. By wrapping shifted bits around, the operation avoids the loss of information while spreading the influence of the input bits throughout the buffer.
- **Maintaining 16-Bit Consistency:** The use of bitwise AND ($\wedge 0xFFFF$) ensures that the resulting value remains confined to 16 bits, maintaining compatibility with the hash algorithm's constraints.
- **Enhancing Non-Linearity:** The combination of left and right shifts, along with the bitwise OR ($\vee$), enhances the non-linear transformations required for cryptographic strength, making it harder for attackers to predict or reverse-engineer the hash output.

4. Buffer Rotation To rearrange A, B, C, and D for the next word block operations, we perform buffer value rotation following this pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

These updated buffer values are then carried over to the next group of operations that are to be performed in conjunction with the next word in the sequence. For example, once we compute our updated buffer values using $W[0]$ in our operations, we carry over these values of A, B, C, D to be used in conjunction with $W[1]$ across Steps 1-4.

Why is this needed?

- **Mixing the data:** The XOR functions F and G ensure non-linear mixing of the buffer values, providing the cryptographic strength necessary for hashing.
- **Circular shifts:** These shifts ensure that the bits are mixed further while preserving the original bit-length of the buffers.
- **Modular arithmetic:** Keeps the buffer values within the 16-bit limit, ensuring consistency across the hashing process.

Step 5: Producing the Final Hash Output After completing both rounds, the final values of the buffers A, B, C, D are concatenated to produce a 64-bit hash. The concatenation of the buffers is denoted using the symbol $||$:

$$\text{Hash} = A||B||C||D$$

Why is this needed?

- **Fixed-size output:** The final 64-bit hash is a compact, fixed-size representation of the original message, which is the hallmark of cryptographic hash functions. This hash can be used to verify the integrity of the message.

11.5 Explanation of Mini-MD5

Statement of the Problem We aim to demonstrate that the Mini-MD5 system transforms an input message, M, into a valid 64-bit hash using two rounds of operations. The first round utilizes the function:

$$F(B, C, D) = B \oplus C \oplus D$$

While the second round uses a different function:

$$G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$$

Both functions are XOR-based, ensuring non-linearity and effective bit mixing. This explanation will address the following steps:

1. **Message Padding:** Ensuring that any input message fits into a 256-bit block.
2. **XOR Functions F and G:** Demonstrating that the XOR functions mix the buffer values effectively.
3. **Modular Arithmetic:** Ensuring that operations remain within the 16-bit range.
4. **Circular Shifts:** Proving that 3-bit (for Round 1) and 5-bit (for Round 2) circular shifts mix the bits without loss.
5. **Buffer Rotation:** Ensuring that the buffer values are properly rotated after each operation to distribute transformations across the buffers.
6. **Final Hash Output:** Demonstrating that the final buffer values are concatenated to form the correct 64-bit hash.

Message Padding

1. Assumptions The original message has a length L (in bits). We must ensure it fits into a padded 256-bit block message M. For example, if our original message is "MATH," this would be calculated as $L = 4 \times 8$, since each letter is 1-byte in length. This would be represented in binary as:

M	A	T	H
01001101	01000001	01010100	01001000

Since we are working with hexadecimal values, the binary value is converted to:

M	A	T	H
4D	41	54	48

To complete the 256-bit message M, we implement the Message Padding process to fill the remaining 224-bits.

2. Message Padding Process

1. Append a 1-byte (10000000 in binary) to signify the end of the message. In hexadecimal, 10000000 is converted to 80.
2. Add enough 0 bits to bring the message to 224 bits.
3. The last 32 bits of the 256-bit message is the hexadecimal value representing the original message length. For example, as the original message length is 32 bits, in hexadecimal this converts to 20.

The message is then divided into sixteen 16-bit blocks labeled $W[0]$ to $W[15]$. As we are using a simplified version of the MD5 algorithm, our Mini-MD5 system will operate only on words $W[0]$ to $W[7]$.

3. Deductive Argument Padding ensures that messages of arbitrary length can fit into a fixed-size block of 256 bits, as required by the algorithm [6, 7]. Without padding, the algorithm would be unable to process messages of variable length in a consistent manner. The addition of the 1-bit at the end of the original message ensures that the message length is preserved, and the final 32-bit block representing the original length confirms this.

4. Conclusion The message is padded to a 256-bit block, which is essential for ensuring that the algorithm can process the message in a fixed-length format. This step guarantees that the entire message is correctly accounted for in the hashing process.

XOR Functions F and G

To restate, we will be denoting the XOR operation using the symbol $\oplus$.

1. Function $F(B, C, D)$ for Round 1 The function $F(B, C, D) = B \oplus C \oplus D$ combines three 16-bit buffers using the XOR operation [7]. This function is both commutative and associative, ensuring that the order of operations does not affect the result.

Deductive Argument: For each operation in Round 1, $F(B, C, D)$ is applied to the current buffer values, mixing them in a non-linear way and enhancing the security of the system.

2. Function $G(B, C, D)$ for Round 2 In Round 2, we use a different XOR function:

$$G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$$

This function adds complexity by using different buffer pairings for the XOR operations.

3. Deductive Argument: The function $G(B, C, D)$ further mixes the buffers by combining the results of multiple XOR operations. This ensures that even after Round 1, the bits are mixed again, improving security.

4. Conclusion The use of two different XOR functions $F(B, C, D)$ and $G(B, C, D)$ ensures that the buffers are non-linearly combined in each round. This provides strong cryptographic properties by making it difficult to reverse the operations.

Modular Arithmetic

1. Assumptions After computing $F(B, C, D)$ or $G(B, C, D)$, buffer A is updated to value A_U using the formula:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Where $W[i]$ is the current 16-bit word we are operating on. Please refer to Section 1.1, General Steps of the Mini-MD5 Algorithm, if further clarification is needed regarding the above formula.

2. Modular Arithmetic Modular addition is used to prevent overflow and ensure that A stays within the 16-bit limit after every update. By applying the modulus 2^{16}, the result of the addition is guaranteed to remain a 16-bit value.

3. Deductive Argument Without the use of modular arithmetic, the sum $A_U = (A + \text{function}(B, C, D) + W[i])$ could exceed 16 bits, leading to overflow errors and inconsistent buffer values. The modulus operation ensures that all buffer values remain within the 16-bit range, which is critical for maintaining the integrity of the algorithm [6, 7]. Modular arithmetic ensures the uniformity and consistency of the hash output, even when different messages are processed.

4. Conclusion Modular arithmetic guarantees that buffer values never exceed the 16-bit limit, keeping the results consistent with the algorithm's design and preventing overflow errors.

Circular Shifts

1. Assumptions In Round 1, a 3-bit left circular shift is applied to compute the value of A_S after the modular arithmetic has been performed to find A_U. In Round 2, a 5-bit left circular shift is applied to introduce further complexity into the algorithm.

2. Circular Shift Properties A circular shift moves bits to the left, with the leftmost bits being reintroduced at the right end. Circular shifts are applied to the buffer values, with a 3-bit shift in Round 1 and a 5-bit shift in Round 2 [7]. This operation ensures that no information is lost, and the bit structure is maintained.

Deductive Argument For example, performing a 3-bit left circular shift on $A = 71A9$ in Round 1 results in:

$$A_S = ((71A9 \ll 3) \wedge FFFF) \vee (71A9 \gg 13) = 8D4B$$

Similarly, a 5-bit left circular shift in Round 2 moves the bits appropriately. The formula for Round 2 is:

$$A_S = ((71A9 \ll 5) \wedge FFFF) \vee (71A9 \gg 11) = 8D4B$$

Please refer to Section 1.1, General Steps of the Mini-MD5 Algorithm, for further clarification regarding the above formula.

3. Conclusion The circular shifts guarantee that the bits are mixed without any loss of information.

Buffer Rotation

1. Assumptions After the circular shift operation, the buffer values A, B, C, D are rotated to introduce further complexity and randomness within the algorithm. This ensures that each buffer plays a role in subsequent operations. The rotation pattern is as follows:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

2. Buffer Rotation Process The values of the buffers shift so that the result of each computation affects all the subsequent buffers in conjunction with the next word, $W[i]$, in the sequence [6, 7].

3. Deductive Argument Buffer rotation ensures that all buffers participate equally in the transformation process. Without buffer rotation, only one buffer (e.g., A) would be consistently updated, while the others would remain unchanged, resulting in a weaker and incomplete hash. By rotating the buffers, each value is transformed by multiple rounds and operations, ensuring the full diffusion of the message throughout the buffers. This diffusion is critical to the cryptographic strength of the algorithm.

4. Conclusion Buffer rotation ensures that all buffer values are transformed and updated across the operations. Without buffer rotation, only buffer A would be updated consistently, while the others would remain unchanged [6, 7].

Final Mini-MD5 Hash Output

1. Assumptions After two rounds, the buffer values A, B, C, D are concatenated to form the final 64-bit hash.

2. Deductive Argument After completing two rounds, if the final buffer values are:

$$A = \texttt{5A69}, \quad B = \texttt{1D51}, \quad C = \texttt{3031}, \quad D = \texttt{24FF}$$

The final hash is formed by concatenating the buffer values:

$$\text{Hash} = A||B||C||D = \texttt{5A691D51303124FF}$$

3. Conclusion The Mini-MD5 system transforms the input message into a valid 64-bit hash. These explanations demonstrate that the use of XOR functions $F(B, C, D)$ and $G(B, C, D)$, modular arithmetic, and circular shifts guarantees the integrity and correctness of the hashing process.

11.6 Mathematical Calculations of Mini MD5

The following calculations show the complete two rounds of the Mini-MD5 algorithm. As we proceed, we will reference specific elements from the explanations outlined in **Sections 1.1** and **1.2** to confirm that each step is logically sound and adheres to the cryptographic principles discussed.

For this example, we have been tasked with producing a **64-bit hash value** for the message "**MATH**".

Step 1: Initial Setup and Message Padding

We begin with the original message "MATH". Converting this message into its binary representation yields:

M	A	T	H
01001101	01000001	01010100	01001000

Since each letter is represented by 8 bits, the total original message length is calculated as:

$$4 \times 8 = 32\text{-bits}$$

As hash values are always represented in hexadecimal, we convert the binary representation to hexadecimal:

M	A	T	H
4D	41	54	48

Thus, the message becomes:

$$M = \texttt{4D415448}$$

Since this message is 32 bits long, it does not currently fit into the required 256-bit block. The message is padded to a 256-bit block, ensuring that the algorithm can process the message in a fixed-length format [6]. To do this, we apply message padding as described in Section 1.1. According to the message padding scheme, we append:

- `80` after the message (the hexadecimal representation of `10000000`) to signify the end of the original message.
- Sufficient `0` bits to bring the total length to 224 bits.
- The final 32 bits represent the original message length (32 bits) in hexadecimal. Since the original message length is 32 bits, we convert 32 to its hexadecimal value, which is `20`.

The padded message becomes:

$$M_{\text{padded}} = \texttt{0x4D41544880000000000000000000000000000000000000000000000000000020}$$

The padded message is then divided into sixteen 16-bit blocks $W[0], \ldots, W[15]$:

W[0] : 0x4D41	**W[1] : 0x5448**	**W[2] : 0x8000**	**W[3] : 0x0000**
W[4] : 0x0000	**W[5] : 0x0000**	**W[6] : 0x0000**	**W[7] : 0x0000**
W[8] : 0x0000	**W[9] : 0x0000**	**W[10] : 0x0000**	**W[11] : 0x0000**
W[12] : 0x0000	**W[13] : 0x0000**	**W[14] : 0x0000**	**W[15] : 0x0020**

Additional Notes on Message Padding It should be noted that the first half of $W[2]$ contains the appended 1-byte (`80`) to signify the end of the original message. $W[14]$ and $W[15]$ represent the hexadecimal representation of the original message length.

Addressing Explanation (Message Padding) This padding step ensures that the message is correctly formatted into a 256-bit block, consisting of sixteen 16-bit sub-blocks. The appended `80` byte and the final 32 bits representing the original length guarantee that the message is processed as per the Explanation of Message Padding.

Step 2: Initial Buffer Values

We initialize our four fixed 16-bit buffer values as described in Section 1.1:

$$A = \texttt{1234}, \quad B = \texttt{5678}, \quad C = \texttt{9ABC}, \quad D = \texttt{DEF0}$$

Step 3: Round 1 (with $F(B, C, D)$)

In the first round, we use the XOR function:

$$F(B, C, D) = B \oplus C \oplus D$$

and apply a 3-bit left circular shift after each buffer update.

We start the round by computing $W[0]$ across our four operations to calculate the updated buffer values, which will then be used in conjunction with $W[1]$.

$\boldsymbol{W[0] = 4D41}$

Operation 1: XOR Function Compute $F(B, C, D) = B \oplus C \oplus D$:

$$F(B, C, D) = F(5678, 9ABC, DEF0) = 5678 \oplus 9ABC \oplus DEF0 = 1234$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = 1234, \quad F(B, C, D) = 1234, \quad W[i] = W[0] = 4D41$$

In base 10:

$$A_U = (4660 + 4660 + 19777) \mod 2^{16} = 29097 \mod 2^{16} = 29097$$

Converting to base 16:

$$29097 \text{ (base 10)} = 71A9 \text{ (base 16)}$$

In base 16 (for simplicity in future iterations):

$$A_U = (1234 + 1234 + 4D41) \mod 2^{16} = 71A9$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = 71A9$ with a shift amount of 3 (for Round 1):

$$A_S = ((71A9 \ll 3) \wedge 0xFFFF) \vee (71A9 \gg 13) = 8D4B$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A = 1234, \quad B = 5678, \quad C = 9ABC, \quad D = DEF0$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = DEF0, \quad B = 8D4B, \quad C = 5678, \quad D = 9ABC$$

These buffer values are now carried over to our operations involving $W[1]$.

Addressing Explanation (Buffer Rotation) Buffer rotation ensures that the buffer values shift accordingly. This step guarantees that all buffers are transformed evenly, as described in the Explanation of Buffer Rotation.

$W[1] = 5448$

Operation 1: XOR Function Compute $F(B, C, D) = B \oplus C \oplus D$:

$$F(B, C, D) = F(8D4B, 5678, 9ABC) = 8D4B \oplus 5678 \oplus 9ABC = 418F$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{DEF0}, \quad F(B, C, D) = \texttt{418F}, \quad W[i] = W[1] = \texttt{5448}$$
$$A_U = (\texttt{DEF0} + \texttt{418F} + \texttt{5448}) \mod 2^{16} = \texttt{74C7}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{74C7}$ with a shift amount of 3 (for Round 1):

$$A_S = ((\texttt{74C7} \ll 3) \wedge 0xFFFF) \vee (\texttt{74C7} \gg 13) = \texttt{A63B}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{A63B}, \quad B = \texttt{8D4B}, \quad C = \texttt{5678}, \quad D = \texttt{9ABC}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{9ABC}, \quad B = \texttt{A63B}, \quad C = \texttt{8D4B}, \quad D = \texttt{5678}$$

These buffer values are now carried over to our operations involving $W[2]$.

$W[2] = 8000$

Operation 1: XOR Function Compute $F(B, C, D) = B \oplus C \oplus D$:

$$F(B, C, D) = F(A63B, 8D4B, 5678) = A63B \oplus 8D4B \oplus 5678 = 7D08$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{9ABC}, \quad F(B, C, D) = \texttt{7D08}, \quad W[i] = W[2] = \texttt{8000}$$
$$A_U = (\texttt{9ABC} + \texttt{7D08} + \texttt{8000}) \mod 2^{16} = \texttt{97C4}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{97C4}$ with a shift amount of 3 (for Round 1):

$$A_S = ((\texttt{97C4} \ll 3) \wedge 0xFFFF) \vee (\texttt{97C4} \gg 13) = \texttt{BE24}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{BE24}, \quad B = \texttt{A63B}, \quad C = \texttt{8D4B}, \quad D = \texttt{5678}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{5678}, \quad B = \texttt{BE24}, \quad C = \texttt{A63B}, \quad D = \texttt{8D4B}$$

These buffer values are now carried over to our operations involving $W[3]$.

$W[3] = 0000$

Operation 1: XOR Function Compute $F(B, C, D) = B \oplus C \oplus D$:

$$F(B, C, D) = F(\texttt{BE24}, \texttt{A63B}, \texttt{8D4B}) = \texttt{BE24} \oplus \texttt{A63B} \oplus \texttt{8D4B} = \texttt{9554}$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{5678}, \quad F(B, C, D) = \texttt{9554}, \quad W[i] = W[3] = \texttt{0000}$$
$$A_U = (\texttt{5678} + \texttt{9554} + \texttt{0000}) \mod 2^{16} = \texttt{EBCC}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{EBCC}$ with a shift amount of 3 (for Round 1):

$$A_S = ((\texttt{EBCC} \ll 3) \wedge 0xFFFF) \vee (\texttt{EBCC} \gg 13) = \texttt{5E67}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{5E67}, \quad B = \texttt{BE24}, \quad C = \texttt{A63B}, \quad D = \texttt{8D4B}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{8D4B}, \quad B = \texttt{5E67}, \quad C = \texttt{BE24}, \quad D = \texttt{A63B}$$

These buffer values are now carried over to our operations involving $W[4]$.

Step 4: Round 2 (with $G(B, C, D)$)

For the second round, we switch to using the function:

$$G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$$

and apply a 5-bit left circular shift.

Addressing Explanation (XOR Functions)

Here, we apply the new function $G(B, C, D)$ to further mix the buffer values. As noted in the explanation, this new XOR function enhances the non-linearity of the algorithm, ensuring additional complexity in Round 2.

Operation 1: XOR Function Compute $G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$:

$$G(5E67, BE24, A63B) = (5E67 \oplus A63B) \oplus (BE24 \oplus A63B) = \texttt{E043}$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{8D4B}, \quad G(B, C, D) = \texttt{E043}, \quad W[i] = W[4] = \texttt{0000}$$
$$A_U = (\texttt{8D4B} + \texttt{E043} + \texttt{0000}) \mod 2^{16} = \texttt{6D8E}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{6D8E}$ with a shift amount of 5 (for Round 2):

$$A_S = ((\texttt{6D8E} \ll 5) \wedge 0xFFFF) \vee (\texttt{6D8E} \gg 11) = \texttt{B1CD}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{B1CD}, \quad B = \texttt{5E67}, \quad C = \texttt{BE24}, \quad D = \texttt{A63B}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{A63B}, \quad B = \texttt{B1CD}, \quad C = \texttt{5E67}, \quad D = \texttt{BE24}$$

These buffer values are now carried over to our operations involving $W[5]$.

$W[5] = 0000$

Operation 1: XOR Function Compute $G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$:

$$G(B, C, D) = G(\texttt{B1CD}, \texttt{5E67}, \texttt{BE24}) = (\texttt{B1CD} \oplus \texttt{BE24}) \oplus (\texttt{5E67} \oplus \texttt{BE24}) = \texttt{EFAA}$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{A63B}, \quad G(B, C, D) = \texttt{EFAA}, \quad W[i] = W[5] = \texttt{0000}$$
$$A_U = (\texttt{A63B} + \texttt{EFAA} + \texttt{0000}) \mod 2^{16} = \texttt{95E5}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{95E5}$ with a shift amount of 5 (for Round 2):

$$A_S = ((\texttt{95E5} \ll 5) \wedge 0xFFFF) \vee (\texttt{95E5} \gg 11) = \texttt{BCB2}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{BCB2}, \quad B = \texttt{B1CD}, \quad C = \texttt{5E67}, \quad D = \texttt{BE24}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{BE24}, \quad B = \texttt{BCB2}, \quad C = \texttt{B1CD}, \quad D = \texttt{5E67}$$

These buffer values are now carried over to our operations involving $W[6]$.

$W[6] = 0000$

Operation 1: XOR Function Compute $G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$:

$$G(B, C, D) = G(\texttt{BCB2}, \texttt{B1CD}, \texttt{5E67}) = (\texttt{BCB2} \oplus \texttt{B1CD}) \oplus (\texttt{5E67} \oplus \texttt{B1CD}) = \texttt{0D7F}$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{BE24}, \quad G(B, C, D) = \texttt{0D7F}, \quad W[i] = W[6] = \texttt{0000}$$
$$A_U = (\texttt{BE24} + \texttt{0D7F} + \texttt{0000}) \mod 2^{16} = \texttt{CBA3}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{CBA3}$ with a shift amount of 5 (for Round 2):

$$A_S = ((\texttt{CBA3} \ll 5) \wedge 0xFFFF) \vee (\texttt{CBA3} \gg 11) = \texttt{7479}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{7479}, \quad B = \texttt{BCB2}, \quad C = \texttt{B1CD}, \quad D = \texttt{5E67}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{5E67}, \quad B = \texttt{7479}, \quad C = \texttt{BCB2}, \quad D = \texttt{B1CD}$$

$W[7] = 0020$

Operation 1: XOR Function Compute $G(B, C, D) = (B \oplus D) \oplus (C \oplus D)$:

$$G(B, C, D) = G(\texttt{7479}, \texttt{BCB2}, \texttt{B1CD}) = (\texttt{7479} \oplus \texttt{B1CD}) \oplus (\texttt{BCB2} \oplus \texttt{B1CD}) = \texttt{C8CB}$$

Operation 2: Modular Arithmetic Find the value of A_U using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Substituting:

$$A = \texttt{5E67}, \quad G(B, C, D) = \texttt{C8CB}, \quad W[i] = W[7] = \texttt{0020}$$
$$A_U = (\texttt{5E67} + \texttt{C8CB} + \texttt{0020}) \mod 2^{16} = \texttt{2732}$$

Operation 3: Circular Shift Find the value of A_S using:

$$A_S = ((A_U \ll \text{shift\_amount}) \wedge 0xFFFF) \vee (A_U \gg (16 - \text{shift\_amount}))$$

Performing the circular shift on $A_U = \texttt{2732}$ with a shift amount of 5 (for Round 2):

$$A_S = ((\texttt{2732} \ll 5) \wedge 0xFFFF) \vee (\texttt{2732} \gg 11) = \texttt{E644}$$

Operation 4: Buffer Rotation Our current buffer values are:

$$A_S = \texttt{E644}, \quad B = \texttt{7479}, \quad C = \texttt{BCB2}, \quad D = \texttt{B1CD}$$

With the buffer rotation pattern:

$$A \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values become:

$$A = \texttt{B1CD}, \quad B = \texttt{E644}, \quad C = \texttt{7479}, \quad D = \texttt{BCB2}$$

Addressing Explanations of (XOR Function, Modular Arithmetic, and Circular Shift):

- We applied $G(B, C, D)$ as per the explanations in Section 1.2, ensuring that the buffer values are correctly mixed using the XOR operation.
- Modular arithmetic ensures that buffer A remains within the 16-bit range, consistent with the explained concepts.
- The 5-bit circular shift further mixes the bits, as demonstrated in the explanation.

Step 5: Final Hash Output

After completing both rounds, the final buffer values are:

$$A = \texttt{B1CD}, \quad B = \texttt{E644}, \quad C = \texttt{7479}, \quad D = \texttt{BCB2}$$

The final 64-bit hash is generated by concatenating the buffer values $A||B||C||D$, where $||$ references the concatenation of the buffer values [6, 7]:

$$\text{Hash} = A||B||C||D = \texttt{B1CDE6447479BCB2}$$

Thus, the final Mini-MD5 hash for the input message "MATH" is:

$$\texttt{B1CDE6447479BCB2}$$

Addressing Explanation (Final Hash Output):

- The concatenation of A, B, C, and D ensures that the final hash reflects the transformations applied to all buffers throughout the two rounds of the Mini-MD5 algorithm [6, 7].
- This final step ensures that the message's entire transformation history is included in the final output, meaning that the 64-bit hash is a secure and unique representation of the original message.
- Each buffer has undergone transformations from XOR functions, modular arithmetic, circular shifts, and buffer rotations, as established in the earlier proofs. The combination of these buffer values guarantees that the final hash output is non-trivial, irreversible, and secure.

Addressing Proof (XOR Function, Modular Arithmetic, and Circular Shift):

- We applied $F(B, C, D)$ as per the Explanation of XOR Functions in Section 1.2, ensuring that the buffer values are correctly mixed using the XOR operation.
- Modular arithmetic ensures that buffer A remains within the 16-bit range, consistent with the Explanation of Modular Arithmetic concepts.
- The 3-bit circular shift further mixes the bits, as demonstrated in the Explanation of Circular Shifts.

1.5 Conclusion of Mini-MD5 Analysis

In this section, we have thoroughly analysed the Mini-MD5 algorithm by addressing each of the six core mathematical explanations: message padding, XOR functions, modular arithmetic, circular shifts, buffer rotations, and the final hash output. Through a combination of theoretical reasoning, practical manual calculations, and SageMath code, we have demonstrated the correctness and integrity of the Mini-MD5 system.

Each step, from initial message processing to the final 64-bit hash output, aligns with cryptographic principles, ensuring that the algorithm achieves its goal of producing secure and irreversible hash values.

With this analysis, we conclude our in-depth study of the Mini-MD5 algorithm. A broader reflection on the Mini-MD5 system will be provided in the conclusion of this report.

2.1 General Steps of the Mini-SHA-1 Algorithm

Before we delve into the formal mathematical proofs of Mini-SHA-1, it is essential to understand the general steps of the algorithm and why each step is needed. The Mini-SHA-1 algorithm follows a simplified version of the SHA-1 cryptographic hash function, designed to transform an input message into a fixed-size 64-bit hash (NIST, 2011). This section provides a high-level overview of the Mini-SHA-1 algorithm and prepares us for the formal mathematical proofs and detailed calculations that follow.

Step 1: Message Padding

In the Mini-SHA-1 algorithm, both the message and block size are 32 bits. The padding process ensures that the message fills a single 32-bit block [7]. The padding is applied as follows:

- Appending a 1-bit to the message (to signify the end of the message),
- Adding enough 0-bits to reach 32 bits.

The resulting padded message is exactly 32 bits long, including the original message, the appended 1-bit, and the original message length [5]. For example, if the message is "MATH" (32 bits), it will be processed directly without requiring additional zero-padding.

Why is this needed?

- **Fixed block size:** Since both the message and block size are 32 bits, padding ensures that the message conforms to the 32-bit block size required by the Mini-SHA-1 system.
- **Message integrity:** The appended 1-bit signals the end of the actual message, while the final length field preserves the original size of the message in bits, ensuring consistency during hashing.

Step 2: Initialize Buffers

The algorithm starts by initializing 5 buffers with pre-defined 16-bit values:

$$A = \texttt{1234}, \quad B = \texttt{5678}, \quad C = \texttt{9ABC}, \quad D = \texttt{DEF0}, \quad E = \texttt{2468}$$

Why is this needed?

- **Starting state:** These buffers represent the state of the hash computation at any given point. The initial values are fixed and provide a consistent starting point for all messages.

Step 3: Divide the Message into Blocks

The message is split into 16-bit blocks, labelled as $W[0]$ and $W[1]$. For example, the message "MATH" in hexadecimal format will be divided as:

$$W[0] = \texttt{4D41}, \quad W[1] = \texttt{5448}$$

Why is this needed?

- **Data structure:** Splitting the message into smaller blocks allows the algorithm to process each block individually. This division is essential for applying the cryptographic transformations in subsequent rounds.

Step 4: Rounds of Processing

The Mini-SHA-1 system uses four XOR-based functions to process the message blocks. These functions are applied in four rounds, where each round applies one XOR function twice [7]. The XOR functions are:

$$F_1(B, C, D) = B \oplus C \oplus D, \quad F_2(B, C, D) = (B \oplus D) \oplus (C \oplus B),$$

$$F_3(B, C, D) = (B \oplus C) \oplus (C \oplus D), \quad F_4(B, C, D) = B \oplus C \oplus D$$

Note that F_1 and F_4 are the same for simplicity.

The message is processed over four rounds as follows:

- **Round 1:** F_1 is applied twice.
- **Round 2:** F_2 is applied twice.
- **Round 3:** F_3 is applied twice.
- **Round 4:** F_4 is applied twice.

Each round involves:

1. Computing the XOR function on buffers B, C, D.
2. Updating buffer A using:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

3. Performing a circular shift on buffer A:

$$A = ((A \ll \text{shift\_amount}) \wedge \texttt{FFFF}) \vee (A \gg (16 - \text{shift\_amount}))$$

   The shift amount is either 3 (for Rounds 1–2) or 5 (for Rounds 3–4).

4. Rotating the buffers:

$$A \to E, \quad B \to A_S, \quad C \to B, \quad D \to C, \quad E \to D$$

Steps 1–4 are repeated for the second function application in each round, substituting $W[1]$ for $W[0]$.

Why is this needed?

- **Mixing the data:** XOR functions ensure non-linear mixing of buffer values, providing cryptographic strength.
- **Circular shifts:** These shifts mix bits further while preserving buffer bit-lengths.
- **Modular arithmetic:** Maintains buffer values within the 16-bit range, ensuring consistency.
- **Buffer rotation:** Ensures all buffers are involved in subsequent operations.

Step 5: Producing the Final Hash Output

After completing all four rounds, the final values of buffers A, B, C, D are concatenated to produce a 64-bit hash:

$$\text{Hash} = A||B||C||D$$

(Note: Buffer E is discarded.)

Why is this needed?

- **Fixed-size output:** The final 64-bit hash provides a compact representation of the original message, ensuring message integrity.

2.2 Explanation of Mini-SHA-1

The Mini-SHA-1 algorithm, though simplified compared to the full SHA-1 algorithm, shares many structural similarities with the Mini-MD5 system. Given that SHA-1 was largely based on MD5, many of the explanation parameters follow the same format as the Mini-MD5 algorithm. However, important differences, such as block size, buffer values, and the number of rounds, must be accounted for in the explanation.

The following explanation will cover the same key parameters, adjusted to reflect the specifics of the Mini-SHA-1 system:

Statement of the Problem

We aim to show that the Mini-SHA-1 system transforms an input message M into a valid 64-bit hash using four rounds of operations [7, 5]. The algorithm uses four XOR functions $F_1(B, C, D)$, $F_2(B, C, D)$, $F_3(B, C, D)$, and $F_4(B, C, D)$, with alternating circular shifts and buffer rotations. Each step in the algorithm (message padding, XOR functions, modular arithmetic, circular shifts, and buffer rotations) contributes to the correct hash output.

This explanation covers the following steps:

1. **Message Padding:** Ensuring that any input message fits into a 32-bit block.
2. **XOR Functions F_1, F_2, F_3, F_4:** Demonstrating that the XOR functions mix the buffer values effectively.
3. **Modular Arithmetic:** Ensuring that operations remain within the 16-bit range.
4. **Circular Shifts:** Proving that 3-bit (for Rounds 1 and 2) and 5-bit (for Rounds 3 and 4) circular shifts mix the bits without loss.
5. **Buffer Rotation:** Ensuring that the buffer values are properly rotated after each operation to distribute transformations across the buffers.
6. **Final Hash Output:** Demonstrating that the final buffer values are concatenated to form the correct 64-bit hash.

Message Padding

1. Assumptions: The input message M has a length L (in bits), and we must ensure it fits into a 32-bit block for Mini-SHA-1.

2. Message Padding Process:

- Append 80 (the hexadecimal representation of a "10000000" byte) to the message, signifying the end of the message.
- Since both the message and the block size are 32 bits, no further padding is necessary, and the final block already includes the original message length.

3. Deductive Argument: Padding ensures that the message fits perfectly into a 32-bit block, which is the required size for the Mini-SHA-1 algorithm (NIST, 2011). Unlike larger block systems, no additional padding bits are required. The 80 byte signifies the end of the message, and the block remains the correct size for processing.

4. Conclusion: The message is padded to a 32-bit block, ensuring that it can be processed by the Mini-SHA-1 system. This step guarantees that the entire message is correctly accounted for in a fixed-length format, aligning with the requirements of the algorithm.

XOR Functions F_1, F_2, F_3, F_4

1. Assumptions: The Mini-SHA-1 algorithm uses XOR-based operations in each round to achieve cryptographic mixing of message blocks [7]. The algorithm employs four XOR-based functions to process buffer values across four rounds of operations. The functions used are:

$$F_1(B, C, D) = B \oplus C \oplus D$$
$$F_2(B, C, D) = (B \oplus D) \oplus (C \oplus B)$$
$$F_3(B, C, D) = (B \oplus C) \oplus (C \oplus D)$$
$$F_4(B, C, D) = B \oplus C \oplus D$$

2. XOR Function Process:

- In each round, the function F is applied twice. The values B, C, and D are transformed non-linearly using the XOR operation.
- The function computes the result of the XOR operation and passes the result on to update buffer A.

3. Deductive Argument: Each XOR function is designed to ensure non-linear mixing of the buffer values B, C, and D. By applying different XOR functions in each round, the algorithm achieves a high level of data complexity, preventing patterns from forming and enhancing the cryptographic strength of the system.

4. Conclusion: The XOR functions effectively mix the buffer values in each round. By applying a different XOR-based function per round, the system guarantees that the message undergoes secure, non-linear transformations throughout the hashing process.

Modular Arithmetic

1. Assumptions: After applying the XOR function, buffer A is updated using modular arithmetic to prevent overflow beyond the 16-bit limit [7, 5]. The modular arithmetic ensures that the buffer remains within the 16-bit range.

2. Modular Arithmetic Process:

1. The buffer A is updated to value A_U according to the formula:

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

2. This step ensures that the result of the update stays within the 16-bit limit.

3. Deductive Argument: Modular arithmetic prevents overflow, ensuring that the buffer values A, B, C, D, and E remain consistent across operations. The modulus 2^{16} guarantees that no buffer exceeds the allowed bit size, preserving the integrity of the algorithm.

4. Conclusion: Modular arithmetic is applied after each buffer update to ensure that the result remains within the 16-bit range. This step is essential for maintaining the consistency and security of the Mini-SHA-1 algorithm.

Circular Shifts

1. Assumptions: After computing A_U, a left circular shift is applied to further mix the bits. The number of bits shifted depends on the round:

- In Rounds 1 and 2, a 3-bit left circular shift is applied.
- In Rounds 3 and 4, a 5-bit left circular shift is applied [7, 5].

2. Circular Shift Process: The circular shift is performed as follows:

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \mid (A_U \gg (16 - \text{shift\_amount}))$$

- For Rounds 1 and 2, the shift amount is 3.
- For Rounds 3 and 4, the shift amount is 5.

3. Deductive Argument: The circular shift ensures that the bits in buffer A are further mixed, maintaining the 16-bit structure of the buffer while preventing information loss. Bits shifted out of the left end are wrapped around to the right, ensuring that all bits are retained and mixed effectively.

4. Conclusion: The circular shift helps to randomize the bits within buffer A, ensuring that the buffer remains unpredictable while preserving the 16-bit limit. This step strengthens the cryptographic properties of the Mini-SHA-1 algorithm.

Buffer Rotation

1. Assumptions: After each operation, the buffer values A, B, C, D, and E are rotated to ensure that all buffers participate in the transformation process [7**?**].

2. Buffer Rotation Process:

- After each operation, the rotation follows this pattern:

$$A \to E, \quad B \to A_S, \quad C \to B, \quad D \to C, \quad E \to D$$

- This rotation ensures that each buffer value is transformed and updated in subsequent operations.

3. Deductive Argument: Buffer rotation ensures that the transformation results are distributed evenly across all buffers. Without rotation, only one buffer would be consistently updated, while the others would remain largely unchanged. Rotation prevents redundancy and ensures that all buffers undergo multiple transformations throughout the rounds.

4. Conclusion: Buffer rotation is a key aspect of the Mini-SHA-1 algorithm, ensuring that all buffers are updated across operations. This step guarantees that each buffer contributes equally to the final hash output.

Final Hash Output

1. Assumptions: After completing all four rounds, the final buffer values A, B, C, and D are concatenated to form the final 64-bit hash.

2. Final Hash Calculation: The final hash is produced by concatenating the 16-bit buffer values [7**?**]:

$$\text{Hash} = A \,||\, B \,||\, C \,||\, D$$

3. Deductive Argument: The final hash combines the results of all rounds, ensuring that the full transformation history of the message is represented in the output. Each buffer value contributes to the final 64-bit hash, ensuring that the message has been fully transformed by the cryptographic operations.

4. Conclusion: The final 64-bit hash is a secure representation of the original message. By combining the buffer values after all rounds of operations, the Mini-SHA-1 algorithm produces a fixed-size output that reflects the cryptographic transformations applied during the hashing process.

This explanation demonstrates the correctness of the Mini-SHA-1 algorithm by covering key areas: message padding, XOR functions, modular arithmetic, circular shifts, buffer rotation, and final hash output. These steps ensure that the algorithm operates effectively and produces secure hash outputs.

2.3 Mathematical Calculations of Mini-SHA-1

In this section, we will demonstrate the manual calculations of the Mini-SHA-1 algorithm. These calculations follow the steps outlined in the theoretical section and will be applied to the message "MATH". At each step, we will address the relevant proofs to ensure the correctness of the operations, covering message padding, XOR functions, modular arithmetic, circular shifts, buffer rotations, and the final hash output.

Step 1: Initial Setup and Message Padding

We begin with the original message "MATH". We can convert this into its binary representation as follows:

M	A	T	H
01001101	01000001	01010100	01001000

Since each letter is represented by 8 bits, the total original message length is:

$$4 \times 8 = 32 \text{ bits}$$

As hash values are always represented in hexadecimal, the binary value is converted to:

M	A	T	H
$4D$	41	54	48

The input message "MATH" is represented in hexadecimal as:

$$M = \texttt{4D415448}$$

This is a 32-bit message. Since the block size for Mini-SHA-1 is also 32 bits, no additional padding is required.

Addressing Explanation (Message Padding): As per the Explanation of Message Padding, the input message is padded to fit exactly into a 32-bit block. The padding ensures that the message can be processed correctly by the algorithm (NIST, 2011).

Step 2: Initial Buffer Values

We initialize the buffers as follows:

$$A = \texttt{1234}, \quad B = \texttt{5678}, \quad C = \texttt{9ABC}, \quad D = \texttt{DEF0}, \quad E = \texttt{2468}$$

Step 3: Divide the Message into Blocks

The message is split into 16-bit blocks, which are labeled as $W[0]$ and $W[1]$.

We then break down the 32-bit block into two 16-bit words:

$$W[0] = \texttt{4D41}, \quad W[1] = \texttt{5448}$$

Step 4: Rounds of Processing

Round 1 (Using $F_1(B, C, D)$ and 3-bit Circular Shift) In Round 1, we use the function $F_1(B, C, D) = B \oplus C \oplus D$ and apply a 3-bit left circular shift after updating buffer A.

W[0] = 4D41 **Operation 1: XOR Function:** Compute $F_1(B, C, D) = B \oplus C \oplus D$

$$F_1(B, C, D) = F_1(5678, 9ABC, DEF0) = 5678 \oplus 9ABC \oplus DEF0 = 1234$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = 1234, \quad F_1(B, C, D) = 1234, \quad \text{and} \quad W[i] = W[0] = 4D41$$

we substitute into the formula:

$$A_U = (1234 + 1234 + 4D41) \mod 2^{16} = 71A9$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 1, we need to shift the values to the left by 3:

$$A_S = ((71A9 \ll 3) \& 0xFFFF) \,|\, (71A9 \gg 13) = 8D4B$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = 8D4B, \quad B = 5678, \quad C = 9ABC, \quad D = DEF0, \quad E = 2468$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = 2468, \quad B = 8D4B, \quad C = 5678, \quad D = 9ABC, \quad E = DEF0$$

We now carry these buffer values over to our operations involving $W[1]$.

Addressing Explanation (XOR Functions, Modular Arithmetic, and Circular Shift):

- The XOR function $F_1(B, C, D)$ was used to effectively mix the buffer values, as per the Explanation of XOR Functions.
- Modular arithmetic was applied to keep A_U within 16 bits, aligning with the Explanation of Modular Arithmetic.
- The 3-bit circular shift further mixed the bits, following the Explanation of Circular Shifts.

W[1] = 5448 **Operation 1: XOR Function:** Compute $F_1(B, C, D) = B \oplus C \oplus D$

$$F_1(B, C, D) = F_1(8D4B, 5678, 9ABC) = 8D4B \oplus 5678 \oplus 9ABC = 418F$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = 2468, \quad F_1(B, C, D) = 418F, \quad \text{and} \quad W[i] = W[1] = 5448$$

we substitute into the formula:

$$A_U = (2468 + 418F + 5448) \mod 2^{16} = B338$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 1, we need to shift the values to the left by 3:

$$A_S = ((B338 \ll 3) \& 0xFFFF) \,|\, (B338 \gg 13) = 99C5$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = 99C5, \quad B = 8D4B, \quad C = 5678, \quad D = 9ABC, \quad E = DEF0$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = DEF0, \quad B = 99C5, \quad C = 8D4B, \quad D = 5678, \quad E = 9ABC$$

We now carry these buffer values over to Round 2 of the $F_2(B, C, D)$ operations involving $W[0]$.

Step 4: Round 2 (Using $F_2(B, C, D)$ and 3-bit Circular Shift) In Round 2, we use the function $F_2(B, C, D) = (B \oplus D) \oplus (C \oplus B)$ and apply another 3-bit left circular shift after updating buffer A.

W[0] = 4D41 **Operation 1: XOR Function:** Compute $F_2(B, C, D) = (B \oplus D) \oplus (C \oplus B)$

$$F_2(B, C, D) = F_2(99C5, 8D4B, 5678) = (99C5 \oplus 5678) \oplus (8D4B \oplus 99C5) = DB33$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = DEF0, \quad F_2(B, C, D) = DB33, \quad \text{and} \quad W[i] = W[0] = 4D41$$

we substitute into the formula:

$$A_U = (DEF0 + DB33 + 4D41) \mod 2^{16} = E6B$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 2, we need to shift the values to the left by 3:

$$A_S = ((E6B \ll 3) \& 0xFFFF) \,|\, (E6B \gg 13) = 7358$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = 7358, \quad B = 99C5, \quad C = 8D4B, \quad D = 5678, \quad E = 9ABC$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = 9ABC, \quad B = 7358, \quad C = 99C5, \quad D = 8D4B, \quad E = 5678$$

We now carry these buffer values over to our repeating of the $F_2(B, C, D)$ operations involving $W[1]$.

W[1] = 5448 **Operation 1: XOR Function:** Compute $F_2(B, C, D) = (B \oplus D) \oplus (C \oplus B)$

$$F_2(B, C, D) = F_2(7358, 99C5, 8D4B) = (7358 \oplus 8D4B) \oplus (99C5 \oplus 7358) = 148E$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = 9ABC, \quad F_2(B, C, D) = 148E, \quad \text{and} \quad W[i] = W[1] = 5448$$

we substitute into the formula:

$$A_U = (9ABC + 148E + 5448) \mod 2^{16} = 392$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 2, we need to shift the values to the left by 3:

$$A_S = ((392 \ll 3) \& 0xFFFF) \,|\, (392 \gg 13) = 1C90$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = 1C90, \quad B = 7358, \quad C = 99C5, \quad D = 8D4B, \quad E = 5678$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = 5678, \quad B = 1C90, \quad C = 7358, \quad D = 99C5, \quad E = 8D4B$$

We now carry these buffer values over to Round 3 of the $F_3(B, C, D)$ operations involving $W[0]$.

Step 4: Round 3 (Using $F_3(B, C, D)$ and 5-bit Circular Shift) In Round 3, we use the function $F_3(B, C, D) = (B \oplus C) \oplus (C \oplus D)$ and apply a 5-bit left circular shift after updating buffer A.

W[0] = 4D41 **Operation 1: XOR Function:** Compute $F_3(B, C, D) = (B \oplus C) \oplus (C \oplus D)$

$$F_3(B, C, D) = F_3(1C90, 7358, 99C5) = (1C90 \oplus 7358) \oplus (7358 \oplus 99C5) = 8555$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = 5678, \quad F_3(B, C, D) = 8555, \quad \text{and} \quad W[i] = W[0] = 4D41$$

we substitute into the formula:

$$A_U = (5678 + 8555 + 4D41) \mod 2^{16} = 290E$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 3, we need to shift the values to the left by 5:

$$A_S = ((290E \ll 5) \& 0xFFFF) \,|\, (290E \gg 11) = 4871$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = 4871, \quad B = 1C90, \quad C = 7358, \quad D = 99C5, \quad E = 8D4B$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = 8D4B, \quad B = 4871, \quad C = 1C90, \quad D = 7358, \quad E = 99C5$$

We now carry these buffer values over to our repeating of the $F_3(B, C, D)$ operations involving $W[1]$.

W[1] = 5448 **Operation 1: XOR Function:** Compute $F_3(B, C, D) = (B \oplus C) \oplus (C \oplus D)$

$$F_3(B, C, D) = F_3(4871, 1C90, 7358) = (4871 \oplus 1C90) \oplus (1C90 \oplus 7358) = 3B29$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = 8D4B, \quad F_3(B, C, D) = 3B29, \quad \text{and} \quad W[i] = W[1] = 5448$$

we substitute into the formula:

$$A_U = (8D4B + 3B29 + 5448) \mod 2^{16} = 15B5$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 3, we need to shift the values to the left by 5:

$$A_S = ((15B5 \ll 5)\&0xFFFF) \,|\, (15B5 \gg 11) = ADA8$$

Operation 4: Buffer Rotation: Our current buffer values are:

$$A_S = ADA8, \quad B = 4871, \quad C = 1C90, \quad D = 7358, \quad E = 99C5$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = 99C5, \quad B = ADA8, \quad C = 4871, \quad D = 1C90, \quad E = 7358$$

We now carry these buffer values over to Round 4 of the $F_4(B, C, D)$ operations involving $W[0]$.

Step 4: Round 4 (Using $F_4(B, C, D)$ and 5-bit Circular Shift) In Round 4, we use $F_4(B, C, D) = B \oplus C \oplus D$ and apply a 5-bit left circular shift after updating buffer A.

Operation 1: XOR Function: Compute $F_4(B, C, D) = B \oplus C \oplus D$

$$F_4(B, C, D) = F_4(\texttt{ADA8}, \texttt{4871}, \texttt{1C90}) = \texttt{ADA8} \oplus \texttt{4871} \oplus \texttt{1C90} = \texttt{F949}$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = \texttt{99C5}, \quad F_4(B, C, D) = \texttt{F949}, \quad \text{and} \quad W[i] = W[0] = \texttt{4D41}$$

we substitute into the formula:

$$A_U = (\texttt{99C5} + \texttt{F949} + \texttt{4D41}) \mod 2^{16} = \texttt{E756}$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount})\&0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 4, we need to shift the values to the left by 5:

$$A_S = ((\texttt{E756} \ll 5)\&0xFFFF) \,|\, (\texttt{E756} \gg 11) = \texttt{3AB7}$$

Operation 4: Buffer Rotation:

Our current buffer values are:

$$A_S = \texttt{3AB7}, \quad B = \texttt{ADA8}, \quad C = \texttt{4871}, \quad D = \texttt{1C90}, \quad E = \texttt{7358}$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = \texttt{7358}, \quad B = \texttt{3AB7}, \quad C = \texttt{ADA8}, \quad D = \texttt{1C90}, \quad E = \texttt{7358}$$

We now carry these buffer values over to our repeating of the $F_4(B, C, D)$ operations involving $W[1]$.

W[1] = 5448 **Operation 1: XOR Function:** Compute $F_4(B, C, D) = B \oplus C \oplus D$

$$F_4(B, C, D) = F_4(\texttt{3AB7}, \texttt{ADA8}, \texttt{1C90}) = \texttt{3AB7} \oplus \texttt{ADA8} \oplus \texttt{1C90} = \texttt{DF6E}$$

Operation 2: Modular Arithmetic: Find the value of A_U using

$$A_U = (A + \text{function}(B, C, D) + W[i]) \mod 2^{16}$$

Since:

$$A = \texttt{7358}, \quad F_4(B, C, D) = \texttt{DF6E}, \quad \text{and} \quad W[i] = W[1] = \texttt{5448}$$

we substitute into the formula:

$$A_U = (\texttt{7358} + \texttt{DF6E} + \texttt{5448}) \mod 2^{16} = \texttt{A70E}$$

Operation 3: Circular Shift: Find the value of A_S using

$$A_S = ((A_U \ll \text{shift\_amount}) \& 0xFFFF) \,|\, (A_U \gg (16 - \text{shift\_amount}))$$

As we are in Round 4, we need to shift the values to the left by 5:

$$A_S = ((\texttt{A70E} \ll 5) \& 0xFFFF) \,|\, (\texttt{A70E} \gg 11) = \texttt{3875}$$

Operation 4: Buffer Rotation:

Our current buffer values are:

$$A_S = \texttt{3875}, \quad B = \texttt{3AB7}, \quad C = \texttt{4871}, \quad D = \texttt{1C90}, \quad E = \texttt{7358}$$

As our buffer rotation pattern is:

$$A \to E, \quad E \to D, \quad D \to C, \quad C \to B, \quad B \to A_S$$

The updated buffer values are:

$$A = \texttt{1C90}, \quad B = \texttt{3875}, \quad C = \texttt{3AB7}, \quad D = \texttt{ADA8}, \quad E = \texttt{4871}$$

At this point, we have concluded Step 4 and can proceed to computing our final hash value.

Step 5: Final Hash Output After completing all 4 rounds, the final buffer values are:

$$A = \texttt{1C90}, \quad B = \texttt{3875}, \quad C = \texttt{3AB7}, \quad D = \texttt{ADA8}, \quad E = \texttt{4871}$$

The final 64-bit hash is generated by concatenating the buffer values $A||B||C||D$, where $||$ denotes concatenation:

$$\text{Hash} = A||B||C||D = \texttt{1C9038753AB7ADA8}$$

Thus, the final Mini-SHA-1 hash for the input message "MATH" is:

```
1C9038753AB7ADA8
```

Addressing Explanation (Final Hash Output): The final 64-bit hash is formed by concatenating the buffer values A, B, C, D, demonstrating the correctness of the Mini-SHA-1 system, as per the Explanation of Final Hash Output.

Conclusion

In this report, we embarked on an exploration of two simplified cryptographic systems: **Mini-MD5** and **Mini-SHA-1**, both modeled after the full MD5 and SHA-1 algorithms. These systems were carefully crafted to offer a hands-on, educational experience in understanding the fundamental principles behind cryptographic hashing. Through the development of these mini-systems, we were able to delve deeply into the core mathematical components such as XOR functions, modular arithmetic, message padding, circular shifts, and buffer rotations, while also aligning the simplified systems with cryptographic principles used in real-world hashing algorithms.

The **Mini-MD5** system demonstrated how reducing buffer sizes and rounds still preserves the essence of the original MD5 cryptographic process. By using simplified XOR-based functions, we observed how buffer updates and rotations function to transform a message into a secure 64-bit hash. Each step was backed by mathematical proofs, which we supplemented using **SageMath** code. This allowed us to validate the theoretical constructs with practical implementation, confirming that the mini-system follows the same essential logic as the full MD5 algorithm, albeit in a more concise form.

On the other hand, the **Mini-SHA-1** system takes a different approach compared to Mini-MD5. While both systems share some foundational cryptographic elements, the Mini-SHA-1 design introduces unique structural choices. These structural differences emphasize buffer rotation and additional non-linear mixing through its alternating XOR functions. The greater emphasis on round complexity ensures that each step spreads the influence of the message throughout the buffer states.

In both systems, our mathematical proofs established that the mini-algorithms effectively condense the cryptographic processes while ensuring integrity, security, and non-linearity in the transformations. The proofs demonstrated that modular arithmetic kept operations within the correct bounds, while circular shifts and buffer rotations ensured that the message data was thoroughly mixed across each round. The final hash outputs from both Mini-MD5 and Mini-SHA-1 were thus secure, consistent representations of the input data.

In conclusion, the study of **Mini-MD5** and **Mini-SHA-1** provided a detailed understanding of how cryptographic hash functions operate. By reducing these systems to their core components, we gained insight into how modern cryptographic techniques ensure data integrity and security. The **SageMath** code further reinforced the theoretical understanding by offering practical execution of the algorithms, thereby ensuring that the systems we developed are both mathematically sound and operationally secure.

About the Author

Casey Wilfling is a first year Bachelor of Cybersecurity (Honours) student specialising in Network Security and Penetration Testing. He enjoys board/video games, AFL, and spending time with his young family.

Acknowledgements

I would like to acknowledge all the staff at Deakin University who have provided a learning environment where I truly believe I can reach my potential. Special thanks also go to Guillermo Pineda Villavicencio for his supportive feedback throughout the development of this project.

References

[1] D. Eastlake and P. Jones. *RFC3174: US Secure Hash Algorithm 1 (SHA1).* RFC Editor, 2001.

[2] J. Katz and Y. Lindell. *Introduction to Modern Cryptography, 3rd ed.* CRC Press, 2020.

[3] W. Mao. *Modern Cryptography: Theory and Practice.* Prentice Hall, 2004.

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[5] National Institute of Standards and Technology . SHA-1 Secure Hash Algorithm. Technical report, NIST Computer Security Resource Center, 2015. Available online: https://csrc.nist.gov/publications/detail/fips/180/4/final.

[6] R. Rivest. *RFC1321: The MD5 Message-Digest Algorithm.* RFC Editor, 1992.

[7] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed.* John Wiley and Sons, 1996.

[8] W. Stallings. *Cryptography and Network Security: Principles and Practice, 7th ed.* Pearson, 2017.

12

A cost effective framework for the design and deployment of B5G in rural Australia

Brianna Laird

Abstract

This report introduces a novel optimisation framework for the design and deployment of B5G networks in rural and remote areas. The framework leverages Mixed Integer Linear Programming (MILP) to determine the optimal placement of Radio Units (RUs), Distributed Units (DUs), Centralised Units (CUs) and fibre paths to minimise deployment costs while meeting coverage requirements. The model is evaluated using CPLEX, Greedy, Genetic Algorithm (GA), and Local Search (LS) solvers to identify the most effective approach for cost-efficient network deployment. The simulation results demonstrate the framework's effectiveness in minimising deployment costs and providing optimal network coverage in rural regions, marking a significant advancement in network planning for underserved areas.

12.1 Introduction

The exponential growth in connected devices, real-time applications, and data-hungry users is pushing the limits of 5G networks. Beyond 5G (B5G) or 6G networks are being researched to address these demands by offering ultra-fast data rates, minimal latency and massive connectivity, essential for supporting advancements such as virtual reality, tactile internet and autonomous systems. As 5G evolves, it is clear that existing infrastructure cannot meet future demands. To address the challenges of scalability, energy efficiency, and flexibility, 6G will require novel approaches to meet these demands. Open RAN (O-RAN) is one such approach that aims to address these challenges, introduced by the O-RAN Alliance in 2018 [5], O-RAN divides traditional network components into the Central Unit (CU), Distributed Unit (DU), and Radio Unit (RU), which allows for more flexible and efficient network management and deployment.

In this disaggregated architecture, shown in Figure 12.1, RUs are responsible for wireless communication with user devices, DUs handle data processing closer to the network edge to reduce latency, and CUs manage network control and higher-layer protocols. These components must be strategically placed to provide efficient, scalable, and cost-effective network coverage, a task that becomes particularly complex in rural and remote regions. This motivates the need for an optimisation framework that can determine the optimal placement of these key network components.

For example, in regional Australia, inadequate mobile and internet service remains a persistent issue, with many areas either underserved or entirely disconnected. In such cases, connection costs, particularly for laying optical fibre, can easily outweigh the benefits due to the region's size and sparse population. This challenge highlights the need for an efficient and cost-effective

network deployment strategy, especially as B5G and 6G networks introduce increased complexity with the need for ultra-reliable, low-latency communication and higher bandwidth. In addition, balancing the deployment of optical fibre connections with wireless connectivity is crucial to ensuring network viability in these low-density areas.

This report introduces the first novel optimisation framework for network design and deployment that considers all O-RAN components, leveraging Mixed Integer Linear Programming (MILP to minimise the cost of deploying RUs, DUs, and optical fibre paths while meeting coverage requirements. Designed with flexibility, the framework accommodates geographic and economic constraints specific to rural areas and allows the integration of additional objectives or constraints. Its goal is to determine the optimal network topology by strategically placing and connecting RUs and DUs to achieve the desired coverage levels at minimal cost.

The framework's performance is evaluated with CPLEX, comparing its scalability and efficiency against heuristic algorithms, including Greedy, Genetic and Local Search approaches. While CPLEX ensures mathematically optimal solutions, heuristics are assessed for their speed and scalability in larger problem instances. This analysis identifies the most effective strategies for cost-efficient deployment of rural B5G networks, marking a significant step forward in network planning for underserved areas.

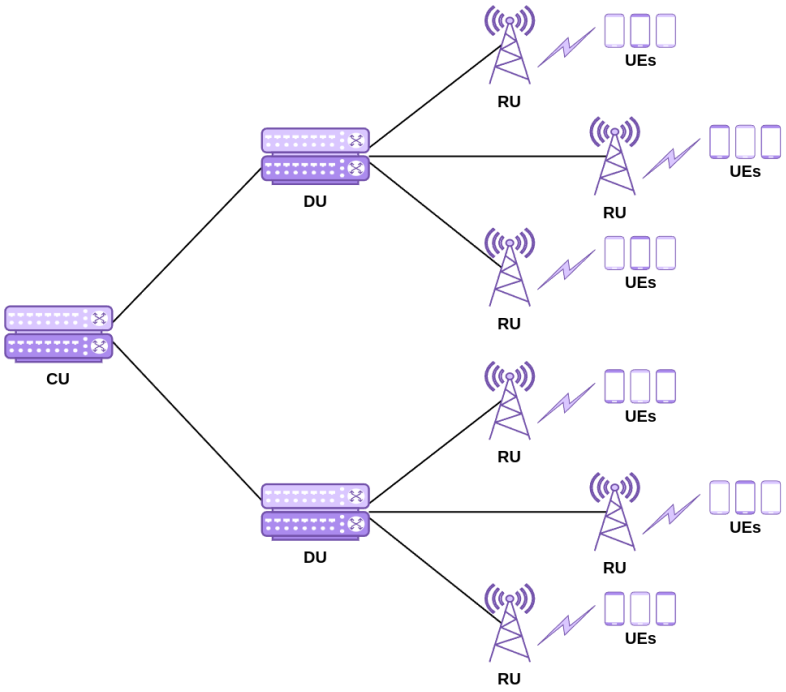


Figure 12.1: O-RAN Network structure diagram depicting the CU, DU, RU and users. Created using `draw.io`.

12.2 System Model

This section describes the network architecture and the model used to evaluate the network's deployment cost. The network consists of three main components: CU, DU, and RU. The CU is responsible for managing the network, the DU connects to the CU as a fibre connection, the RU connects to the DU also as a fibre connection, and the user connects wirelessly to the RU, as shown in Figure 12.1. The objective of the model is to minimise the total deployment cost while ensuring that user coverage and capacity requirements are met. This balance between cost

efficiency and network performance determines the optimal placement of RUs and DUs in the network.

Model Structure and Parameters

The model consists of several sets, parameters and decision variables which are summarised in Table 12.1. The framework includes set of users $\mathfrak{U}$, set of RUs $\mathfrak{R}$, set of DUs $\mathfrak{D}$, set of CUs $\mathfrak{C}$, set of fibre pieces $\mathfrak{F}$ and set of road segments $\mathfrak{S}$ where fibre pieces are installed along road segments to facilitate connectivity.

Objective Function

The objective of the MILP model is to minimise the total cost of deploying the network, this is the sum of the cost of the RUs, DUs, CU, their installation costs, fibre connection costs and fibre path cost. The objective function is defined as follows in (12.1).

$$\begin{aligned}\text{Minimise} \quad & \underbrace{\sum_{d\in\mathfrak{D}} \zeta_d \times K_{W_d} + \sum_{d\in\mathfrak{D}} \varrho_d - \zeta_d \times K_{X_d} + \sum_{r\in\mathfrak{R}}\sum_{d\in\mathfrak{D}} \mu_{r,d} \times K_{L_{r,d}}}_{\text{Total DU Cost}} \\ & + \underbrace{\sum_{r\in\mathfrak{R}} \varpi_r \times K_{V_r}}_{\text{Total RU Cost}} + \underbrace{\sum_{c\in\mathfrak{C}}\sum_{d\in\mathfrak{D}} \nu_{d,c} \times K_{B_{d,c}}}_{\text{Total CU Cost}} + \underbrace{\sum_{s\in\mathfrak{S}} \xi_s \times K_{S_s}}_{\text{Path Cost}} \end{aligned} \tag{12.1}$$

User and Coverage Constraints

Each user can only be connected to at most one RU at any given time, as specified by (12.2). Additionally, the user must be within the coverage range of that RU, as enforced by (12.3).

$$\sum_{r\in\mathfrak{R}} \alpha_{u,r} \leq 1 \quad \forall\, u \in \mathfrak{U} \tag{12.2}$$

$$\alpha_{u,r} \leq \delta_r \quad \forall\, u \in \mathfrak{U},\ \forall r \in \mathfrak{R} \tag{12.3}$$

To ensure the model meets varying levels of coverage requirements, the total number of users connected to the RUs must be at-least equal to the minimum user coverage requirement, as specified by (12.4).

$$\sum_{u\in\mathfrak{U}}\sum_{r\in\mathfrak{R}} \alpha_{u,r} \geq\ UC \tag{12.4}$$

This allows the model to be evaluated for different coverage scenarios, determining the optimal placement of RUs and DUs to meet the specified coverage requirements.

Device Activation Constraints

The model ensures that each active RU is connected to exactly one DU, as defined in (12.5), and that inactive RUs are not connected to any DU, as enforced by (12.6). Similarly, each active DU must be connected to exactly one CU, as specified by (12.7).

$$\sum_{d\in\mathfrak{D}} \beta_{r,d} = \delta_r \quad \forall\, r \in \mathfrak{R} \tag{12.5}$$

$$\beta_{r,d} \leq \zeta_d \quad \forall\, r \in \mathfrak{R},\ d \in \mathfrak{D} \tag{12.6}$$

$$\sum_{c\in\mathfrak{C}} \gamma_{d,c} = \zeta_d \quad \forall\, d \in \mathfrak{D} \tag{12.7}$$

Table 12.1: Notations used in the MILP model

Sets	
$\mathfrak{U}$	Set of users requiring network coverage
$\mathfrak{R}$	Set of potential Radio Unit (RU) installation locations
$\mathfrak{D}$	Set of potential Distributed Unit (DU) installation locations
$\mathfrak{C}$	Set of established Central Unit (CU) locations
$\mathfrak{F}$	Set of available fibre optic cable segments
$\mathfrak{S}$	Set of road segments available for network infrastructure
Parameters	
UM	Maximum bandwidth per user to be served by network
RC	Maximum capacity of each RU for connecting users
DC	Maximum capacity of each DU for handling RU connections
DP	Maximum number of physical ports available on each DU
FC	Maximum bandwidth capacity per fibre optic cable
MR	Maximum number of RUs that can be installed at a single location
MD	Maximum number of DUs that can be installed at a single location
CR	Maximum number of fibre connections allowed between a DU-CU pair
L_r	Geographic coordinates of potential RU installation locations
L_d	Geographic coordinates of potential DU installation locations
$P_{r,d}$	Set of feasible fibre paths connecting RU r to DU d
$P_{d,c}$	Set of feasible fibre paths connecting DU d to CU c
K_{V_r}	Installation and equipment cost for deploying RU r
K_{W_d}	Installation and equipment cost for deploying first DU d at location
K_{X_d}	Incremental cost for adding additional DU d at existing location
K_{B_c}	Installation and equipment cost for deploying CU c
K_L	Cost per fibre connection terminating at DU
K_B	Cost per fibre connection terminating at CU
Variables	
$\alpha_{u,r}$	Binary variable: 1 if user u is connected to RU r, 0 otherwise
$\beta_{r,d}$	Binary variable: 1 if RU r is connected to DU d, 0 otherwise
$\gamma_{d,c}$	Binary variable: 1 if DU d is connected to CU c, 0 otherwise
δ_r	Binary variable: 1 if RU r is activated, 0 otherwise
ζ_d	Binary variable: 1 if DU d is activated, 0 otherwise
ξ_s	Binary variable: 1 if segment s is used in any path, 0 otherwise
ϖ_r	Integer variable: Number of RUs installed at location r
$\mu_{r,d}$	Integer variable: Number of RU cells connected between RU r and DU d
ϱ_d	Integer variable: Total number of DUs installed at location d
$\eta_{d,c}$	Integer variable: Number of DU cells connected between DU d and CU c
$\nu_{d,c}$	Integer variable: Bandwidth scaling for connection between DU d and CU c

RUs Per Location Constraints

Each location can host a maximum number of RUs, as specified by (12.8). Equation (12.9) ensures that the total number of RUs allocated at a location matches the bandwidth multiplier for the RU to DU connection.

$$\mu_{r,d} \leq MR \times \beta_{r,d} \quad \forall r \in \mathfrak{R},\ \forall d \in \mathfrak{D} \tag{12.8}$$

$$\sum_{d \in \mathfrak{D}} \mu_{r,d} = \varpi_r \quad \forall r \in \mathfrak{R} \tag{12.9}$$

DUs Per Location Constraints

Each location can host a maximum number of DUs, as specified by (12.10). Equation (12.11) ensures that the total number of DUs allocated at a location matches the bandwidth multiplier for the DU to CU connection.

$$\eta_{d,c} \leq MD \times \gamma_{d,c} \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \tag{12.10}$$

$$\sum_{c \in \mathfrak{C}} \eta_{d,c} = \varrho_d \quad \forall d \in \mathfrak{D} \tag{12.11}$$

Capacity Constraints

Each device type in the network model has specific capacity limits that must be adhered to. Equations (12.12) and (12.13) ensure that these limits are not exceeded. Specifically, (12.12) ensures that the total bandwidth used by users connected to an RU does not exceed the RU's capacity, while equation (12.13) ensures that the total bandwidth from RUs connected to a DU does not exceed the DU's capacity.

$$\sum_{u \in \mathfrak{U}} \alpha_{u,r} \times UM \leq RC \times \varpi_r \quad \forall\ r \in \mathfrak{R} \tag{12.12}$$

$$\sum_{r \in \mathfrak{R}} \mu_{r,d} \times RC \leq DC \times \eta_{d,c} \quad \forall\ d \in \mathfrak{D} \tag{12.13}$$

Each device also has a limited number of ports. The model must ensure that the total number of RUs connected to each DU does not exceed the DU's port capacity, as specified by (12.14).

$$\sum_{r \in \mathfrak{R}} \mu_{r,d} \leq\ DP \times \eta_{d,c} \quad \forall\ d \in \mathfrak{D} \tag{12.14}$$

Fibre Connection Constraints

The model ensures that the total bandwidth from RUs to a DU does not exceed the fibre capacity, as defined in (12.15). Equation (12.16) adjusts the maximum fibre connections between a DU and CU based on the number of DUs at the location. Equation (12.17) ensures that the total bandwidth demand from all RUs passing through a DU stays within the adjusted fibre capacity.

$$\sum_{d \in D} \mu_{r,d} \times RC \leq FC \times \varpi_r \quad \forall\ r \in \mathfrak{R} \tag{12.15}$$

$$\eta_{d,c} \leq \nu_{d,c} \times CR \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \tag{12.16}$$

$$\sum_{r \in \mathfrak{R}} \mu_{r,d} \times RC \leq FC \times \nu_{d,c} \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \tag{12.17}$$

Shortest Path Constraints

In the optical fibre network model, road segments are shared by multiple devices but counted once for cost calculations. Equations (12.18) and (12.19) ensure that if a road segment is used in any RU to DU or DU to CU path, it is activated at least once and only counted once.

$$\xi_s \geq \beta_{r,d} \quad \forall\, r \in \mathfrak{R},\ \forall d \in \mathfrak{D},\ \forall h \in P_{r,d} \tag{12.18}$$

$$\xi_s \geq \gamma_{d,c} \quad \forall\, d \in \mathfrak{D},\ \forall c \in \mathfrak{C},\ \forall h \in P_{d,c} \tag{12.19}$$

Full Model

The full model can be summarised as follows:

$$\begin{aligned}
\text{Minimise} \quad & \underbrace{\sum_{d\in\mathfrak{D}} \zeta_d \times K_{W_d} + \sum_{d\in\mathfrak{D}} \varrho_d - \zeta_d \times K_{X_d} + \sum_{r\in\mathfrak{R}}\sum_{d\in\mathfrak{D}} \mu_{r,d} \times K_{L_{r,d}}}_{\text{Total DU Cost}} \\
& + \underbrace{\sum_{r\in\mathfrak{R}} \varpi_r \times K_{V_r}}_{\text{Total RU Cost}} + \underbrace{\sum_{c\in\mathfrak{C}}\sum_{d\in\mathfrak{D}} \nu_{d,c} \times K_{B_{d,c}}}_{\text{Total CU Cost}} + \underbrace{\sum_{s\in\mathfrak{S}} \xi_s \times K_{S_s}}_{\text{Path Cost}} \\
& \sum_{r\in\mathfrak{R}} \alpha_{u,r} \leq 1 \quad \forall\, u \in \mathfrak{U} \\
& \alpha_{u,r} \leq \delta_r \quad \forall\, u \in \mathfrak{U},\ \forall r \in \mathfrak{R} \\
& \sum_{u\in\mathfrak{U}}\sum_{r\in\mathfrak{R}} \alpha_{u,r} \geq UC \\
& \sum_{d\in\mathfrak{D}} \beta_{r,d} = \delta_r \quad \forall\, r \in \mathfrak{R} \\
& \beta_{r,d} \leq \zeta_d \quad \forall\, r \in \mathfrak{R},\ d \in \mathfrak{D} \\
& \sum_{c\in\mathfrak{C}} \gamma_{d,c} = \zeta_d \quad \forall\, d \in \mathfrak{D} \\
& \mu_{r,d} \leq MR \times \beta_{r,d} \quad \forall r \in \mathfrak{R},\ \forall d \in \mathfrak{D} \\
& \sum_{d\in\mathfrak{D}} \mu_{r,d} = \varpi_r \quad \forall r \in \mathfrak{R} \\
& \eta_{d,c} \leq MD \times \gamma_{d,c} \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \\
& \sum_{c\in\mathfrak{C}} \eta_{d,c} = \varrho_d \quad \forall d \in \mathfrak{D} \\
& \sum_{u\in\mathfrak{U}} \alpha_{u,r} \times UM \leq RC \times \varpi_r \quad \forall\, r \in \mathfrak{R} \\
& \sum_{r\in\mathfrak{R}} \mu_{r,d} \times RC \leq DC \times \eta_{d,c} \quad \forall\, d \in \mathfrak{D} \\
& \sum_{r\in\mathfrak{R}} \mu_{r,d} \leq DP \times \eta_{d,c} \quad \forall\, d \in \mathfrak{D} \\
& \sum_{d\in D} \mu_{r,d} \times RC \leq FC \times \varpi_r \quad \forall\, r \in \mathfrak{R} \\
& \eta_{d,c} \leq \nu_{d,c} \times CR \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \\
& \sum_{r\in\mathfrak{R}} \mu_{r,d} \times RC \leq FC \times \nu_{d,c} \quad \forall d \in \mathfrak{D},\ \forall c \in \mathfrak{C} \\
& \xi_s \geq \beta_{r,d} \quad \forall\, r \in \mathfrak{R},\ \forall d \in \mathfrak{D},\ \forall h \in P_{r,d} \\
& \xi_s \geq \gamma_{d,c} \quad \forall\, d \in \mathfrak{D},\ \forall c \in \mathfrak{C},\ \forall h \in P_{d,c}
\end{aligned}$$

12.3 Solvers

This section details the solvers employed in the framework, which aim to optimise the placement of RUs and DUs by minimising the total deployment costs while ensuring sufficient coverage to meet user demand. The cost minimisation process is subject to the constraints outlined in the previous section. The solvers used in this framework include CPLEX, Greedy Algorithm, Genetic Algorithm (GA) and Local Search (LS). Each solver offers distinct advantages and limitations, and their performance is evaluated across various scenarios to identify the most effective approach for solving the problem.

CPLEX

CPLEX solves the MILP from section 12.2. The problem is mathematically formulated with the objective function representing the total cost, and the constraints representing the requirements for coverage, demand points, devices and optical fibre paths. The model tracks the total cost, the number of RUs and DUs deployed, and the distance between them. CPLEX solves the problem providing an optimal solution. However, the time to find the optimal solution varies greatly depending on the scenario given to the solver. This solver is used to provide the optimal solution to the problem, which is then used as a benchmark to compare the results of the other solvers.

Greedy Algorithm

Greedy is used to provide a suboptimal solution to the problem quickly. The algorithm starts by selecting the RU location that covers the most demand points, it keeps adding RUs by order of the next largest coverage until the coverage requirement is met, then ignores any other users. It then does a post-process check to minimise the amount of underutilised RUs based on a specifically set RU threshold of 30%. It selects the DUs based on the shortest distance to the RUs, then calculates the shortest path between the DUs and the CU. Greedy is not guaranteed to find the optimal solution, but it is computationally less expensive than CPLEX. The solution provided by Greedy is used as a starting point for GA and LS.

Genetic Algorithm

GA is a population-based optimisation technique inspired by the process of natural selection. It evolves a population of candidate solutions (chromosomes) over several generations, striving to find an optimal or near-optimal solution. Initially, the population is generated using Greedy, which provides a reasonable starting point. The GA then refines this population through a combination of crossover and mutation operations. These operations ensure the evolution of the population, encouraging the exploration of the solution space and convergence toward better solutions.

Within GA, a dynamic mutation rate is set to avoid stagnation, the mutation rate is increased if the best solution has not changed for 5 generations. It also preserves the best individuals into an elite pool to ensure the best solution is not lost.

Mutation Operations

In GA, four distinct mutation operations are employed to introduce diversity and explore different areas of the solution space. Each mutation type is chosen randomly, serving a specific purpose in maintaining or improving solution quality:

- `mutateRU:` This operation randomly activates or deactivates 1 to 5 RUs in the solution. Upon mutation, the paths between the affected RUs and their corresponding DUs are recalculated, and the states of the RUs are updated accordingly. If a mutation results in an invalid solution, the mutation is reverted to maintain solution feasibility.

- `mutateDU:` Similar to `mutateRU`, this mutation targets DUs, activating or deactivating 1 to 2 DUs at random. Changes are accompanied by updates to the network, including reassigning RUs to alternative active DUs when necessary. If a mutation violates capacity constraints or results in an infeasible configuration, it is reversed to preserve solution validity.
- `mutateUsers:` This mutation selects 1 to 10 users and randomly adds, removes, or reassigns them to different RUs. Each modification prioritises feasible allocations that meet coverage requirements. If a user cannot be successfully modified (e.g., due to exceeding RU capacity or no possible option for the user), it is reverted to its original state prior to the attempted change, ensuring the solution remains valid.
- `mutateUserCount:` This operation adjusts the total number of active users to align with coverage requirements. If coverage falls below the desired threshold, new users are activated and assigned to RUs. Conversely, if coverage exceeds the required level, users are deactivated. In cases where coverage is already balanced, users are randomly activated or deactivated to promote exploration of the solution space. All changes are validated to maintain feasibility.

These mutation operations are designed to balance solution exploration and feasibility, ensuring that the population remains diverse while adhering to problem constraints. Both the GE and LS algorithms use these functions, selecting a mutation type at random each time the mutation operator is called.

Crossover Operations

The crossover operation plays a central role in recombining solutions. The two parent solutions are selected randomly from the population. The crossover operator selects a random crossover point, in this case being the DU connection of that RU. The parents then swap their DU connections at this crossover point, generating two offspring solutions. The offspring solutions are evaluated, and the best solution is selected to replace the worst solution in the population. This process continues until the population is filled with new offspring solutions.

Local Search Algorithm

LS relies on a neighbourhood search to explore the solution space. It starts with the best solution generated by Greedy and iteratively explores neighbouring solutions by making small changes to the population via the mutate functions. The algorithm evaluates the fitness of each neighbouring solution and accepts it if it is better than the current solution. This process continues for a set number of iterations or until no better solution is found. LS is used to refine the solution generated by Greedy, providing a more optimal solution than Greedy alone.

12.4 Simulation for Evaluating the Model

This section describes the simulation setup used to evaluate the model's performance in minimising the total cost of network deployment. The simulation takes into account various cost components, data set structures, and scenarios to assess how effective each solver is in minimising the total cost of deployment.

Cost of Deployment

The primary objective of the model is to minimise the total cost of deploying a 6G network. The total deployment cost is the sum of three main components:

1. **Total cost of network layout**: The cost of trenching and laying fibre between RUs, DUs, and the CU.
2. **Total cost of RUs**: The cost of installing RUs, including labour and fibre connection points.

3. **Total cost of DUs**: The cost of installing DUs, including labour and fibre connection points, placement of the first DU incurs an installation cost, while additional DUs at the same location don't incur the installation cost.

Table 12.2 provides a breakdown of the individual network components and their associated costs used within this simulation framework. Each cost component influences the decisions on RU/DU placement and the paths chosen for fibre installation.

Table 12.2: Cost of network Components Normalised to the Cost of Fibre.

Item	Normalised Cost
Piece of fibre per metre	\$1
Trenching per metre	\$5
Fibre Connection Cost at DU	\$24
Fibre Connection Cost at CU	\$47
RU Cost and Install	\$421
DU Cost and Install (first DU at location)	\$632
DU Cost (additional DU at location)	\$600
Existing Fibre per metre	\$6

Data Set Structure

To evaluate the model, a custom dataset has been developed to represent a regional area in New South Wales (NSW), Australia. This dataset captures the unique characteristics of rural and remote network deployment challenges. It incorporates the CU located at the Telstra exchange office to leverage existing infrastructure. DUs are evenly distributed across the region to ensure optimal coverage and connectivity, while RUs are strategically placed at road intersections, where they can be mounted on streetlights.

- **Radio Units (RUs)**: There are 108 potential locations for RUs, distributed at key points within the region, such as roundabouts or intersections. Each RU has a specific coverage radius of 500m and the selection of RU locations is determined both on cost and meeting the demand and coverage requirements.
- **Distributed Units (DUs)**: There are 16 potential DU locations distributed evenly across the region. The placement of DUs is important for ensuring optimal coverage and minimising deployment costs.
- **Centralised Units (CUs)**: There is a single CU in the network, representing the core of the architecture where all data traffic ultimately converges. The CU's location is predefined and fixed.
- **Fibre Paths**: The model considers the shortest possible fibre paths between RUs to DUs and DUs to the CU. The paths are pre-calculated using NetworkX [3]. The total fibre length affects the overall deployment cost, particularly for trenching and fibre bundles, as detailed in Table 12.2.

The full region dataset is shown in Figure 12.2. In this representation, the black paths indicate assumed existing fibre paths within the region. The blue hexagons represent the locations of existing DUs, and the red circles mark the positions of existing RUs. The overlaid grid represents user density, sourced from Australian Bureau of Statistics [1] population data, with green indicating low density, orange for medium density, and red for high density. The road network was collected via custom scripts using the OSMnx library [2] to extract road segments. With the maps displayed using the Folium library [7] to display OpenStreetMap [6] data.

Simulation Environment

The simulations were conducted on the Deakin High Performance Computing (HPC) cluster, leveraging the computational resources to run the CPLEX solver [4] and the custom Python implementations of Greedy, GA, and LS. The heuristics use parallel processing to evaluate multiple solutions concurrently, enhancing the efficiency of the optimisation process. The simulation environment is designed to evaluate the performance of each solver across different scenarios, assessing their ability to minimise deployment costs and meet coverage requirements effectively.

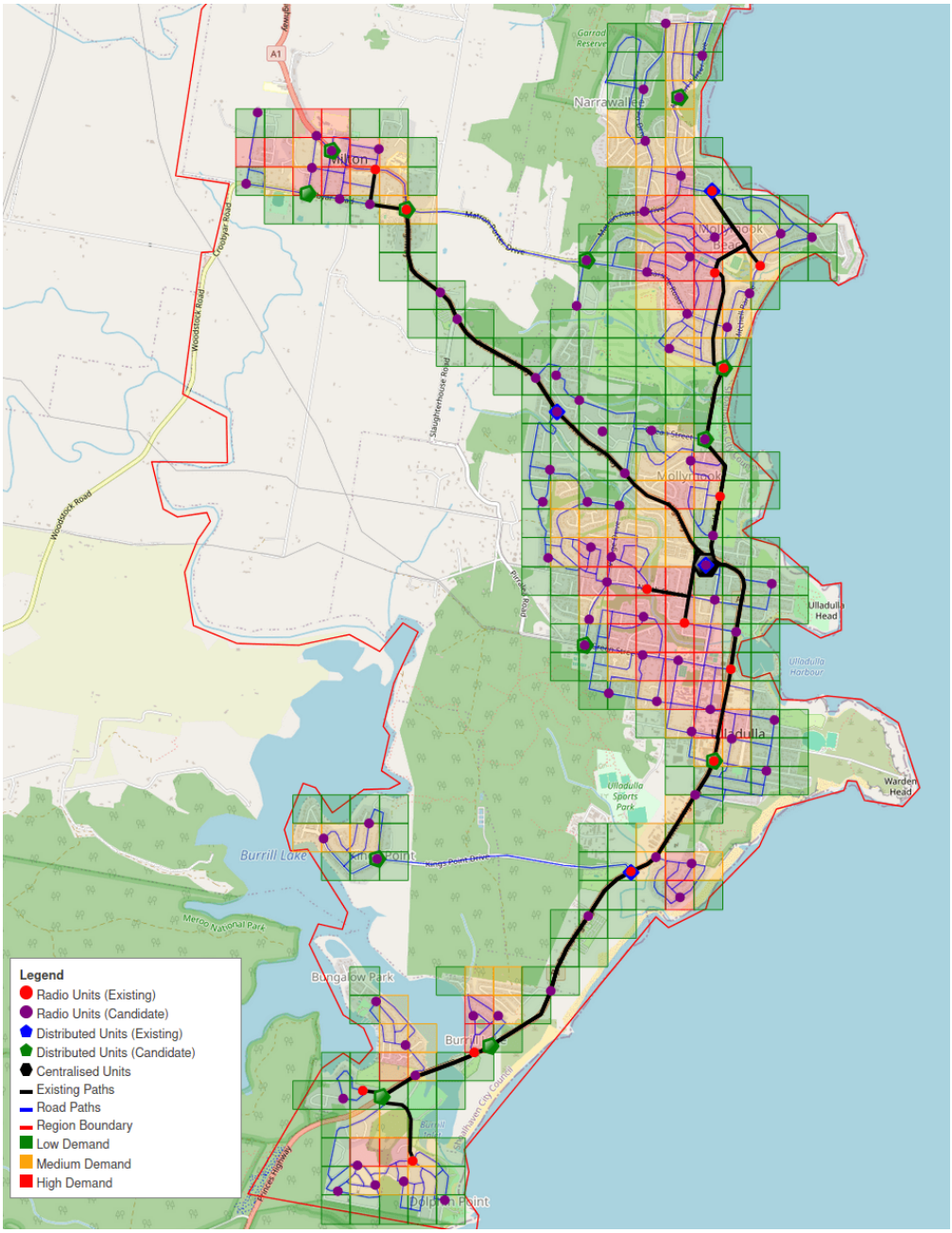


Figure 12.2: Overview map of the dataset region, including the device locations, population distribution and road network. Created using `Folium` in Python.

12.5 Results

The model was evaluated at coverage levels of 80% and 90%, while considering varying user bandwidths of 1000mb, 2000mb and 3000mb. The results provide insights into the performance of each solver in minimising the total cost of network deployment while meeting coverage requirements.

Overview of Results

The results are presented in two figures, one illustrating the normalised costs achieved by each solver across different coverage levels and user bandwidths (Figure 12.3), and another comparing solution time for all solvers (Figure 12.4).

From these results, CPLEX consistently provides the lowest cost across all scenarios, achieving the optimal solution. In contrast, Greedy consistently delivers a suboptimal solution, with its cost range spanning between 3 to 4 times that of CPLEX. GA and LS methods offer solutions within the ranges of 1 to 2 times the cost of CPLEX. GA performs better when coverage requirements are higher, while LS excels under lower coverage levels. It can also be noted from the results in Figure 12.3 that LS provides a consistent gradient, similar to how CPLEX does as the coverage/bandwidth increases, while GA and Greedy have differing results.

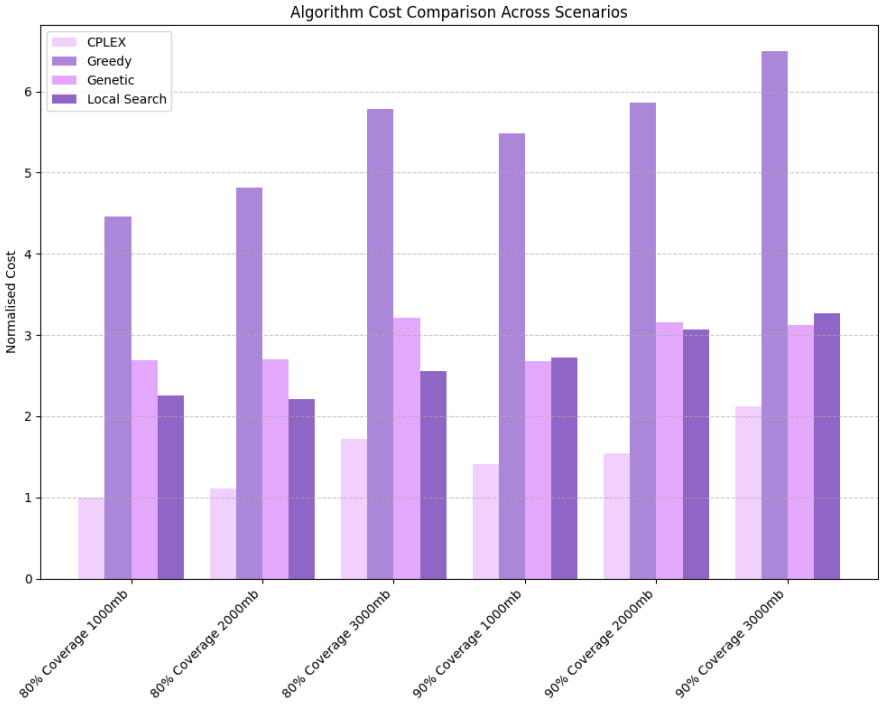


Figure 12.3: Results of the simulation showing the normalised cost achieved by each solver at different coverage levels and user bandwidths. Created using `Matplotlib` in Python.

While the costs achieved by Greedy, GA, and LS are higher than those obtained by CPLEX, they offer viable alternatives in scenarios where computational resources are limited, or the optimal solution is not required. Additionally, for cases where solution time is critical, as demonstrated in Figure 12.4, the solution times of each model are compared. CPLEX requires an average of 4000

to 5000 seconds to compute a solution, whereas Greedy, GA, and LS significantly outperform it in speed, averaging between just 10 and 120 seconds. These results highlight that Greedy, GA, and LS are effective methods for producing near-optimal solutions in a fraction of the time required by CPLEX, making them practical for time-sensitive or resource-constrained applications.

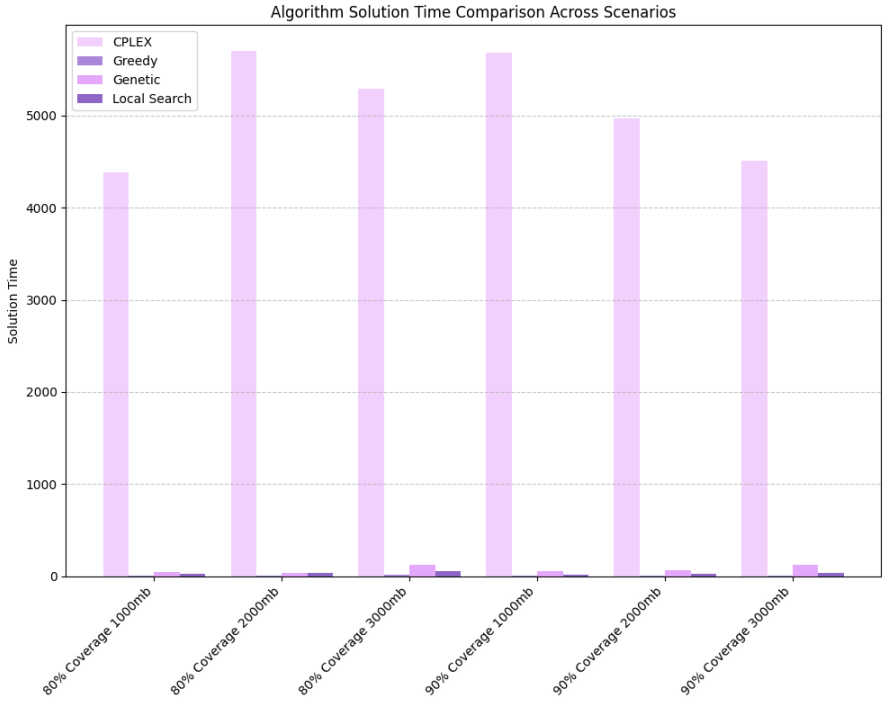


Figure 12.4: Results of the simulation showing the solution time achieved by each solver at different coverage levels and user bandwidths. Created using `Matplotlib` in Python.

Mapped Results

The results of the simulation were mapped using Folium in Python to visualise the deployment of RUs and DUs across the region. The maps provide a geographic representation of the network layout, highlighting the placement of devices and the paths of fibre connections. The maps offer a visual comparison of the deployment strategies employed by each solver, illustrating the differences in RU and DU locations, fibre paths, and coverage areas. Figure 12.5 shows the results of CPLEX, while Figures 12.6, 12.7, and 12.8 display the results of Greedy, GA, and LS, respectively.

Each solver produces a unique network structure, with variations in RU and DU placement as well as fibre path selection. While CPLEX delivers the most optimal solution, the heuristic approaches provide competitive alternatives that effectively meet coverage requirements while striving to minimise deployment costs. These differences are evident in the mapped results, where each method prioritises different trade-offs between cost, coverage, and computational efficiency.

The simulation results highlight the strengths and trade-offs of each solver in optimising network deployment. CPLEX, as an exact solver, ensures the lowest-cost solution while meeting all constraints. Meanwhile, the heuristic methods offer near-optimal solutions with significantly reduced computational complexity and solution time, making them practical alternatives for large-scale deployments. The visualisations provide a geographic perspective on the network

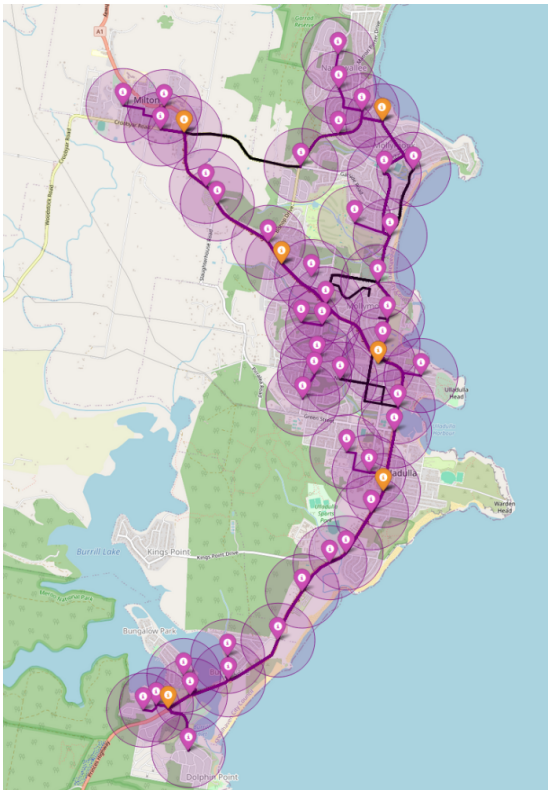


Figure 12.5: The results of CPLEX at 90% coverage and 1000mb bandwidth. Created using `Folium` in Python.

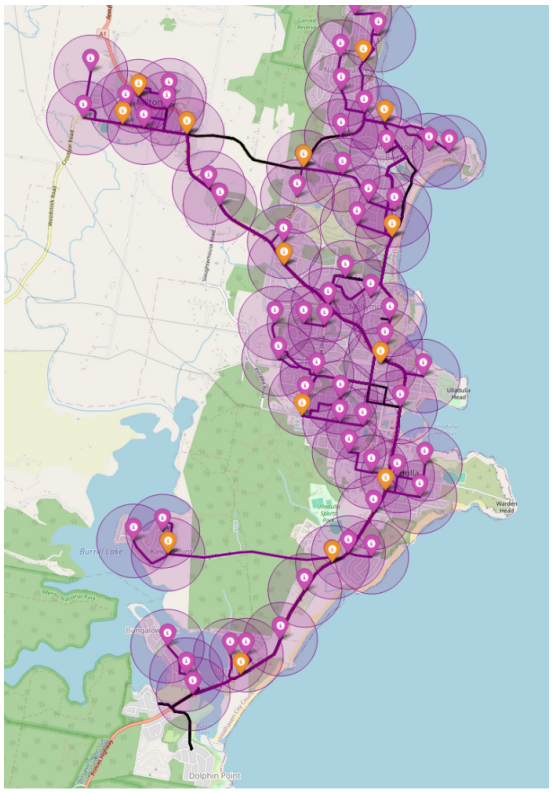


Figure 12.6: The results of Greedy at 90% coverage and 1000mb bandwidth. Created using `Folium` in Python.

layout, illustrating the placement of RUs, DUs, and fibre paths. These insights are valuable for understanding how different deployment strategies impact overall network efficiency and cost-effectiveness in a 6G scenario.

12.6 Conclusion

This report evaluated the performance of both MILP solvers, such as CPLEX, and heuristic methods for optimising network deployment in rural areas using the proposed novel MILP framework. The results demonstrate the effectiveness of this framework in modeling and solving large-scale network deployment problems, providing valuable insights into the trade-offs between exact and heuristic approaches.

CPLEX, as an exact solver, guarantees an optimal solution by exhaustively searching the solution space. Using the proposed MILP framework, CPLEX successfully finds the exact solution to the MILP, ensuring that the most cost-effective deployment strategy was identified. This validates the robustness of the framework in handling complex network optimisation problems. However, achieving full optimality required several hours of computation on an HPC cluster, highlighting the trade-off between solution accuracy and computational effort. While CPLEX is well-suited for strategic, cost-sensitive deployments where time constraints are less critical, its high computational cost makes it impractical for real-time decision-making or dynamic network planning.

In contrast, heuristic approaches such as Greedy, GA, and LS offer viable alternatives when rapid solutions are required. While they do not guarantee global optimality, the proposed

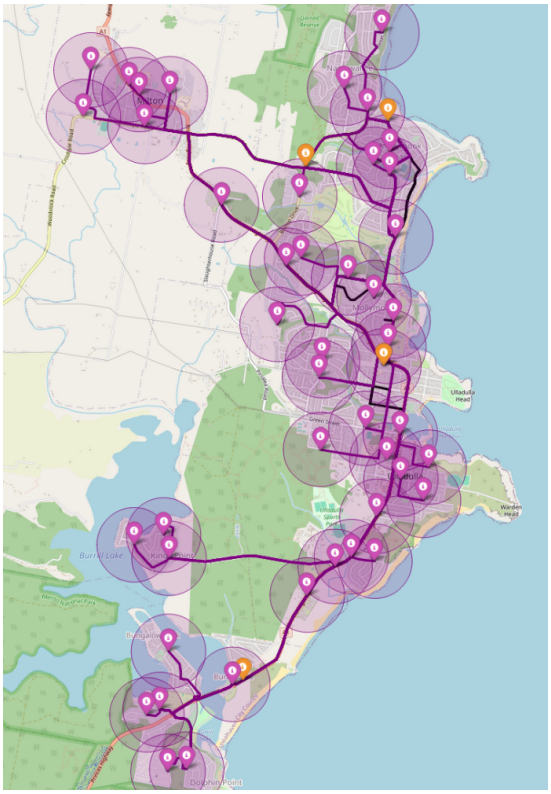


Figure 12.7: The results of GA at 90% coverage and 1000mb bandwidth. Created using `Folium` in Python.

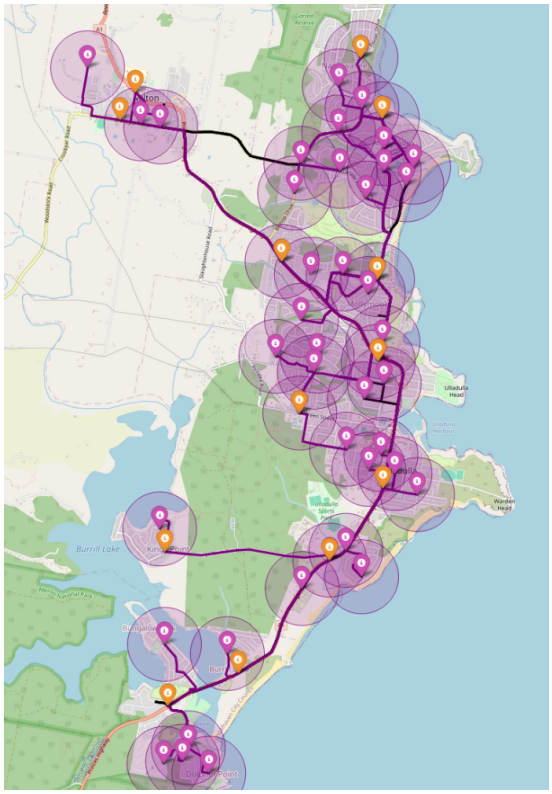


Figure 12.8: The results of LS at 90% coverage and 1000mb bandwidth. Created using `Folium` in Python.

framework enabled these methods to produce near-optimal solutions efficiently. Among them, LS demonstrated the best performance, frequently outperforming Greedy and GA by refining solutions iteratively. However, heuristic methods tend to be limited in their ability to fully explore the solution space, often converging on locally optimal solutions based on the initial conditions set by Greedy.

The results demonstrate that the proposed MILP framework effectively balances solution quality and computational efficiency, making it adaptable for both exact and heuristic solvers. Heuristics remain the preferred choice in time-sensitive scenarios or when an approximate solution is sufficient for deployment. However, when computational resources allow and cost minimization is the primary objective, CPLEX, within the proposed framework, remains the superior choice, achieving significant cost savings over heuristic alternatives.

Context

This article was adapted from a report submitted during SIT316.

About the Author

I'm currently pursuing a Bachelor's degree in Cyber Security with an Honours specialisation in telecommunications and mathematics. In 2025, I will continue into a PhD in the same field. While I initially enrolled in cybersecurity out of curiosity for the field, my research interests shifted towards telecommunications after completing the Associate Dean's Research Summer Scholarship in 2023/24.
My current research focuses on extending my existing model in this report, into a graph-based framework, incorporating more complex constraints and developing advanced heuristic algorithms to enhance its scalability and efficiency. This work aims to optimise network deployment strategies, contributing to the broader field of telecommunications and operations research.

Acknowledgements

I would like to express my gratitude to Julien Ugon for his invaluable support and encouragement throughout SIT316, as well as for being the unit chair in SIT192, where I first discovered my passion for mathematics. His guidance has been instrumental in shaping my academic journey and inspiring my love for problem-solving.

I also extend my thanks to Chathu Ranaweera for her support and expert guidance in the telecommunications aspects of this project. Her insights and mentorship have been crucial in navigating this challenging and rewarding area of research.

Together, their encouragement, mentorship and unwavering belief in my potential has been instrumental in my decision to pursue honours and a PhD with them. I'm deeply grateful for their role in shaping my future.

References

[1] Australian Bureau of Statistics. ABS australian population grid 2022. Dataset, 2022.

[2] Geoff Boeing. OSMnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, September 2017.

[3] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[4] IBM. IBM ILOG CPLEX optimization studio, 2023. https://www.ibm.com/products/ilog-cplex-optimization-studio. Version 22.1.

[5] O-RAN. About O-RAN Alliance. Technical report, O-RAN Alliance, 2018.

[6] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org , 2017. https://www.openstreetmap.org .

[7] Rob Story. Folium (version 0.11.0), 2020. https://python-visualization.github.io/folium/.

13

The mathematics behind models: a Bayesian framework

Trill White

Abstract

One of my first science trips at school was run at a native bird sanctuary in Geelong somewhere, learning about natural habitats. Walking around the sanctuary, measuring and counting things, writing down natural phenomenon. The measurements between my partner and I so wildly different, with different ideas about what it all meant. My point? As you grow up, you realise not all that much changes.

My clipboard and real world data have a few things in common. For instance, real-world data is rarely pristine. It's often "messy", which means that we cannot be guaranteed that the measurements we took (or have been given) are free from imprecision, or all equally important results. This makes conclusions harder to draw, especially when the data we have gets very, very big.

A large problem remains in how we can best analyse big data. The growing need to derive accurate insights from these massive datasets presents a challenge that is computationally intensive. And, without a framework to develop how we could quantify our uncertainties or understandings, the resulting models could be inaccurate.

This chapter devotes time to developing a mathematical framework for creating an accurate model informed by a belief that we might have about our data. We show that by quantifying our informed belief about our random and not-so-accurate dataset, we can still by-large reproduce the original function, in a time effective and consistent manner. We discuss the success of this framework in an example problem, and comment on future work.

13.1 Introduction

To begin this chapter, we shall investigate a simple example of fitting a function to a set of random points, and then introduce our problem statement. The reader is invited to pay particular attention to three key but simple ideas. Firstly, that we are approximating a function according to some random data, (whose real function we know not). Secondly, we have a belief about what sort of function this should be.

And finally, thirdly: Our problem statement, first introduced below, will undergo change as we develop our knowledge and understanding to a point where we can approximate polynomials using Bayesian regression. As we update our knowledge, we add this understanding to our problem statement, to reflect our growth throughout this chapter.

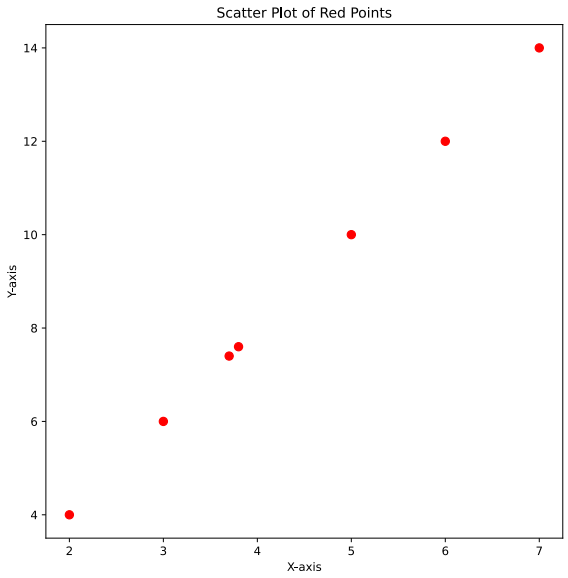


Figure 13.1: Some not-so random points.

Problem Statement 13.1: Simple Problem Outline

- We have a set of random data, and we wish to find a model that will accurately explain the data.
- We don't know what the generating function for our dataset was, but we are assuming it is a polynomial of the below form, where degree m is unknown:

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

For now, let us work on modelling a linear relationship.

Starting with a simple example, finding the best model - or in this case, the "best line of fit" is quite trivial. We can intuitively tell from looking at the graph below (Figure 13.1) that a nice way to limit the error (Euclidean difference our line could have, from each point) will be to draw a straight line through our data:

Such a line might look something like Figure 13.2.

We made this decision using:

- The points we have,
- By minimising the distance of each of the points from the line,
- Our intuition about a possible solution

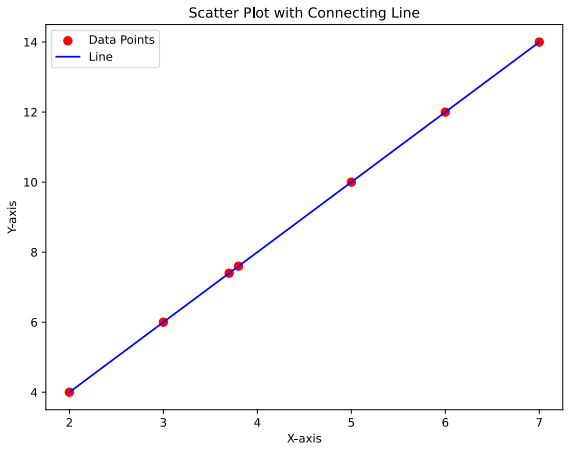


Figure 13.2: Nice simple solution

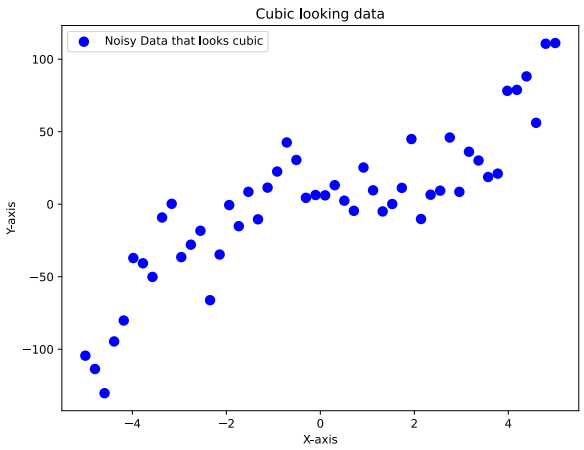


Figure 13.3: Some random points (but we think it looks cubic)

All well and good. However, in Figure 13.3, we can see that a good solution for our dataset is no longer going to be linear (this is because, if we were to fit a linear function, it would result in more error, than, say, a cubic solution).

We have two solutions for possible equations to explain the data in Figure 13.3 and these are shown in Figure 13.4.

One solution tries to minimise the error by going through every single point in the graph. But, when we look at Figure 13.4, we can see that this is not a good solution. Why? When we take measurements in real life, we know that some of them are not going to be accurate measurements - but, it is important to capture the overall flow, the overall message our data tells. This allows us to extrapolate better for our future predictions - over time, for example, or as temperatures change.

Even though the "red" model goes through all the points, which minimises error from the points, it does not take into account the fact that these points themselves have error. Because the model closely fits to these points, it is overfitting, and not capturing the real story behind the data.

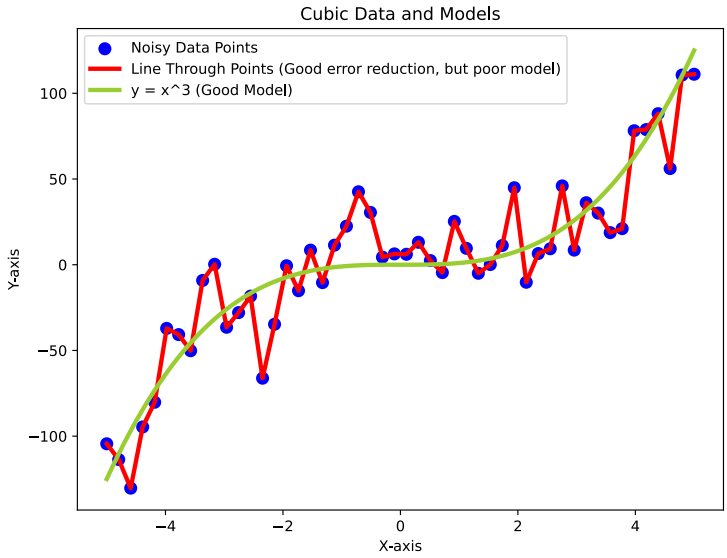


Figure 13.4: Two approximations, one is obviously prettier

In the "green" model, we still minimise the error (the average vertical distance between our data points and the approximate data points offered by the model) but we are no longer creating a solution that looks quite erratic - we can see it still fits the data ok, and provides a smoother and well-behaved solution.

Note 13.1: Where Intuition can Guide Decisions

What we saw in the models we provided as solutions is differences in accuracy, versus model smoothness and stability. What we observed, is that as we move to datasets with more data and less certainty in accuracy, after a point (like in Figure 13.4), the best solution is no longer running a line between every single point.
Earlier, we had simple solutions that seemed to work quite well. And now, we have to rethink how we define success and how a "best solution" can be found. Recall we defined our line of best fit as one that reduced the error, and was as close to our points as possible.

Minimising the errors is a concept is known as **regression**.

13.2 Regression

Regression is a mathematical approach to finding the relationship between a single dependent variable and its' one or more independent variables.

Different methods of regression calculate this relationship using different techniques, including by the way it calculates the error. Note, error here is defined as the Euclidean difference between a data point in the dataset, and the predicted value per the model). One way of defining the error is by using Ordinary Least Squares Technique.

Definition 13.1: Ordinary Least Sum of Squares

Suppose we have some data where y is dependent on x. We're not sure what the relationship is, but lets suppose for now it is linear. We know our measurements have some error in our readings, but we would like to model $\hat{y}$, our approximate function to this noise as:

$$\hat{y} = b \times x + a + \epsilon$$

Where:

- $\hat{y}$ is our approximate function of $y(x)$
- b is our unknown coefficient
- a is our y-intercept
- ϵ is our data error

We find the regression coefficients (a and b) so that the sum of the squared residuals (RSS) is minimised. This approach is called ordinary least squares (OLS).

$$SSR = \sum (y_i - \hat{y}_i)^2 \tag{13.1}$$

Example 13.1: Linear regression

One way of finding a line of best fit is using **simple linear regression** which is where a line is fitted according to the **ordinary least squares**.

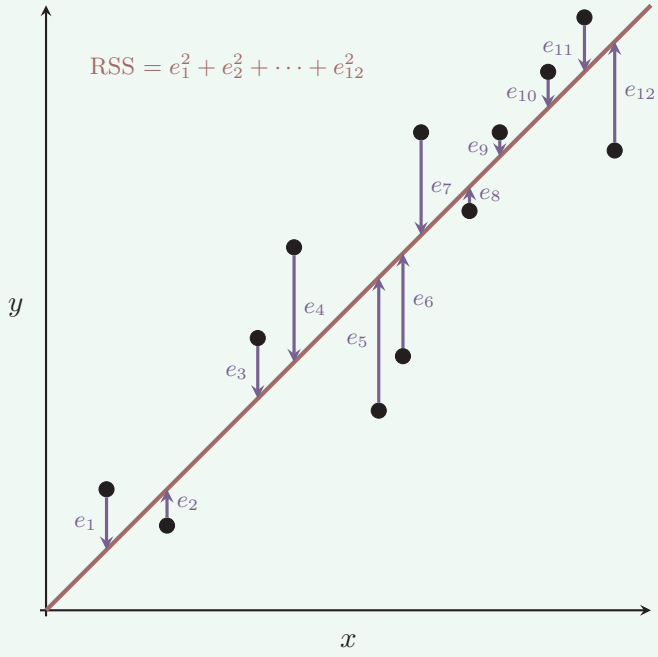


Figure 13.5: Diagram for comprehension of residual sum of squares. Created using Tikz [11].

When we create a model, we can use OLS to assess how accurate it really is.

Problem Statement 13.2: Update - Defining our model accuracy

- We have a set of random data, and we wish to find a model that will accurately explain the data.
- We don't know what the generating function for our dataset was, but we are assuming it is a polynomial of the below form, where degree m is unknown:

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

- We know that we will be assessing the accuracy of our model using OLS by comparing against the original generating function

Now we have a clearer idea of the sort of model that we are creating.

Another area we can clarify about our model now, is how complex we want to make it. This is where regularization comes in.

13.3 Regularization

One problem we face, as stated earlier, is that we are unsure about how complex our model really is. Because we cannot know how complex our model should be, based off the random data we have, there are a range of possible different models we could create.

Some things, like the position in time of a ball falling through the sky, can be modelled by quadratic equations. Others, like the volume of water in a tank when someone fills it up, can be modelled by cubic solutions.

The real issue here is deciding how complex we would like to make our model, based on the data that we have. If we make it more complicated than it needs to be, the time to compute results will be longer, and we could over-fit to our dataset. But, if it is not complex enough, then it could under-fit the data set.

Definition 13.2: Overfitting

Over-fitting is where a mathematical model too closely fits the dataset that it has been provided to extrapolate for. That is, just like the erratic model in figure 13.4, it might wildly oscillate in between all the points in the dataset, to reduce error - nut then as a predictive function provide wildly different answers than we would expect

Definition 13.3: Underfitting

Under-fitting is where a mathematical model does not capture enough of the trend of the dataset, meaning that it, too, is not a useful tool for extrapolation. Under-fitted models tend to not perform well because they are too simple.

Regularization is a process where a simpler solution is favoured over a more complicated one. It is a mathematical way of asserting a preference for solutions to be smoother and easier to computer, than a more complicated solution.

It is a technique that is applied to ill-posed problems, problems that have no single solution, or to prevent overfitting. This is a core concept in mathematics that underpins many fundamental technologies such as machine learning and neural networks [8].

But, accuracy versus simplicity are competing goals. A combination of the two is beneficial when we are trying to make a balance between good fit and simplicity. However, it is important that the solutions are not so simple that we under-fit.

Each type of regularization creates a penalty term to limit the size of the estimated coefficients. You might see the term "shrinkage estimator", in other texts, and that is a because of its general effect in modelling. The penalty it creates is a bias, or limit on the size of the generated coefficients.

Here we present two common types of regularization, known as LASSO, and Tikhonov.

LASSO Regularization

LASSO (Least Absolute Shrinkage and Selection Operator), or L1, regularization is a regression method that helps minimise the maximum values in our model, to make it more smooth. It does this by minimising the following equation:

Definition 13.4: LASSO Regularization

LASSO regularization minimises RSS + $\lambda \times ||C(a_S)||$:

1. Where the RSS = sum of the squared residuals = $\sum(y_i - \hat{y}_i)^2$
2. Here, λ is our penalty function. It is what drives the balance between model fit and simplicity. A larger value, and our function creates a larger penalty on the coefficients. A small value, and there is less penalty on the coefficients.
3. $||C(a_S)||$ is the magnitude of all our models' coefficients
4. L_1 is $\lambda \times ||C(a_S)||$

How does LASSO work? LASSO creates simpler models by forcing the coefficients of the model to be reduced in a consistent manner across the entire function. This is known as a dimensionality reduction technique, because it forces our model to become simpler [2].

Let us demonstrate the effect that the size of our coefficients has in LASSO. LASSO, recall, is performing a minimisation based on the magnitude of our coefficients. For coefficients a_i of $f(x) = \sum_{i=0}^{m} a_i x^i$, suppose $\lambda = 1$; then we are only looking at the sum of the coefficients. The effect of the **value** of our coefficients (as opposed to the **order**) is shown below;

Example values for a_1	Example impact of coeffs under LASSO
10,000	10,000
8	8
4	4
2	2
1	1
0.5	0.5
0.1	0.1
0.01	0.01

Table 13.1: Impact of coefficients under LASSO

Each coefficient has been minimised in the same way, that is, they have been shrunk by the same factor. If λ had placed a higher penalty on the coefficients (i.e so that they were halved) then this would have affected each coefficient equally. This means that we can force the size for each coefficient to become much smaller. We also see that this impact is linear - in that the large coefficients are impacted the same as the smaller coefficients.

However, our primary objective is model simplicity. Because LASSO minimises the coefficients in the same way, we would like to minimise the effect of the larger coefficients relatively more, using some other method.

This is where we can use Tikhonov regularization.

Tikhonov regularization or Ridge regression

Note 13.2: Tikhonov regression or regularization?

Helpfully enough, Tikhonov regularization is also known as ridge regression, but that is not to say regression is the same as regularization. Regression is about ensuring that our line fits the data, whereas regularization is about ensuring that our solution is simple and nice. From now on, I'm just going to be sticking to the name "Tikhonov Regularization", or Tikhonov for short.

Definition 13.5: Tikhonov Regularization

Tikhonov regularization [9, 6] minimises the RSS $+ \lambda \times ||C(a_S)||^2$. Here, $\lambda \times ||C(a_S)||^2$ is our Tikhonov Regularization penalty, also known as the **L2 Norm**. Tikhonov regularization prefers a solution by minimising $||\hat{y} - f(x)||^2 + \lambda ||C(a_S)||^2$.

1. Again, RSS = sum of the squared residuals = $\sum (y_i - \hat{y_i})^2$
2. And λ is still our penalty function. It has the same effect as before, if this constant is larger, then it creates a larger penalty on the coefficients. If it is smaller there is less of a penalty on the coefficients.
3. $||C(a_S)||$ is the magnitude of all our models' coefficients
4. And the L_2 Norm is $\lambda \times ||C(a_S)||^2$

So, in Tikhonov, we are minimising according to the magnitude of the **square** of the coefficient. This next table shows the impact that our coefficients a_i of $f(x) = \sum_{i=0}^{m} a_i x^i$ have on this minimisation.

Again, we are talking about the **value** of our coefficients, and not the **order**.

So, if we consider the coefficient 8, its impact is going to be squared, since we care about the square of the coefficient. This means larger coefficients have much more impact, and therefore the shrinkage on these terms is much larger (four times larger in the case of 8) than the shrinkage on the coefficient of 4. And, as we approach zero, the change in the coefficient become more and more irrelevant as the square tends to zero.

Because Tikhonov has such a bigger impact on large coefficients than smaller coefficients, this is useful for our model, where we would like to minimize larger terms.

So what is the real difference between LASSO and Tikhonov? LASSO minimises $||C(a_S)||$, and Tikhonov minimises $||C(a_S)||^2$. A key difference is that LASSO doesn't has such a "harsh"

Example values for a_1	Impact of coefficient under Tikhonov
8	64
4	16
2	4
1	1
0.5	0.25
0.1	0.01
0.01	0.0001

Table 13.2: Impact of coefficients under Tikhonov

reduction fact that Tikhonov does, for very large coefficients but is harsher for small coefficients. This is because LASSO does not have a quadratic effect, but a linear one.

Using both the $L1$ and $L2$ norm are both useful when it comes to finding a good solution, and they are often used together [5, 2].

To show a side by side comparison of the different effects they have on the coefficients, see the table below:

Coefficient	Coefficient impact in the LASSO, L_1 $\|\|C(a_S)\|\|$	Tikhonov, $L_2, \|\|C(a_S)\|\|^2$
8	8	64
4	4	16
2	2	4
1	1	1
0.5	0.5	0.25
0.1	0.1	0.01
0.01	0.01	0.0001

Table 13.3: Comparison of the impact of the coefficients of LASSO vs Tikhonov for different values

When we use both methods, we minimise the size of the largest coefficients more than the smaller ones, but also shrink the smaller ones as well. Since they minimise different measurements, they can both be employed to create a smoother solution.

Problem Statement 13.3: Update: Regularization

- We have a set of random data, and we wish to find a model that will accurately explain the data.
- We don't know what the generating function for our dataset was, but we are assuming it is a polynomial of the below form, where degree m is unknown:

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

- We know that we will be assessing the accuracy of our model using OLS by comparing against the original generating function
- **We will be using LASSO (RSS $+ \lambda \times ||C(a_S)||$) and Tikhonov regularization (RSS $+ \lambda \times ||C(a_S)||^2$) to find the smoothest solution for model.**

A vague area in our problem statement is our idea about the noise in our measurements. We know that our measurements have error, but we are uncertain as to the amount of error that our measurements contain.

This is an issue because if we are not able to quantify our uncertainty of the error, we can't estimate how good our eventual solutions are.

To be able to do this, therefore, we need to find a way to quantify the uncertainty that we have in our measurements - or, in other-words, the range of values we expect the error to take.

We can do this using a **probability density functions**.

13.4 Probability Distribution Functions (pdfs)

Generally, if an event is constant, discrete and not continuous, the probability of the event will also be constant and uniform.

In a finite sample space, for all total possible events Ω, with event A and independence, and total n ways that event A could occur, the probability that event A occurs, P(event A) $= \frac{n(A)}{n(\Omega)}$.

However, what if there are infinitely many events possible? i.e $n(\Omega) \to \infty$, then we are not able to calculate the possibility of a single event. Then we need to find a way of describing the probability of an event occurring by using a **probability density function**.

Definition 13.6: probability density functions

A probability density functions, or pdf, is a function that is used to give the probability of an event occurring. There are a few key features that a probability function must contain:

1. The sum of the probabilities of all events inside the domain of the pdf is 1.
2. The pdf function must be positive

Below is an example that illustrates how a pdf can be used to find the probability of a single event or outcome in a problem with independent events and continuous measurements.

In this example, we highlight that the density function is not constant and does indeed depend on the particular outcome. We further highlight that if we take an event space that has probability equivalent to a constant, then the pdf will not change, but if we take a probability that is **not** constant, the probability is likely not constant either.

Example 13.2: Concentric Rings on a Circle

Suppose we have a circle board which we split into two equal sized concentric rings (we halve the radius, see Figure 13.6). Suppose we want to calculate the chances of our dart hitting a particular part of the board, ring 1 or ring 2.

We can calculate the area of each concentric ring and they are as follows:

$$A_1 = \pi(1)^2 = \pi \tag{13.2}$$

$$A_2 = \pi(2)^2 - A_1 = 3\pi \tag{13.3}$$

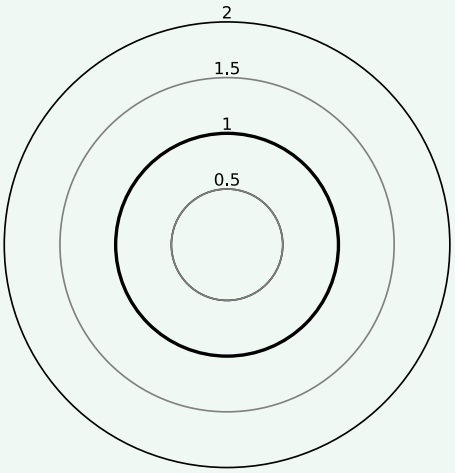


Figure 13.6: Circle Dart Board

The dart is not weighted to one side, which means that it has **no bias**.

If a_1 and a_2 are the associated probabilities that a dart lands in the area of the board, we can calculate:

$$P(a_1) = \frac{\pi}{4\pi} = 1/4 \tag{13.4}$$

$$P(a_2) = \frac{6\pi}{4\pi} = 3/4 \tag{13.5}$$

We shall continue this experiment and calculate the area of concentric circles if we doubled the radius in A_1 and A_2. Therefore we get the following areas, A_1, A_2, A_3 and A_4, which using basic subtraction and the equation for the area of a circle, can calculate to be

$$A_1 = \pi\left(\frac{1}{2}\right)^2 = \frac{\pi}{4}, \quad A_2 = \pi\left(\frac{2}{2}\right)^2 - A_1 = \frac{3\pi}{4},$$

$$A_3 = \pi\left(\frac{3}{2}\right)^2 - A_2 - A_1 = \frac{5\pi}{4}, \quad A_4 = \pi\left(\frac{4}{2}\right)^2 - A_3 - A_2 - A_1 = \frac{7\pi}{4}$$

Then, the probability of a dart falling in each area is listed as below:

- $P(a_1) = \frac{A_1}{total} = \frac{\frac{\pi}{4}}{4\pi} = \frac{1}{16}$
- $P(a_2) = \frac{A_2}{total} = \frac{\frac{3\pi}{4}}{4\pi} = \frac{3}{16}$
- $P(a_3) = \frac{A_3}{total} = \frac{\frac{5\pi}{4}}{4\pi} = \frac{5}{16}$
- $P(a_4) = \frac{A_4}{total} = \frac{\frac{7\pi}{4}}{4\pi} = \frac{7}{16}$

Graphing we can see that this is modelled as below:

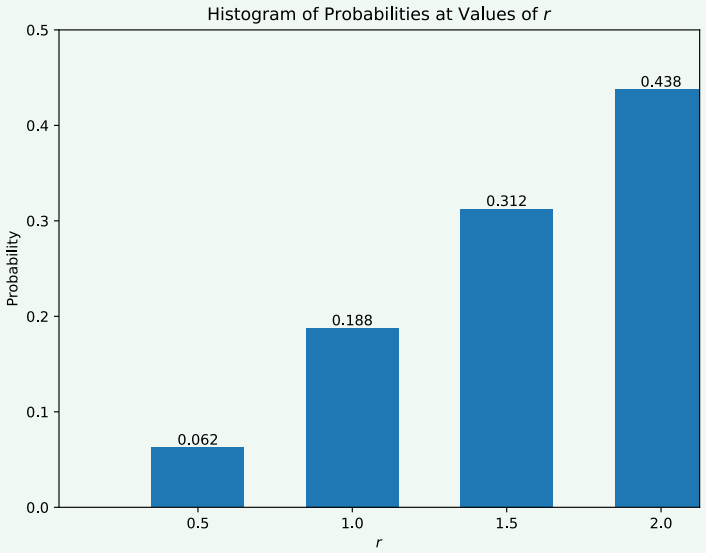


Figure 13.7: Histogram

From this, we can see that not every function will have a constant density function (that is, not every probability for different events is as likely as each other).

We might like to see how, if we have a probability for a range of values, that we can see that maximising the probability density function will allow us to maximise our chances of finding a probable solution.

Note 13.3: Maximising Probabilities and Log Laws

If $P(y_i)$ represents the probability (or probability density, if it's a continuous distribution) of observing y_i, then the probability of observing all of them together is (given independent events) is :

$$P(y_1, y_2, ..., y_n) = P(y_1) \times P(y_2) \times ... \times P(y_n)$$

This can be written more concisely using product notation:

$$P(y_1, y_2, ..., y_n) = \prod_{i=1}^{n} P(y_i) \tag{13.6}$$

To simplify finding the maximum probability of all events, i.e the pdf, we can take the log of both sides:

$$\ln\left(P(y_1, y_2, ..., y_n)\right) = \ln\left(\prod_{i=1}^{n} P(y_i)\right)$$

$$= \ln\left(P(y_1) \times P(y_2) \times ... \times P(y_n)\right)$$

Recall logarithmic laws, where $ln(a \times b) = ln(a) + ln(b)$:

$$ln(pdf) = \ln(P(y_1)) + \ln(P(y_2)) + ... + \ln(P(y_n)) \tag{13.7}$$

$$ln(pdf) = \sum_{i=1}^{n} (\ln(P(y_i))) \tag{13.8}$$

Our aim is to identify the model that is most likely to explain our data.
Using this formula, on an appropriate pdf, will give us our likelihood. We want to find the most likely model by maximising this function (1.5). Because maximising a sum if often easier than maximising a product, we maximise equation 1.7 instead of 1.5, which will give us the same model.

Normally distributed events and pdfs

PDFs can model the likelihood of many things. Depending on how events are distributed, there are different PDFs to model these events.

One common formula for modelling event likelihood is the normal distribution. Many natural values can be approximated with the normal distribution, such as the range of human height, the weight of a newborn, and shoe-size. The equation for the probability density function for an event that is normally distributed is as follows:

$$P(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

To explain this idea further, let us examine how one might describe a probability distribution for the error in our measurements, with figure 13.8. Recall that the normal distribution has two parameters, the mean μ and the standard deviation, often written as σ.

Smaller values of σ gives a smaller range.

If the red function modelled the pdf for the range of error, this would indicate that we are more confident that our errors fall within a smaller range than the blue model. If we used the blue model however, we are indicating our confidence is low that our range of errors is small.

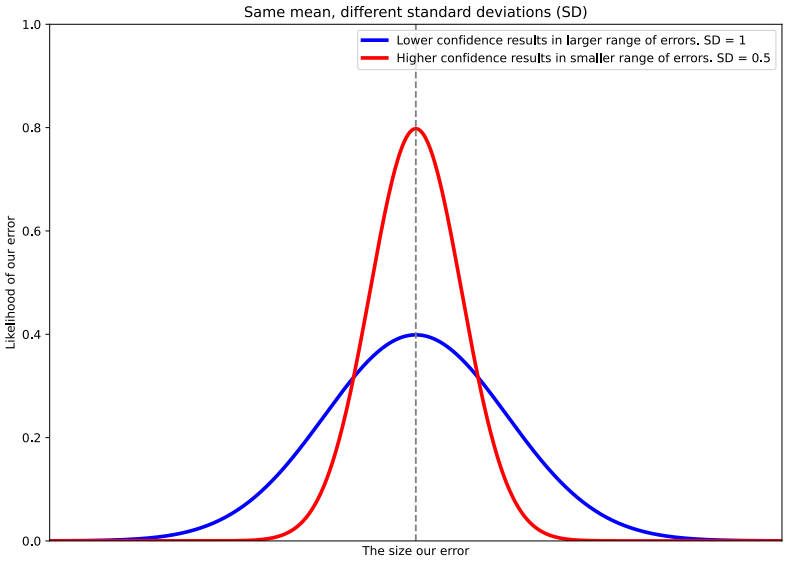


Figure 13.8: Standard Deviations

Since we have a way of quantifying the range of errors we expect to see in our dataset, we can now update our problem statement to include this.

Problem Statement 13.4: Maximising the likelihood of finding our line

- We have a set of random data, and we wish to find a model that will accurately explain this data.
- We don't know what the generating function for our dataset was, but we are assuming it is a polynomial of the below form, where degree m is unknown:

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

- We know that the data we use to make our model will have some errors, but that we will be assessing the accuracy of our model using OLS
- We will be using LASSO (RSS + $\lambda \times ||C(a_S)||$) and Tikhonov regularization (RSS + $\lambda \times ||C(a_S)||^2$) to find the smoothest solution for model.
- We now know that the range that our errors falls in can be modelled using a **probability density function**
- **We now know the noise in our data is normally distributed.** This means we now have a set of random points that have error that is normally distributed, where the chances of getting a value of x depends on the values of the standard deviation of the noise and the interval:

$$P(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{13.9}$$

In this section, we created a pdf that gives us the range for our model.

We now wish to maximise the likelihood of finding the best model. We can show that in the below proof:

Proof 13.1: Maximising the likelihood using Log Laws

We know we have a generating function that is a polynomial of the form

$$y = \sum_{i=0}^{n} a_i x^i$$

and we want to approximate this function using some n sample points, x_i, x_2, x_3 ... x_n to create a dataset.
For these n points, we know that error exists in each measurement in the dataset, and that the error is normally distributed.

$$pdf(error) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{\frac{-(\epsilon-\mu)^2}{2\sigma^2}}$$

We approximate the function that generated this dataset by realising the difference between the generating function and the model is the error:

$$\hat{y}_i = y_i + \epsilon_i$$

Our probability distribution for the total event that uses **all** these points with all these combined errors is known as the **pdf for our model**. To review the equation, please see Note 1.3:

$$pdf(\hat{y}, \sigma) = \prod_{i=1}^{n} pdf(y_i|\hat{y}_i, \sigma)$$

Using the results in Note 1.3, we can take the log of both sides of this and expand out in similar fashion to (13.7).

We use the results from (13.8) to apply it as a sum of terms:

$$\ln(pdf(\hat{y}, \sigma)) = \sum_{i=1}^{n} \left(-\ln(\sigma\sqrt{2\pi}) - \frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right)$$

Here, we can **ignore constants** like $\ln(\sigma\sqrt{2\pi})$ because we are trying to find the maximum pdf based on our variables.

$$ln(pdf_{max}) = \sum_{i=1}^{n} \left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right)$$

$$ln(pdf_{max}) = -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2)$$

Note that maximising the likelihood is the same as minimising the below term:

$$ln(pdf_{max}) = -\frac{1}{2\sigma^2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2)$$

Which we can see gives us OLS. Recall $SSR = \sum(y_i - \hat{y}_i)^2$

Note 13.4: Understanding Bayesian

We have assumed that the error we have in each point of our dataset is normally distributed. Now, we want to explore the belief we might have about our model. Our belief is that we assume our final model going to have fewer big terms than small terms.

We can see how we have used regression in our solution, and how this feeds into our problem statement by using OLS. We have also talked about LASSO and Tikhonov, and how we use this to minimise certain coefficients. But, how do we do this, and how does it fit into our problem statement and model?

This comes about through Bayesian Statistics.

13.5 Belief and Bayesian Statistics

Bayesian statistics is a way of quantifying the certainty or belief that we have about a particular outcome, by combining what is called prior beliefs about a parameter, with evidence from data to form a posterior distribution.

Prior belief is the idea that one can make informed decisions based on what we believe about a situation. Prior belief hopefully becomes less important as you collect more data but in the initial stages of an experiment it can be useful if this data has not been collected.

For example, in figure 13.9, we can see that there is more than one candidate for the line of best fit - however, if we were using a prior belief, we might say that we have a higher belief that our line of best fit is modelled by the green model, as opposed to the red model.

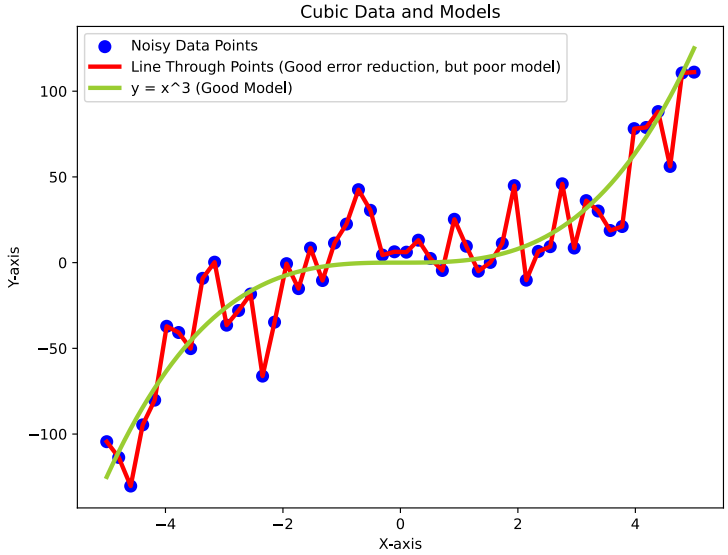


Figure 13.9: We prefer the smoother solution

The posterior distribution represents an updated belief about the parameter, reflecting both what was known beforehand and what was learned from the observations.

This is different to the frequentist approach where no prior belief about a situation is used, and instead the probability of an event occurring is purely based off the long-term calculation of the frequency of an event.

Bayesian is helpful in situations where we have intuition based off physical evidence or previous experimentation, which we can then use in our approximations to help inform the probability of an event.

In our case, we **do** have a prior belief about our function. Without knowing much about the model, we believe it is going to be better approximated by a function of lower order than higher order (such as we have shown in our linear function examples).

Recall when we introduced LASSO and Tikhonov. We have seen how LASSO applies this minimisation equally across all the coefficients and how Tikhonov places a larger minimisation on larger coefficients than smaller ones. **We can use both of these to model our belief.**

This is directly related to Stone-Weierstrass theorem [10], and Runge's Approximation [1]:

Theorem 13.1: Stone-Weierstrass

In mathematical analysis, the Weierstrass approximation theorem states that every continuous function defined on a closed interval [a, b] can be uniformly approximated as closely as desired by a polynomial function.

That is, there exists an $f(x)$ defined over $[a, b]$, a set of polynomial functions $P_n(x)$ for $n = 0, 1, 2, ...$ each of maximum order n that will converge and approximate our continuous function.

So it seems reasonable of us to wish to approximate a continuous function (or, say, what we imagine it to be in the shape of our random data) with a polynomial function. We also have Runge's:

Theorem 13.2: Runge's Approximation

In relation to the Stone-Weierstrass approximation theorem, for a dataset modelled by a polynomial approximation, the polynomial with the highest order is not always the best fit for the data, and in fact, can deteriorate quite quickly outside the scope of the dataset and provide unreliable approximations. For a continuous function, or continuous data, this is a problem.

That is, if there are 25 points of data, by Stone-Weierstrass theorem we might expect a polynomial function with highest order 25 (so $f(x) = 76 + 1.4x + 2.3x^2 + ... + 3.94x^{25}$) might seem like a reasonable solution within the bounds of the dataset. But, if it was used outside the scope of the dataset to predict a future value, for instance, the chances it is accurate are very small.

So just like the approximations in figure 13.4, we know that simply mapping a curve to all the available points in a graph could cause overfitting - and we now know, from Runge's Approximation, that outside our dataset, the function can behave quite erratically.

Ok, great! We know we can use Bayesian statistics, how do we use it?

Bayes' theorem is used to calculate the probability of a hypothesis or event, given prior belief.

Theorem 13.3: Bayes Theorem

Bayes Theorem [7] can be described as follows:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

H represents our prior belief, whose probability is not affected by data.
P(H), the prior probability, is the estimate of the beliefs of the hypothesis **H** before the data **E**, the current evidence, is observed.

Above, **E** is our evidence, and corresponds to new data that was not used in computing the prior probability. We can think of this being the real world data that we might re-train our hypothesis with, when we compare to situational data.

And **P(H | E)**, the posterior probability, is the probability of **H** given **E**, i.e., after **E** is observed. This is what we want to know: the probability of a hypothesis given the observed evidence.

Then **P(E|H)** is the probability of observing **E** given **H**. This is our likelihood. Because **E** and **H** are fixed, **P(**$E|H$**)** indicates the compatibility of the evidence with the given hypothesis. This

is a good indicator when we are trying to examine how well the evidence supports our hypothesis, and any adjustments that we might want to make in the future.

This then gives rise to the following observation, that the likelihood function is a function of the evidence, **E**, while the posterior probability is a function of the hypothesis, **H**.

P(E) can be seen as the marginal likelihood or "model evidence". So, when we are examining how well our probabilities went for the given situation, we can look at the evidence.

This factor is the same for all possible hypotheses being considered (as is evident from the fact that the hypothesis **H** does not appear anywhere in the symbol, unlike for all the other factors) and hence does not factor into determining the relative probabilities of different hypotheses.

Note 13.5: Bayes Statistics and a prior belief about a model

Here, we can think of this probability distribution showing the chances of, for instance, the coefficients of the algorithm being high, given that we have a prior belief that we expect, for instance, random data to be fairly well modelled by a polynomial with a power no higher than, say 100.

Our belief in this, that is quite high, which would imply that σ, the standard deviation, is quite small because we are not expecting a wide range of values for the order of the co-efficient.

Part of our project relies on our investigation of the certain functions that would suit as modelling functions for the penalty λ that we wish to place on coefficients of higher power in our polynomial approximation.

Runge's theory suggests to us that solutions with higher powers do not always give well defined answers, and can be wildly inaccurate when used to predict new data points. Based on this our prior belief is we prefer a simpler solution. The likelihood of having a model with higher powered terms is thus reduced:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

Can be arranged as:

$$P(H = model|E = data) \times P(E) = P(E|H = model) \times P(H = model) \tag{13.10}$$

- H = model
- P(H) = P(Model) is our prior distribution. Basically most models have near zero coefficients for large models. And, we have a belief behind what our prior distribution should be, and this is informed, for example, by the normal distribution we think the coefficients should range in.
- P(E|H) = probability to observe the data given your model. Basically, for some models this would be unlikely, for some models more likely (and of course we will pick the model for which the observation is most likely). We wish to maximise this probability. The probability of our data being correct given the model is based on the pdf for each single sample point that we use for creating our model. This is our pdf for the points.
- P(E) = probability of observing the data, regardless of the model. This constant is independent of H and will end up being ignored.

Let P(E) = k. Substituting this into our equation, we can say:

$$P(H = model|E = data) \times k = P(E|H = model) \times P(H = model)$$

in the values for probability of observing the data, our pdf for our coefficients, and the pdf for our samples points:

We also know that our model contains m coefficients ranged according to the normal distribution, and that the highest ordered term in our polynomial is x^m. Let us call this coefficient function $C(a_i)$.

The total chance of a coefficient for each power of x in our approximate function can be calculated using $C_i = \prod_{i=1}^{m} C(a_i)$.

Substituting this value for P(model) into our equation:

$$P(H = model|E = data) \times k = P(E|H = model) \times \left(\prod_{i=0}^{m} C(a_i)\right)$$

We also have a value for probability of our data given our model, which is the number of sample points calculated with error:

$$pdf(H = model|E = data) \times k = \left(\prod_{i=1}^{n} f(y_i|\hat{y}_i, \sigma)\right) \times \left(\prod_{i=0}^{m} C(a_i)\right)$$

Which we can re-write as:

$$pdf(\hat{y}, \sigma) = \left(\prod_{i=1}^{n} f(y_i|\hat{y}_i, \sigma)\right) \times \left(\prod_{i=0}^{m} (C(a_i))\right) \tag{13.11}$$

Now, near the end of this chapter, we have been able to not only formalise our belief into a framework for creating a simple, accurate model, but can now assess how well it performs. This final problem update brings together all components one last time to present our final statement.

Problem Statement 13.5: Update: Bayesian Framework

- We have a set of random data, and we wish to find a model that will accurately explain this data.
- We don't know what the generating function for our dataset was, but we are assuming it is a polynomial of the below form, where degree m is unknown:

$$f(x) = \sum_{i=0}^{m} a_i x^i$$

- We know that the data we use to make our model will have some errors, but that we will be assessing the accuracy of our model using OLS
- We will be using LASSO (RSS + $\lambda \times ||C(a_S)||$) and Tikhonov regularization (RSS + $\lambda \times ||C(a_S)||^2$) to find the smoothest solution for model.
- We know that a polynomial approximation can provide a reasonable solution, given Stone-Weierstrass theorem.

- We model the range of our datasets' errors using a probability density function
- We don't know the order of our polynomial, but our **prior belief** is that the function approximating the data will be as simple as possible. So for instance, if we have 50 points of data, we certainly don't believe it should be modelled by an order-50 polynomial.
- **We will use an equation to change the value of λ in our regularizer.**
- **This penalty function will help determine what weight we place on the terms of larger order in our polynomial**
- **We have some different penalty functions for generating different values for λ, which will be shown in our Analysis page**
- **Equation (13.10) gives us the general form Bayes Theorem to describe our model**
- Equation (13.11) is created by inserting the values we have to each components of Bayes into (13.10)

Proof 13.2: Maximising Likelihood of finding solution with our prior belief

We know we have a generating function that is a polynomial of the form

$$y = \sum_{i=0}^{n} a_i x^i$$

and we want to approximate this function using some n sample points, x_i, x_2, x_3 ... x_n to create a dataset.

For these n points, we know that error exists in each measurement in the dataset, and that the error is normally distributed.

$$pdf(error) = \frac{1}{\sqrt{2\pi\sigma^2}} \times e^{\frac{-(\epsilon-\mu)^2}{2\sigma^2}}$$

We approximate the function that generated this dataset by realising the difference between the generating function and the model is the error:

$$\hat{y}_i = y_i + \epsilon_i$$

Since updating our belief about our model, it can now be described by (13.11)

$$P(model|data) \times P(data) = P(data|model) \times P(model)$$

Borrowing from equation (13.8), we can model the pdf for the error in all values used for our approximation (see Note 1.3 for more detail):

$$pdf(\hat{y}, \sigma) = \prod_{i=1}^{n} f(y_i|\hat{y}_i, \sigma) \times \prod_{i=0}^{m} (C(a_i))$$

Using the results from Note 1.3 and 13.8, we can put all this together and maximise the likelihood of an outcome:

$$ln(pdf(\hat{y}, \sigma))) = \sum_{i=1}^{n} \left(-\ln(\sigma\sqrt{2\pi}) - \frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right) + \left(\sum_{i=1}^{m} \ln(C(a_i)) \right)$$

Maximising the pdf by ignoring constants, we use the results from (13.8):

$$ln(pdf_{max}(\hat{y}, \sigma)) = \sum_{i=1}^{n} \left(-\frac{(y_i - \hat{y}_i)^2}{2\sigma^2} \right) + \left(\sum_{i=1}^{m} \ln(C(a_i)) \right)$$

This is also equivalent to the addition of RSS and LASSO:

$$ln(pdf_{max}(\hat{y}, \sigma)) = RSS + \ln ||C(a_i))||$$

- Where $\hat{y}$ is the approximation function
- With $C(a_i)$ as our coefficient function
- And $C(a_i) = \frac{1}{\gamma_i \times \sqrt{2\pi}} \times e^{\frac{-\alpha_i}{2\gamma_i}^2}$
- Where λ_i is our standard deviation, our σ, which represents the confidence we have in coefficients of a large size.
- We have n sample points
- And x^m is the highest powered term in our approximate polynomial

When we first started this chapter, we knew we wanted to create a model, but we didn't have the best idea how. We weren't sure how to quantify the doubt we had in our dataset, nor the belief we had, that an accurate model was also likely to be more simple.

Now, in this final section, we have explored how we can quantify our belief and used Bayes Theorem to not only maximise the chances of finding a best performing solution, but to show that this end goal directly retrieved LASSO regularization.

If we use a different value for $C(a_i)$ we could have easily also have retrieved Tikhonov.

13.6 Analysis

The point of this journey has been to introduce readers to the idea of how an informed solution is possible by modelling our belief about a situation, calculating the noise in our data samples, identifying a method of creating modelling for our points, as well as a healthy smattering of statistics. Now we can show the results of using this framework and model our results.

Setup

To start, let us demonstrate a single experiment. All functions have been modelled off the same dataset. There are 30 sample points, noise is 0.2, and the original function that generated these points, our actual function, has its largest powered term at x^{10}.

The penalty functions that we briefly introduced earlier, are applied to our λ and then used in our model.

This is where our prior belief is being demonstrated because, our penalty functions place a greater penalty on larger coefficients than on smaller ones, therefore severely limiting the higher coefficients size relative to the smaller coefficients, as modelled in figure 13.10.

We pick $C(a_i)$ where $\frac{1}{\gamma_i \times \sqrt{2\pi}} \times e^{\frac{-\alpha_i}{2\gamma_i}^2}$ where $\gamma_i = g(i)$ where g is one of the following functions, and $r \in [0, 1, 0.02]$

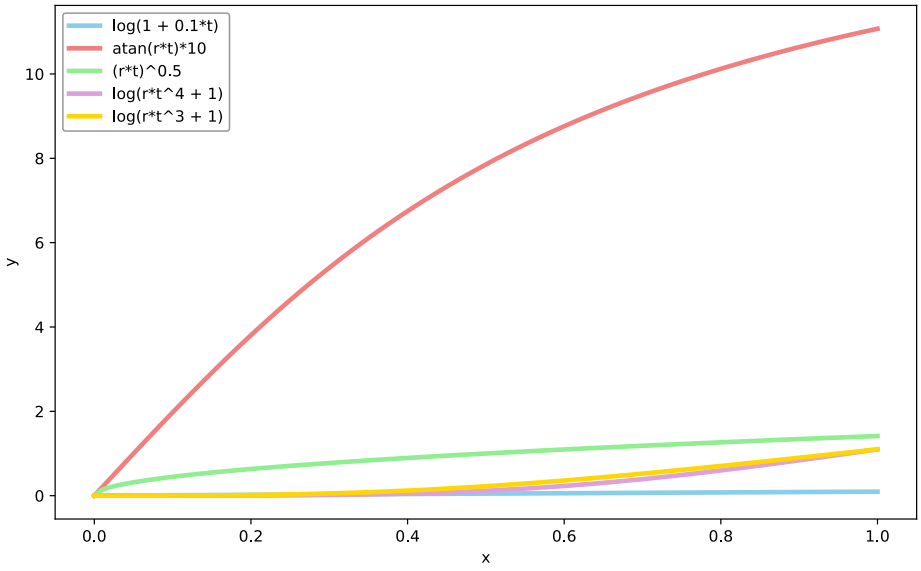


Figure 13.10: Penalty on the coefficients of x modelled by $y = \sum_{i=0}^{100}$ for $(a_i f(x))$ for x in (0..100)

- $\log(1 + 0.1 \times t)$
- $\arctan(rt) \times 10$
- $\sqrt{rt}$
- $(r \times t^4 + 1)$
- $\log(rt^3 + 1)$

These functions are used to calculate our λ, which is used as the parameter by which to tune our regularization. Recall, the larger λ is, the larger the penalty it places on the coefficients.

Results

More thorough testing necessarily means testing a massive wide variation of values for noise, data points, and the expected size of the polynomial. Whilst all experiments are random, the results of this experiment are indicative of previous experiments.

The purpose of further experimentation will be to further study the six equations we could use to model the belief we have about our coefficients. Neither have any merit over the other - these are all at this stage seen as good solutions for modelling our belief. This is where future experimentation will allow us to understand which method may be best. Below are the polynomial

approximations, ordered from function with least error (when compared to the actual function) to most error.

Note, that the first ten terms in each approximation function have been highlighted so that we are able to see the difference in size of each of the approximation functions.

We randomly generated our actual function, which is $y = 2.58 + 5.04x - 6.893x^2 - 17.720x^3 + 3.611x^4 + 22.227x^5 + 3.834x^6 - 11.785x^7 - 4.154x^8 + 2.211x^9 + 1.0x^{10}$.

The final example shows the results of a function that models the dataset without any regularization at all (Figures 13.18 and 13.19).

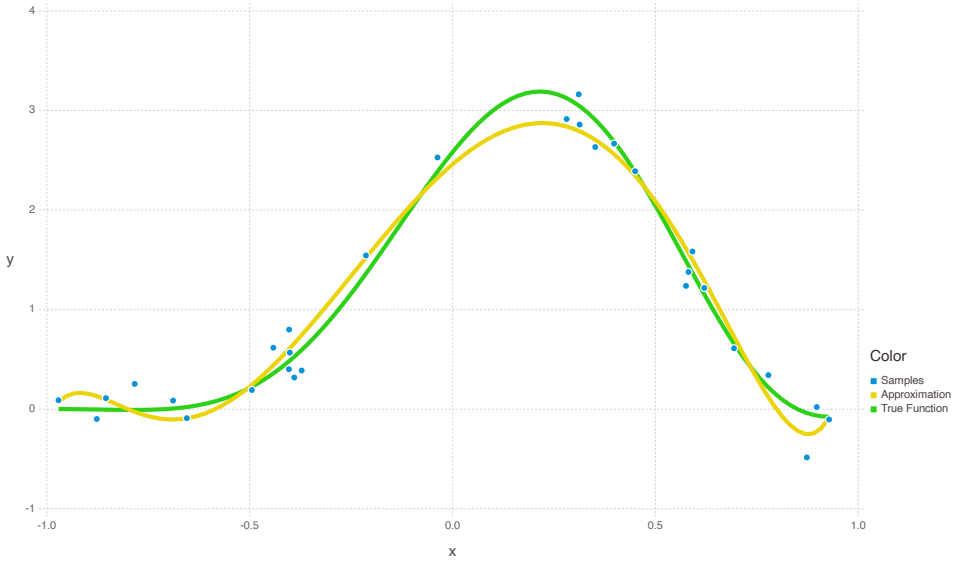


Figure 13.11: Prior belief in function 1 modelled by $h(r) = t \rightarrow \arctan(r * t) * 10$ and the resulting model is $\boldsymbol{y = 2.46 + 3.47x - 5.872x^2 - 6.719x^3 + 2.401x^4 + 0.776x^5 + 0.928x^6 + 1.127x^7 + 0.309x^8 + 0.767x^9 + 0.103x^{10}} + 0.472x^{11} + 0.039x^{12} + 0.286x^{13} + 0.02x^{14} + 0.174x^{15} + 0.015x^{16} + 0.106x^{17} + 0.014x^{18} + 0.065x^{19} + 0.013x^{20} + 0.04x^{21} + 0.013x^{22} + 0.024x^{23} + 0.012x^{24} + 0.014x^{25} + 0.01x^{26} + 0.008x^{27} + 0.009x^{28} + 0.004x^{29} + 0.008x^{30} + 0.001x^{31} + 0.007x^{32} + 0.006x^{34} - 0.001x^{35} + 0.005x^{36} - 0.001x^{37} + 0.004x^{38} - 0.002x^{39} + 0.004x^{40} - 0.002x^{41} + 0.003x^{42} - 0.002x^{43} + 0.003x^{44} - 0.002x^{45} + 0.002x^{46} - 0.002x^{47} + 0.002x^{48} - 0.001x^{49} + 0.002x^{50} - 0.001x^{51} + 0.002x^{52} - 0.001x^{53} + 0.001x^{54} - 0.001x^{55} + 0.001x^{56} - 0.001x^{57} + 0.001x^{58} - 0.001x^{59} + 0.001x^{60} - 0.001x^{61} + 0.001x^{62} - 0.001x^{63} + 0.001x^{64} - 0.001x^{65} + 0.001x^{66} - 0.001x^{67} + 0.001x^{68} - 0.001x^{69}$

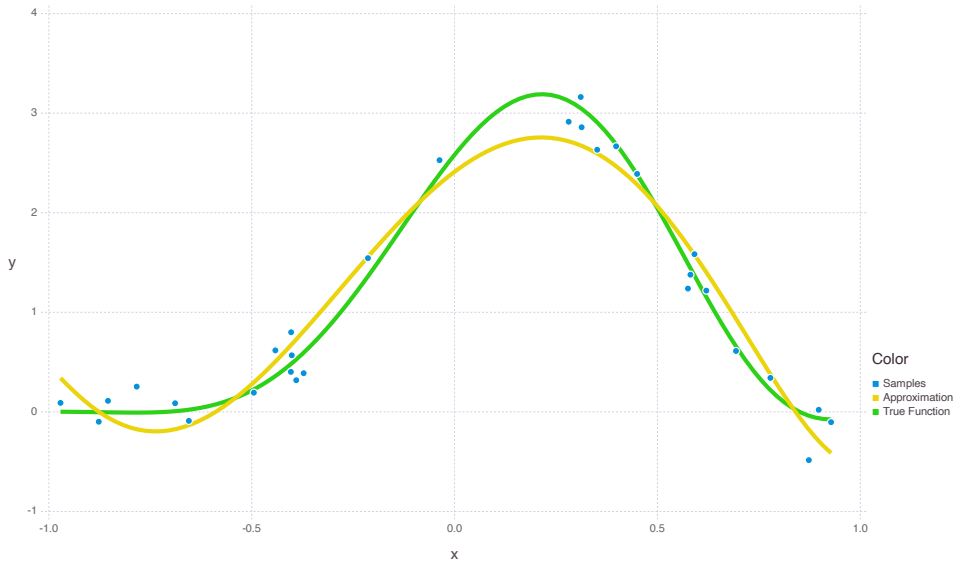


Figure 13.12: Prior belief in function 2 modelled by $p(r) = t \rightarrow \log(r * t^3 + 1)$ and the resulting model is $\boldsymbol{y = 2.268 + 2.827x - 4.434x^2 - 4.101x^3 + 0.554x^4 - 0.741x^5 + 0.768x^6 + 0.241x^7 + 0.537x^8 + 0.441x^9 + 0.344x^{10}} + 0.424x^{11} + 0.22x^{12} + 0.357x^{13} + 0.144x^{14} + 0.288x^{15} + 0.096x^{16} + 0.228x^{17} + 0.065x^{18} + 0.181x^{19} + 0.045x^{20} + 0.143x^{21} + 0.031x^{22} + 0.114x^{23} + 0.021x^{24} + 0.091x^{25} + 0.014x^{26} + 0.073x^{27} + 0.009x^{28} + 0.059x^{29} + 0.005x^{30} + 0.048x^{31} + 0.003x^{32} + 0.04x^{33} + 0.001x^{34} + 0.033x^{35} - 0.001x^{36} + 0.027x^{37} - 0.002x^{38} + 0.023x^{39} - 0.002x^{40} + 0.019x^{41} - 0.003x^{42} + 0.016x^{43} - 0.003x^{44} + 0.014x^{45} - 0.003x^{46} + 0.012x^{47} - 0.003x^{48} + 0.01x^{49} - 0.003x^{50} + 0.009x^{51} - 0.003x^{52} + 0.008x^{53} - 0.003x^{54} + 0.007x^{55} - 0.003x^{56} + 0.006x^{57} - 0.003x^{58} + 0.005x^{59} - 0.003x^{60} + 0.005x^{61} - 0.003x^{62} + 0.004x^{63} - 0.003x^{64} + 0.004x^{65} - 0.002x^{66} + 0.003x^{67} - 0.002x^{68} + 0.003x^{69} - 0.002x^{70} + 0.003x^{71} - 0.002x^{72} + 0.003x^{73} - 0.002x^{74} + 0.002x^{75} - 0.002x^{76} + 0.002x^{77} - 0.002x^{78} + 0.002x^{79} - 0.001x^{80} + 0.002x^{81} - 0.001x^{82} + 0.002x^{83} - 0.001x^{84} + 0.001x^{85} - 0.001x^{86} + 0.001x^{87} - 0.001x^{88} + 0.001x^{89} - 0.001x^{90} + 0.001x^{91} - 0.001x^{92} + 0.001x^{93} - 0.001x^{94} + 0.001x^{95} - 0.001x^{96} + 0.001x^{97} - 0.001x^{98} + 0.001x^{99} - 0.001x^{100}$

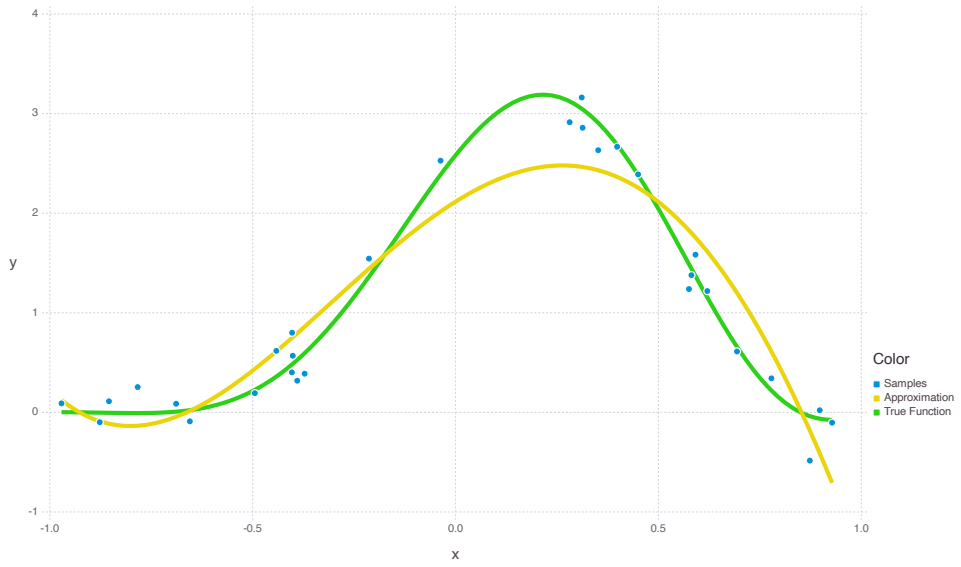


Figure 13.13: Prior belief in function 3 modelled by $n(r) = t \to \log(r * t^4 + 1)$ and the resulting model is $\boldsymbol{y = 2.116 + 2.553x - 3.598x^2 - 3.447x^3 + 0.788x^4 + 0.029x^5 + 0.057x^6 + 0.01x^7 + 0.011x^8 + 0.004x^9 + 0.004x^{10}} + 0.002x^{11} + 0.002x^{12} + 0.001x^{13} + 0.001x^{14} + 0.001x^{15} + 0.001x^{16}$

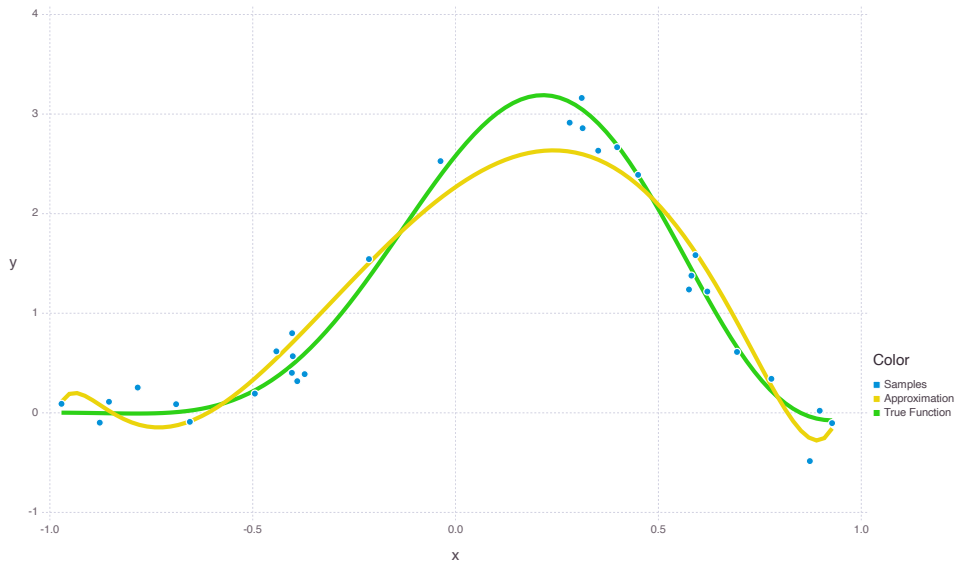


Figure 13.14: Prior belief in function 4 modelled by $m(r) = t \to (r * t)^(0.5)$ and the resulting model is: $\boldsymbol{y = 2.412 + 3.05x - 5.725x^2 - 5.412x^3 + 2.98x^4 + 1.353x^5 + 0.261x^6 + 0.327x^7 + 0.034x^8 + 0.097x^9 + 0.005x^{10}} + 0.037x^{11} + 0.017x^{13} - 0.001x^{14} + 0.009x^{15} - 0.001x^{16} + 0.005x^{17} - 0.001x^{18} + 0.003x^{19} - 0.001x^{20} + 0.002x^{21} - 0.001x^{22} + 0.002x^{23} + 0.001x^{25} + 0.001x^{27} + 0.001x^{29} + 0.001x^{31}$

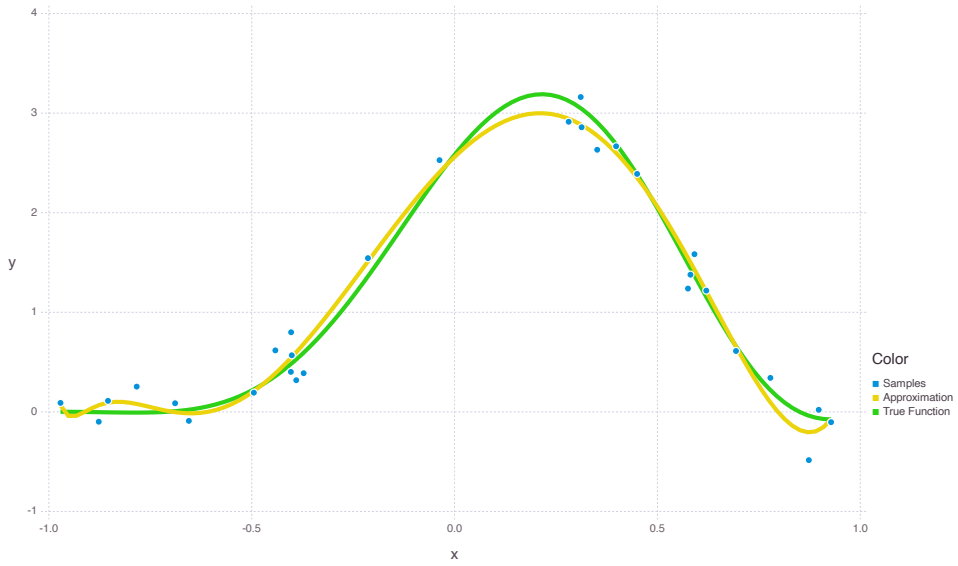


Figure 13.15: Prior belief is function 5 modelled by $g(r) = t \to \log(1+r*t)$ and the resulting model is: $\boldsymbol{y = 2.56 + 3.881x - 6.911x^2 - 8.759x^3 + 4.615x^4 + 2.261x^5 + 0.529x^6 + 1.916x^7 - 0.397x^8 + 1.016x^9 - 0.405x^{10}} + 0.471x^{11} - 0.243x^{12} + 0.183x^{13} - 0.099x^{14} + 0.036x^{15} - 0.002x^{16} - 0.037x^{17} + 0.055x^{18} - 0.071x^{19} + 0.083x^{20} - 0.085x^{21} + 0.094x^{22} - 0.088x^{23} + 0.095x^{24} - 0.085x^{25} + 0.09x^{26} - 0.08x^{27} + 0.082x^{28} - 0.073x^{29} + 0.074x^{30} - 0.066x^{31} + 0.066x^{32} - 0.059x^{33} + 0.058x^{34} - 0.052x^{35} + 0.051x^{36} - 0.046x^{37} + 0.044x^{38} - 0.041x^{39} + 0.039x^{40} - 0.036x^{41} + 0.034x^{42} - 0.032x^{43} + 0.03x^{44} - 0.028x^{45} + 0.026x^{46} - 0.025x^{47} + 0.023x^{48} - 0.022x^{49} + 0.02x^{50} - 0.019x^{51} + 0.018x^{52} - 0.017x^{53} + 0.015x^{54} - 0.015x^{55} + 0.014x^{56} - 0.013x^{57} + 0.012x^{58} - 0.012x^{59} + 0.011x^{60} - 0.011x^{61} + 0.01x^{62} - 0.009x^{63} + 0.009x^{64} - 0.008x^{65} + 0.008x^{66} - 0.008x^{67} + 0.007x^{68} - 0.007x^{69} + 0.006x^{70} - 0.006x^{71} + 0.005x^{72} - 0.005x^{73} + 0.005x^{74} - 0.005x^{75} + 0.004x^{76} - 0.004x^{77} + 0.004x^{78} - 0.004x^{79} + 0.004x^{80} - 0.004x^{81} + 0.003x^{82} - 0.003x^{83} + 0.003x^{84} - 0.003x^{85} + 0.003x^{86} - 0.003x^{87} + 0.002x^{88} - 0.002x^{89} + 0.002x^{90} - 0.002x^{91} + 0.002x^{92} - 0.002x^{93} + 0.002x^{94} - 0.002x^{95} + 0.002x^{96} - 0.002x^{97} + 0.001x^{98} - 0.001x^{99} + 0.001x^{100}$

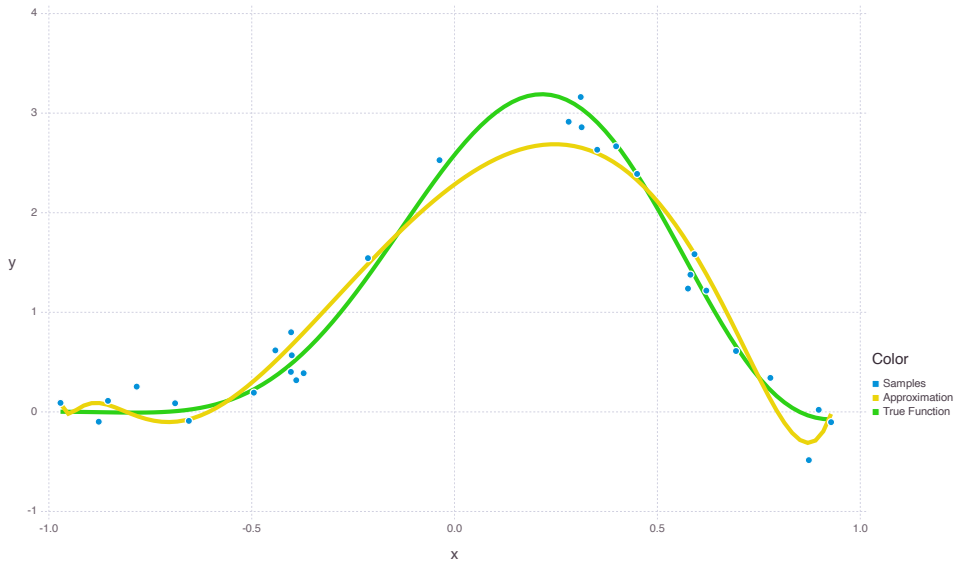
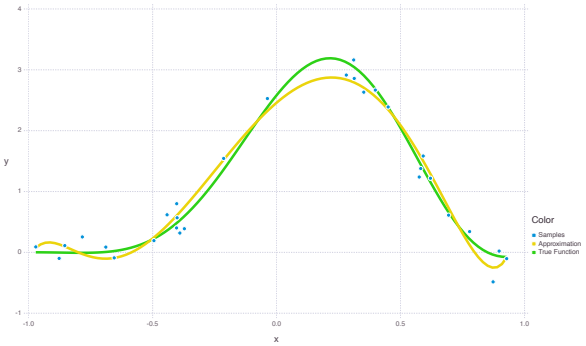
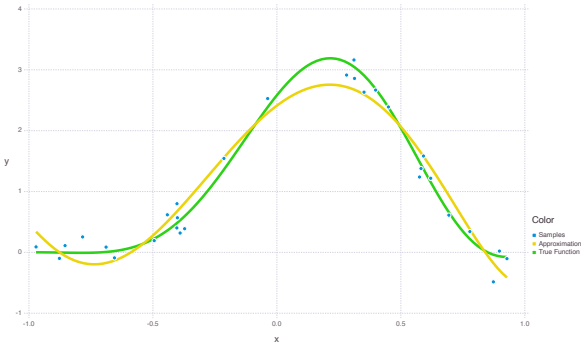


Figure 13.16: Prior belief in function 6 is modelled by the constant value for λ and the resulting model is: $\boldsymbol{y = 2.284 + 2.999x - 4.441x^2 - 4.374x^3 + 0.158x^4 - 1.491x^5 + 1.009x^6 + 0.224x^7 + 0.813x^8 + 0.823x^9 + 0.485x^{10}} + 0.913x^{11} + 0.237x^{12} + 0.81x^{13} + 0.086x^{14} + 0.652x^{15} + 0.006x^{16} + 0.492x^{17} - 0.031x^{18} + 0.351x^{19} - 0.042x^{20} + 0.233x^{21} - 0.041x^{22} + 0.139x^{23} - 0.033x^{24} + 0.065x^{25} - 0.023x^{26} + 0.009x^{27} - 0.013x^{28} - 0.033x^{29} - 0.004x^{30} - 0.063x^{31} + 0.004x^{32} - 0.084x^{33} + 0.011x^{34} - 0.099x^{35} + 0.017x^{36} - 0.107x^{37} + 0.021x^{38} - 0.112x^{39} + 0.025x^{40} - 0.114x^{41} + 0.028x^{42} - 0.113x^{43} + 0.03x^{44} - 0.11x^{45} + 0.032x^{46} - 0.107x^{47} + 0.033x^{48} - 0.102x^{49} + 0.034x^{50} - 0.098x^{51} + 0.034x^{52} - 0.093x^{53} + 0.035x^{54} - 0.087x^{55} + 0.035x^{56} - 0.082x^{57} + 0.034x^{58} - 0.077x^{59} + 0.034x^{60} - 0.072x^{61} + 0.034x^{62} - 0.068x^{63} + 0.033x^{64} - 0.063x^{65} + 0.032x^{66} - 0.059x^{67} + 0.031x^{68} - 0.055x^{69} + 0.031x^{70} - 0.051x^{71} + 0.03x^{72} - 0.048x^{73} + 0.029x^{74} - 0.045x^{75} + 0.028x^{76} - 0.042x^{77} + 0.027x^{78} - 0.039x^{79} + 0.026x^{80} - 0.036x^{81} + 0.025x^{82} - 0.034x^{83} + 0.024x^{84} - 0.032x^{85} + 0.023x^{86} - 0.03x^{87} + 0.022x^{88} - 0.028x^{89} + 0.021x^{90} - 0.026x^{91} + 0.02x^{92} - 0.024x^{93} + 0.019x^{94} - 0.023x^{95} + 0.018x^{96} - 0.021x^{97} + 0.017x^{98} - 0.02x^{99} + 0.016x^{100}$

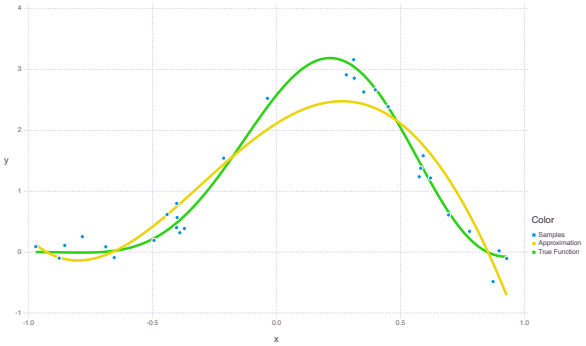
Figure 13.17: Side by Side Comparison of 6 different models labelled by generating penalty function



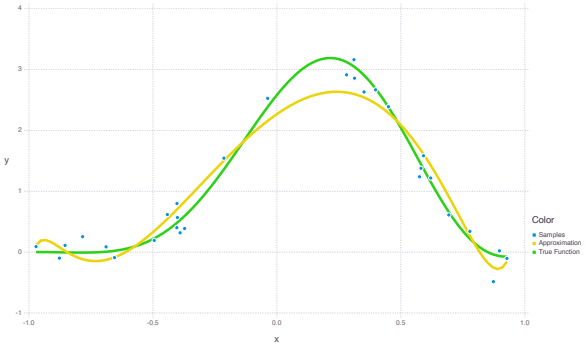
Atan penalty function



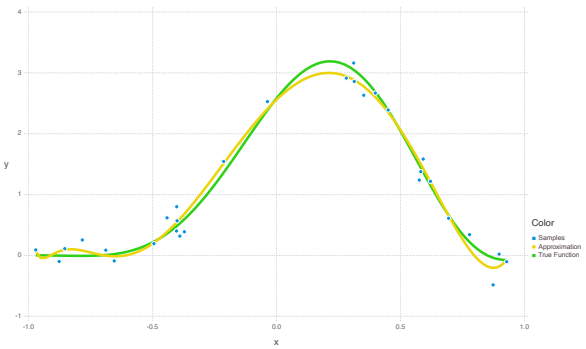
Log^3 penalty function



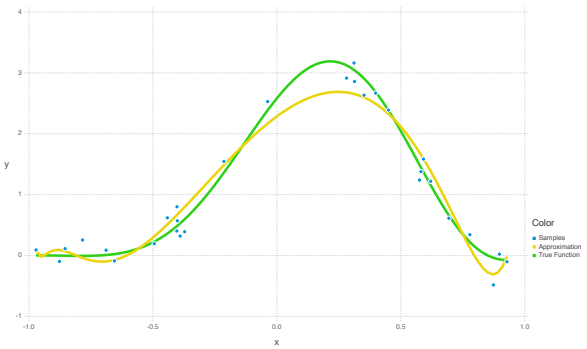
Log^4 penalty function



$\sqrt{}$ penalty function



ln penalty function



Constant penalty function

Figure 13.18: The original function is barely visible on this scale

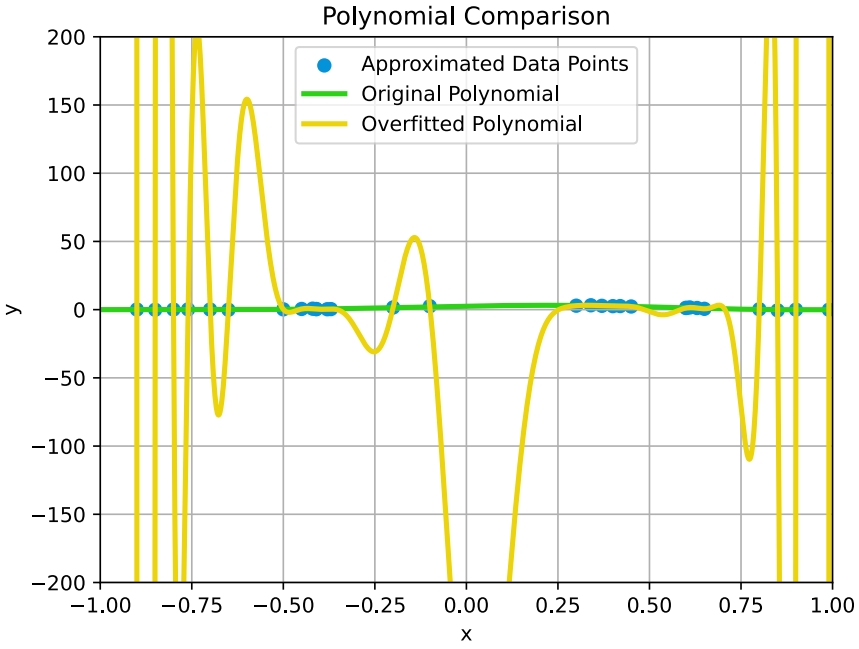
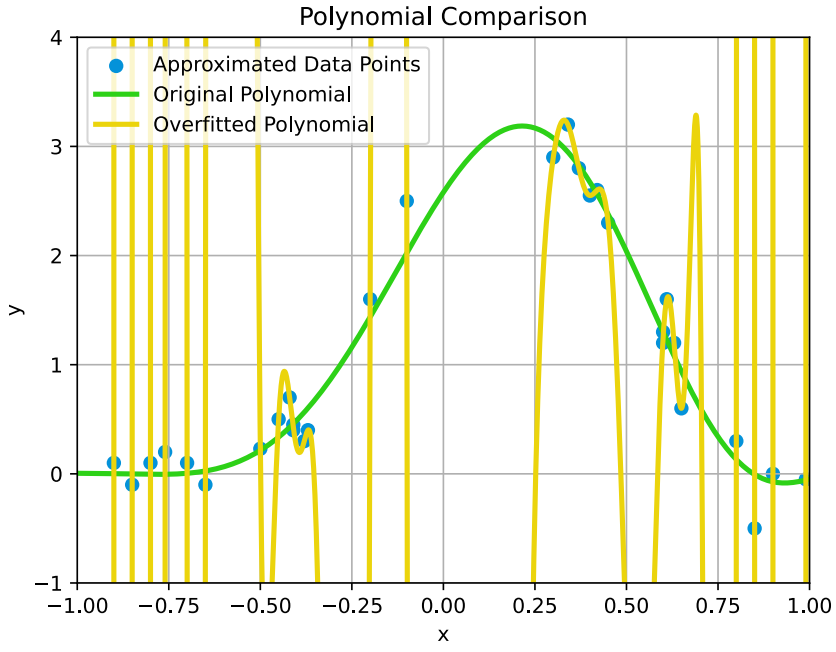


Figure 13.19: But, when we zoom up, we can see that this approximation passes through all points



Comparison of the Polynomials

We observe the following:

- The 1st, 2nd and 3rd, as the best performing solutions are similar (and the coefficients also verify this)
- The 6th polynomial was approximated using a penalty function that has a belief, or penalty, equal for all coefficients. This means that it fits each point more sharply than the others
- While the 6th polynomial has regularization, so is still a good approximation, it still has the largest number of local minimums and maximums which make using it for prediction less likely
- Each polynomial performs significantly better than the 6th, and we can see this is because a single constant wasn't applied to each coefficient for each of the terms in the polynomials

Comparison of the Coefficients

- The first polynomial performs the best and has the size of the terms drop off fairly quickly – for approximation, this is what we wish to see. What predicted that our most accurate solution would have a smaller number of coefficients than others
- In this experiment, the polynomial that had the smallest number of terms was the third, but did not have the least error which links to another future research question about how we balance the size of the solution with its complexity
- The first three approximations performed very similarly, indicating that either would be suitable model approximations
- The last polynomial approximation has the same weight for each parameter, which means all of the terms are of similar size
- The 7th, without regularization, was incredibly wild and erratic

Limitations

We can see that all solutions perform better than both the non-regularized model, and the model that used a constant unchanging value of λ. This is an indication that the experimentation results support our mathematical framework.

Limitations to this are that we have artificially sized our generating functions to contain terms up to 100 places, to demonstrate that most terms generated will be much lower than this number. But, because of this, all models will contain terms up to the 100th place.

Another limitation is whether rounding should be introduced, and the effect of this. Some polynomials appear to have fewer terms when they are rounded to 3 places, because the smallest of them disappeared. Relatively speaking as well, because of this rounding, some models might appear to be smaller and therefore easier to compute.

Another limitation is the confidence we have in what the results suggest based off a single experiment. Depending on the amount of noise, the number of points, and the number of terms in our polynomial, the "best fit" does not come from the same penalty function each time.

This where where we would use numerical analysis, which would involves many thousands or millions of trials, to make a definite determination about the best way to find a good model.

Significance

There was estimated to be some 147 Zettabytes of data generated at the end of 2024 [3]. That is more bytes of data than there are stars in our entire universe.

The significance of this work is to create a framework or a method of thinking about finding a "best" solution that is also the simplest. Whilst we might never have to use all of the data we possible have to create a model, and we would in all cases try to be selective about the kinds of data we use to create a model, sometimes we don't always have this option.

There are some fields, like medical imaging, where processes happen that contain errors, and yet accurate decisions still have to be made. Image scans for example can have all kinds of background noise like patient movement or physical differences, for example. The significance of a model framework that acknowledges this noise in a dataset is valuable, and the way we have demonstrated that it is possible to create simpler solutions over more complicated ones, and yet still not given up on accuracy is also valuable.

Although we strive to build models using curated, high-quality data, real-world scenarios frequently demand we work with whatever data is available, regardless of its quality. This project acknowledges that challenge, by allowing us to quantify our belief in what a likely model should look like.

13.7 Code Implementation

This section covered the code that was used to produces the figures and coefficients in the Results Section. It was written in **Julia** version 1.3.14.

Julia is a high-level high-performance programming language specially designed for technical and time-intensive computing. It was created by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman, with its first public release in 2012 [4]. Julia, like Perl, finds a balance between speedy pre-compiled languages like C and Fortran, yet still maintains the readability of languages like Python. It allows you to problem solve in mathematics much the same way as you would on paper, including the use of Greek symbols in your programming so it can map perfectly to the functions that you are modelling [4]. It is a common language to use for data science, scientific computing, and numerical analysis, because of how fast is runs once compiled.

For our project, it was better to write in a functional programming style, rather than object oriented.

Creating our sample points involved creating points of data for our model to approximate. The values for γ and σ relate to the values for the normal distribution of the noise in the measurements.

```
# will take f
# function random experiment
# single experiment for our project
function single_sample(f,x, $\gamma$, $\lambda$, $\sigma$)
    y = add_gauss(f.(x), $\sigma$)
    #y = x.^2 + 2x
    b = make_b(x, y, $\lambda$, $\gamma$)
    p = plot_p(x, y, b)
    return b,p
end
```

Setting up our Bayesian regression and the polynomial solution is done by creating a function, using different values of λ based on a range of values.

```
function make_b(x, y, $\gamma$, $\lambda$)
    v = [t^i for t in x, i in 0:100]
```

```
    b = BayesianRegression(y, v, $\lambda$, $\gamma$)
    return Polynomial(b)
end
```

Defining a single experiment

The following is an excerpt to show how we can model six different functions for the prior function. Two of these six prior belief functions have been shown below:

```
        number_samples = 20 #but really should be 5:30
        highest_order = 10 #in range 5:30, be 1:100
        noise = 0.2
        e1 = Experiment(highest_order,noise,number_samples)  # current factor of 12...
        # return the best function coefficients based on least normalised distance
        # take in the bvalue, create distance, retutn bestvalue,
        function least_error_result_from_normalised_coeffs(bvalue)
          distances = distance.((e1,), [b[1] for b in bvalue])
          println(distances)
          bestvalue = argmin(distances)
          return bestvalue
        end

   #6 different Prior Belief Functions, 6 different "best"s
   # this one has no funciton because all coefficients are weighted equally
   b1 = single_sample.((e1,), 0, LinRange(0,0.05,100))
   bestlinearvalue = least_error_result_from_normalised_coeffs(b1)
   pol1,plot1 = b1[bestlinearvalue]

        g(r) = t \to log(1+r*t)
        b2 =single_sample.((e1,), 0, g.(LinRange(0.001,.05,100)))
        bestlogfuncvalue = least_error_result_from_normalised_coeffs(b2)
        pol2,plot2 = b2[bestlogfuncvalue]
        ....
```

Creating polynomial comparison methods

The former function was necessary to compare the polynomials and use the function **compareResults** to rank values, store and then save them.

The latter function takes in the values of the coefficients generated by the Bayesian regression and then compares each to see which polynomial generated has the least distance between the point it generates and the points that we randomly sampled.

```
my_optimisation_best_result = compareResults (bestlinearvalue,  bestlogfuncvalue,  bestatanvalue, bestlog3,
bestlog4, bestsqrtvalue)

    function least_error_result_from_normalised_coeffs(bvalue)
      distances = distance.((e1,), [b[1] for b in bvalue])
      println(distances)
      bestvalue = argmin(distances)
      return bestvalue
    end
```

Future renditions of this code will be generated to take in a variety of different functions to approximate, as well as looped trials to conduct thousands of experiments.

13.8 Summary

In this chapter, we review how to make an approximate function by rigorously defining what we mean by error, a range in the values of our errors, a good solution, and maximising likelihood. We moved from having a hand-wavy understanding of the problem we were facing, to being able to code and plot different best possible solutions to explain a dataset with randomly selected non-accurate elements.

We learned about probability density functions, the normal distribution, theorems associated with modelling probable functions and the ways in which we can use information about our prior belief to maximise our chances of finding a good model.

Finally, we were in a position to be able to write code to create some datasets, a hidden generating function, and assess how our prior beliefs affected the accuracy and complexity of our models. We can now experiment within this framework to study how some functions may work as better penalty functions than others, and which method works best most of the time.

Though future efforts will branch out to include the study of other functions, we could see how the effect of modelling our beliefs in the case of a polynomial function. Not only did the results re-affirm the conclusions that the mathematics explained, but it opened the doors on further works in the future.

Context

This work was only possible thanks to a project I started back in June 2024 with Julien Ugon. I have always wanted to have a maths project, and I have learned so much already. Outside the confines of a unit, I could enjoy the mathematics required for this topic and apply them in a way that improved our experimental optimisation outcomes. This project is by no means complete but this chapter here is a nice summary of the foundational knowledge that was required to start the real work.

About the Author

I'm currently a third year student in the Bachelor of Cybersecurity at Deakin University. I had such a hard time choosing what single course excited me the most that it would take me two years to commit to a (first) degree. Although my interests and hobbies range, I love the way mathematics has so many uses in so many different fields. In my first two years at Deakin I have been fortunate enough to have been involved in four different research projects, which further revealed to me what my true passions are. The work I've contributed for this book reflects my enthusiasm for mathematical optimisation and my love of learning. I am planning to pursue an Honours year to explore this further and to contribute to the field of mathematics. My hope is that people reading my chapter learned a lot about maths, and a lot about trying new things.

Acknowledgements

Special thanks goes to Julien, Nadia and Nadya for their amazing input and constant support, Phillip and Andrew for editorial advice, my CSOC team for their interest, and Otis for all the dinners that fed my soul. The best people :)

References

All plots, unless stated otherwise, were created using **Julia** version 1.3.14.

[1] Runge theorem. *Encyclopedia of Mathematics*, 2014. http://encyclopediaofmath.org/index.php?title=Runge_theorem&oldid=32114.

[2] What is LASSO regression? *IBM*, 2024. https://www.ibm.com/think/topics/lasso-regression.

[3] Troy Beamer. 402.74 Million Terrabytes of Data is Created Every Day. Technical report, Business Tech Data, 2024.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] Stanley Chan. ECE595 / STAT598: Machine learning I Lecture 31 Regularization (lecture notes). *Purdue University*, 2012.

[6] Gene H. Golub, Per Christian Hansen, and Dianne P. O'Leary. Tikhonov regulirizations and total least squares. *Society for Industrial and Applied Mathematics*, 21(1):185–194, 1999.

[7] James Joyce. Bayes' theorem. *The Stanford Encyclopedia of Philosophy, Fall 2021 Edition*, 2003. https://plato.stanford.edu/archives/fall2021/entries/bayes-theorem/.

[8] Akshay Padmanabha. Ridge regression. *Brilliant.org*, 2022.

[9] T. Poggio. The learning problem and regularization [online lecture notes]. *Massachusetts Institute of Technology*, 2004.

[10] M. H. Stone. The generalized Weierstrass approximation theorem. *Mathematics Magazine*, 21(4):167–184, 1948.

[11] T. Tantau. *The TikZ and PGF Packages*, 2013. http://sourceforge.net/projects/pgf/.

14

Phonetic spelling correction using dimensionality reduction

Louisa Best

Abstract

Phonetic spelling correction is essential for individuals who rely on non-standard phoneme-based spellings, including those with intellectual disabilities, early learners, and second-language users. Traditional text-based spell-checking struggles to handle these variations due to the complexity of phonetic similarity. This study investigates the use of dimensionality reduction techniques, specifically Principal Component Analysis (PCA) and Singular Value Decomposition (SVD), to improve phonetic word retrieval by reducing sparse, high-dimensional feature spaces. Both pre-scaled and post-scaled SVD were evaluated in terms of variance retention and phonetic similarity matching. Results indicate that post-scaled SVD significantly enhances phoneme representation, leading to more accurate word predictions when combined with cosine similarity. In contrast, Euclidean and Manhattan distances failed to adequately capture phonetic structure, highlighting the importance of feature scaling in phonetic vector spaces. These findings provide a foundation for further advancements in hybrid models, alternative similarity measures, and context-aware phonetic embeddings, with potential applications in assistive spelling tools, speech recognition, and language learning.

14.1 Introduction

Problem Statement and Motivation

Written language presents a barrier for individuals with intellectual disabilities (ID), early learners, and second-language speakers, many of whom rely on phonetic spelling—writing words as they sound rather than following conventional spelling rules. Standard spell checkers, designed for typographical errors, do not address phonetic misspellings, creating a gap in accessible assistive technologies [6, 3, 8]. For instance, using “halow” in place of “hello” and “ingre” instead of “injury” are phonetic spellings that standard spell checkers do not identify, underscoring the need for more comprehensive solutions.

Given the orthographic complexity of English, where phonemes (speech sounds) do not map directly to graphemes (letters), individuals using phonetic spelling require specialised correction tools. Existing phonetic algorithms such as *Soundex* and *Double Metaphone* fail in non-standard linguistic contexts, particularly for those with ID [21, 11, 14, 7].

To address these challenges, this research investigates phonetic spelling errors, existing correction methods, and phoneme feature representation before proposing an approach that takes advantage of *n*-gram vectorisation and dimensionality reduction to improve phonetic word

retrieval. The following sections outline the characteristics of phonetic misspellings, current limitations in phonetic correction, and the research contributions of this study.

Phonetic Spelling and Types of Spelling Errors

Phonetic misspellings arise due to deviations from conventional orthographic norms. These errors fall into three primary categories:

- **Phonological Errors**: Incorrect or missing phonemes *Example:* "ingre" → "injury" (missing /ʊ/ in /ɪnˈdʒʊri/) [5].
- **Orthographic Errors**: Incorrect letter substitutions, *Example:* "helo" instead of "hello" or "hallow" instead of "halo" → /həˈloʊ/ [19].
- **Morphological Errors**: Incorrect word formation, *Example:* "runned" instead of "ran", "mouses" instead of "mice" [1].

To effectively tackle these challenges, a reliable phonetic spelling correction system is crucial, as it needs to comprehend the fundamental phoneme representations and precisely link them to the appropriate words.

Phonetic Spelling Challenges and Existing Approaches

Phonetic spelling, in which words are written based on sound rather than conventional rules, is common among individuals with intellectual disabilities (ID) [3, 8], ESL learners [10], and children developing literacy [2]. Existing algorithms fail to address phonetic misspellings, instead focussing on typographical errors.

Traditional algorithms like *Soundex, Metaphone, and NYSIIS* encode words into phonetic representations but rely on strict letter-sequence rules, making them ineffective for non-standard phonetic variations [22, 13]. This requires more flexible and data-driven approaches.

Phonetic Feature Representation and Vectorisation

Phonetic spelling requires alternative linguistic representation beyond rule-based algorithms. Techniques include character-level embeddings, which represent words at the character level and improve the handling of spelling variants [16]; *n*-gram frequency matrices, which capture phoneme cooccurrence [17, 4]; and word embeddings (Word2Vec, GloVe), which encode words based on semantic similarity but fail to capture phonetic relationships [9, 12]. High-dimensional representations pose computational challenges, making dimensionality reduction necessary.

Research Gap and Project Contribution

Existing models fail in three important aspects: (1) capturing nuanced phoneme-to-grapheme mappings in non-standard spellings [18], (2) efficiently handling high-dimensional phoneme data without losing important structure [20], and (3) generalising well across different spelling patterns in small datasets [23].

To address these limitations, this study proposes a dimensionality reduction-driven approach to phonetic spelling correction. The method takes advantage of phonetic *n*-gram vectorisation instead of rule-based encodings, applies Singular Value Decomposition (SVD) and Principal Component (PCA) to extract principal phoneme features while preserving phonetic integrity, and integrates phonetic similarity metrics to improve word retrieval.

The following sections outline the methodology (Section 14.2), discuss experimental results (Section 14.3) and summarise findings and explore future directions for improving phonetic spelling models (Section 14.4).

14.2 Methodology

Phonetic spelling correction requires a structured data flow to process input, extract meaningful phonetic representations, and apply computational techniques for improved word retrieval. This section outlines the key steps in the correction process, from phonetic preprocessing to dimensionality reduction and similarity-based retrieval.

Phonetic Processing

The phonetic spelling correction system begins with preprocessing phonetic inputs, ensuring a structured representation suitable for computational analysis. The following subsections describe each stage of this process, including phoneme tokenisation, feature vectorisation and the application of dimensionality reduction to optimise computational efficiency.

Lexicon Selection

A corpus of **phoneme-to-word mappings** is required to serve as the reference dataset. Given the lack of a standard phonetic dataset for Australian English, we selected an IPA-based lexicon and structured phonemes as a discrete sequence $\mathbf{P} = (p_1, p_2, ..., p_n)$ where each phoneme p_i represents an atomic speech sound.

Phoneme Tokenisation

A **word**, denoted as w, is defined as a sequence of characters representing a discrete linguistic unit. In this context, words consist solely of alphabetic letters without spaces or punctuation, and their length varies depending on the input.

Each word is tokenised into *n*-grams (bigrams and trigrams) to capture phonetic structures. This process involves breaking the word into overlapping sequences of n consecutive characters. For example, the word *"hello"* is decomposed as follows:

$$\text{“hello”} \rightarrow \{\text{“}he\text{”}, \text{“}el\text{”}, \text{“}ll\text{”}, \text{“}lo\text{”}\} \quad \text{(bigrams)}$$

$$\text{“hello”} \rightarrow \{\text{“}hel\text{”}, \text{“}ell\text{”}, \text{“}llo\text{”}\} \quad \text{(trigrams)}$$

N-grams capture local patterns in phoneme sequences, which are crucial for distinguishing between different phonetic structures. *Bigrams* model direct adjacent phoneme relationships, while *trigrams* provide a broader phonetic context. This multilevel representation enhances the model's ability to generalise across phonetic variations, improving similarity matching for phonetic spelling correction.

Phoneme-to-Letter Mapping

English orthography is non-phonemic, meaning that the spelling of words does not always correspond directly to their pronunciation. To address this, a mapping function $f : \mathbf{P} \rightarrow \mathbf{L}$ was developed, where f assigns phoneme sequences to plausible letter sequences. Some phonemes lacked direct English equivalents, necessitating the use of approximation mappings:

$$f(p_i) = \begin{cases} l_j, & \text{if } p_i \in \text{dom}(f) \\ \text{“UNK”}, & \text{otherwise} \end{cases}$$

In this function, l_j represents the letter sequence corresponding to the phoneme p_i, and "UNK" is used for phonemes without direct English equivalents.

Common phoneme sequences, such as those in words like *TRUNK* (/tɹʌŋk/), are explicitly accounted for in the mapping. The "UNK" token is only assigned when a phoneme has no plausible English representation or when a reasonable approximation does not exist. This ensures that frequently occurring phonemes are mapped accurately, while only rare or non-standard phonemes fall back to the unknown category.

Vectorisation of Phonetic Representations

Once the phonemes are mapped, each phonetic variant is transformed into a numerical vector for further processing.

N-Gram Feature Matrix

We construct an *n*-gram frequency matrix where **rows** represent words and **columns** correspond to *n*-gram features. The matrix in Table 14.1 captures the frequency of each *n*-gram within the word "hello".

Word	he	el	ll	lo	hl	ell
hello	1	1	1	1	0	1
halo	1	0	0	1	0	0

Table 14.1: *N*-Gram Frequency Matrix for Sample Word "hello"

For a given word w, its feature vector $\mathbf{x}_w$ is

$$\mathbf{x}_w = (x_1, x_2, \ldots, x_m) \in \mathbb{R}^m$$

where x_i represents the frequency of the i-th *n*-gram, and m represents the total number of possible bigrams and trigrams.

High-Dimensional Representation

The full dataset results in a high-dimensional space, where words are represented as points in $\mathbb{R}^m$. The *n*-gram feature vectors are constructed using non-negative integer counts, indicating the frequency of *n*-gram occurrences within each word. Although represented in $\mathbb{R}^m$, the feature vectors remain sparse, which means that most entries contain zeros.

This combination of high dimensionality and sparsity introduces additional challenges, as sparse data can lead to inefficiencies in both memory usage and similarity computations. To mitigate these issues, we apply dimensionality reduction techniques to the *n*-gram feature vectors, transforming them into a more compact representation that preserves the key features while improving computational efficiency.

Dimensionality Reduction

To mitigate high dimensionality, we applied Singular Value Decomposition (SVD) and Principal Component Analysis (PCA). By reducing the dimensionality of the *n*-gram feature vectors, we make them more manageable and highlight the most important features.

Principal Component Analysis (PCA)

PCA transforms the data into a new coordinate system where the greatest variances are found along the first coordinates, known as the principal components. Given an input data matrix $\mathbf{X}$,

where $\mathbf{X} \in \mathbb{R}^{n \times m}$ represents a dataset with n samples (rows) and m features (columns), PCA is performed on the covariance matrix:

$$\mathbf{C} = \frac{1}{n}\mathbf{X}^T\mathbf{X}$$

The eigenvectors of $\mathbf{C}$ define the principal components, and the corresponding eigenvalues indicate the amount of variance captured by each component. The transformation to the new coordinate system is given by:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

where $\mathbf{W} \in \mathbb{R}^{m \times k}$ is the matrix whose columns are the top-k eigenvectors of $\mathbf{C}$, corresponding to the largest eigenvalues. The resulting matrix $\mathbf{Z} \in \mathbb{R}^{n \times k}$ represents the transformed data in the reduced k-dimensional space.

Singular Value Decomposition (SVD)

Singular Value Decomposition (SVD) factorises a given matrix into three other matrices, capturing the most significant underlying structures. Unlike PCA, which relies on the covariance matrix, SVD operates directly on the original data and is particularly useful for sparse matrices and text data. Given the n-gram matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$, where n represents the number of samples (words) and m represents the number of features (*n*-grams), we decompose $\mathbf{X}$ as:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

where $\mathbf{U} \in \mathbb{R}^{n \times n}$ contains the left singular vectors, representing word relationships, $\mathbf{\Sigma} \in \mathbb{R}^{n \times m}$ is a diagonal matrix of singular values, indicating the importance of each dimension, and $\mathbf{V}^T \in \mathbb{R}^{m \times m}$ contains the right singular vectors, representing *n*-gram relationships.

To reduce dimensionality while preserving essential structure, we approximate $\mathbf{X}$ using only the top-k singular values:

$$\mathbf{X}_k \approx \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$$

where $\mathbf{U}_k \in \mathbb{R}^{n \times k}$ contains the top-k left singular vectors, $\mathbf{\Sigma}_k \in \mathbb{R}^{k \times k}$ is the truncated diagonal matrix of the largest k singular values, and $\mathbf{V}_k^T \in \mathbb{R}^{k \times m}$ contains the top-k right singular vectors.

This reduced representation retains the most important phonetic structures while significantly reducing computational complexity.

Similarity Measures for Word Prediction

After dimensionality reduction, a new phonetic input is mapped into the reduced feature space, where its phonetic variant is compared to stored word representations. The goal is to retrieve the closest valid word based on phonetic distance.

Distance Metrics

Given a test phoneme input $\mathbf{q}$, similarity is computed against stored word vectors $\mathbf{X}_k$, which are computed from phonetic spellings. The similarity measures determine the most likely intended word.

A **distance metric** $d(\mathbf{q}, \mathbf{x})$ is a function that quantifies how different two phoneme vectors are. Formally, it must satisfy the following properties:

1. *Non-negativity*: $d(\mathbf{q}, \mathbf{x}) \geq 0$, with equality if and only if $\mathbf{q} = \mathbf{x}$.
2. *Symmetry*: $d(\mathbf{q}, \mathbf{x}) = d(\mathbf{x}, \mathbf{q})$.

3. *Triangular inequality*: $d(\mathbf{q}, \mathbf{x}) + d(\mathbf{x}, \mathbf{y}) \geq d(\mathbf{q}, \mathbf{y})$, ensuring that direct distances are never greater than indirect paths.

Since similarity is the inverse of distance, we define similarity measures as transformations of distance functions, such as:

$$s(\mathbf{q}, \mathbf{x}) = \frac{1}{1 + d(\mathbf{q}, \mathbf{x})}$$

or

$$s(\mathbf{q}, \mathbf{x}) = 1 - \frac{d(\mathbf{q}, \mathbf{x})}{\max d}$$

where higher similarity values indicate phonetic closeness. Cosine similarity is an exception, as it is inherently a similarity measure rather than a distance.

In this study, we evaluate phonetic retrieval using Cosine Similarity, Euclidean Distance, and Manhattan Distance, comparing their effectiveness in matching phonetic spellings to intended words.

1. **Cosine Similarity:** Measures the directional alignment of vectors, useful when phonetic representations vary in magnitude but maintain structural consistency:

$$\cos(\theta) = \frac{\mathbf{q} \cdot \mathbf{x}_w}{\|\mathbf{q}\| \|\mathbf{x}_w\|} = \frac{\sum_{i=1}^{n} q_i x_{w_i}}{\sqrt{\sum_{i=1}^{n} q_i^2} \sqrt{\sum_{i=1}^{n} x_{w_i}^2}}$$

   where $\mathbf{q}$ is the test phoneme input vector and $\mathbf{x}_w$ is the stored word vector. This is effective for phonetic mappings where the relative relationships between features matter more than absolute distance.

2. **Euclidean Distance:** Measures the straight-line distance between phonetic vectors in the reduced space:

$$d_E(\mathbf{q}, \mathbf{x}_w) = \|\mathbf{q} - \mathbf{x}_w\|_2$$

   Useful for cases where phonetic misspellings alter multiple phoneme components simultaneously.

3. **Manhattan Distance:** Computes phonetic similarity by summing absolute differences:

$$d_M(\mathbf{q}, \mathbf{x}_w) = \sum_{i=1}^{k} |q_i - x_{w_i}|$$

   Captures phonetic structure by emphasising stepwise differences, making it ideal for cases where phoneme insertions, deletions, or substitutions occur in a structured, sequential manner - similar to navigating a grid where changes occur in discrete steps along individual phoneme dimensions.

Phonetic Spelling Correction and Word Retrieval

To find the best match, we retrieve the closest stored word w^*:

$$w^* = \arg\min_w d(\mathbf{q}, \mathbf{x}_w)$$

where d is the distance metric chosen. The process is as follows.

1. **Phonetic Vectorisation**: Encode the input word as an *n*-gram feature vector.
2. **Dimensionality Reduction**: Project the vector into the SVD/PCA-reduced feature space.
3. **Similarity Computation**: Find the closest stored phoneme representation based on similarity.
4. **Word Retrieval**: Retrieve the best match using SQLite, where phonetic variants are indexed for efficient lookup.

SQLite stores precomputed phoneme variants and their word mappings, allowing fast similarity-based retrieval rather than recomputing vector comparisons for every query.

By combining vector-based phoneme mapping, dimensionality reduction, and efficient similarity search, this approach effectively bridges the gap where traditional rule-based phonetic spelling correction methods fail.

Challenges in the Methodology

This section outlines the key challenges faced during the development of the phonetic spelling correction system and the solutions implemented.

Choosing the right Lexical Dataset

A phonetic lexicon was needed that included phoneme transcriptions to proceed. Several issues arose in finding the right lexicon. Many phonetic lexicons were based on American English, whereas the project focused on Australian English. In addition, different phonetic transcription systems, for example, ARPAbet vs. IPA, where ARPAbet lacks phonemes found in Australian English. An **Australian English lexicon** was selected and preprocessing techniques were applied to normalise the phoneme representation, ensuring spaces between phonemes for correct tokenisation.

Preprocessing Requirements: Formatting and Handling Special Characters

The lexicon contained various formatting inconsistencies. For instance, the phonemes had no spaces between them, making tokenisation difficult. Diacritics (“ ’ ”, “ , ”, and “ : ”) were removed. The entries were cleaned of zero-width joiners and other unicode characters. The solution in implementing these preprocessing steps was to strip stress markers and ensure spaces between phonemes for better tokenisation. The unwanted Unicode characters were removed.

Data Explosion Issue: Exponential growth in data set size

The generation of phoneme variants led to an exponential increase in the size of the data set. Each phoneme sequence had multiple possible letter mappings, resulting in a combinatorial expansion of word spellings. As the number of phonemes per sequence increased, the number of possible spellings grew exponentially, significantly increasing the storage and computational requirements. For instance, if a phoneme sequence contained five (5) phonemes and each phoneme had three (3) possible letter mappings, the number of unique spellings for a single word would be:

$$3^5 = 243$$

This rapid combinatorial growth underscores the need for dimensionality reduction and efficient phonetic encoding techniques to manage the dataset effectively. The solution involved restricting phoneme-letter mappings to a maximum of 10 variants per word, implementing sampling techniques to limit phoneme sequence variations, and queue processing when saving in SQLite to avoid memory overload.

Challenge	Issue	Solution
Finding a lexicon	No standard dataset for **Australian English phonetics**	Selected **Australian lexicon** and applied **IPA normalisation**
Preprocessing phonemes	**No spaces**, **stress markers**, **Unicode issues**	Used **regex-based cleaning**, ensured **space-separated phonemes**
Data Explosion	Phoneme-to-letter mappings **increased dataset size massively**	**Limited mappings**, capped **variant generation**, used **chunked processing**
Phoneme-to-letter mapping	Some phonemes **had no English letter equivalent**	Created **fallback "UNK" mappings**, **logged missing phonemes**
Vectorisation complexity	**Bigram/trigram embeddings** created **high-dimensional vectors**	Applied **PCA/SVD**, optimised **variance retention**
Dimensionality reduction	Needed to balance **information loss vs. vector size**	Used **variance analysis** before and after scaling

Table 14.2: Summary of Challenges and Solutions in Phonetic Spelling Correction

Mapping Issues: Converting Phonemes to Letters

There were many non-singular one-to-one phoneme mappings, meaning that multiple letters could map to the same phoneme. For example, "f", "ph", "gh" all map to IPA [f] and some phonemes had no direct letter equivalent in English. Unknown phonemes appeared that were not in the mapping dictionary. For example, the voiceless velar fricative /x/ as in Scottish English "loch" or German "Bach", has no standard English letter, and for the glottal stop /ʔ/ in Cockney "bo'le" for "bottle", English does not have a letter at all. The solutions involved creating a phoneme-to-letter dictionary with multiple mappings per phoneme. Then, to identify and capture phonemes that had not been converted, introducing a default "UNK" (unknown) mapping to log unhandled phonemes to a separate file (`unhandled_phonemes.log`) for further debugging and mapping. For instance, mapping /x/ → ["kh", "h", "k"] and mapped the glottal stop /ʔ/ → ["", ""]. Additionally, tokenisation methods were implemented to preserve multicharacter phonemes before mapping aided correct phoneme-to-letter transcription.

Vectorisation and Dimensionality Challenges: High-Dimensional Data for Phoneme Embeddings

After mapping phonemes to letters, the next challenge was to vectorise phoneme spellings. The system used bigram and trigram vectorisation, which created a high-dimensional feature space, increasing computational cost. To solve this, PCA (Principal Component Analysis) and SVD (Singular Value Decomposition) were applied to reduce dimensions. Variance retention in scaled versus unscaled data was evaluated to ensure optimised results. Dimensionality was successfully reduced while maintaining the distinctiveness of the word. The Manhattan distance, cosine similarity, and Euclidean distance were used to compare phonetic spellings.

Summary of Key Challenges and Solutions

The challenges in this project influenced the design of the phonetic spelling prediction system, and are summarised in Table 14.2. Addressing these issues required:

- **Selecting the right dataset** (IPA-spaced phonetic lexicon).
- **Implementing strong preprocessing** (cleaning, tokenisation).
- **Optimising computational efficiency** (restricting explosion, reducing dimensions).

- **Testing multiple approaches** (phoneme-letter mapping, vectorisation).

These solutions ensured that the project successfully processed phonetic spellings while maintaining computational feasibility.

14.3 Results

This section presents the impact of pre-scaled and post-scaled dimensionality reduction on phonetic spelling correction. The focus is on variance retention and the effectiveness of similarity measures.

Dimensionality Reduction and Variance Analysis

Dimensionality reduction was applied to retain the essential phonetic structure while reducing computational complexity. PCA and SVD were applied before and after feature scaling, and their effectiveness was analysed based on variance retention.

Variance Per Component

The variance explained per component for PCA and SVD is shown in Figure 14.1.

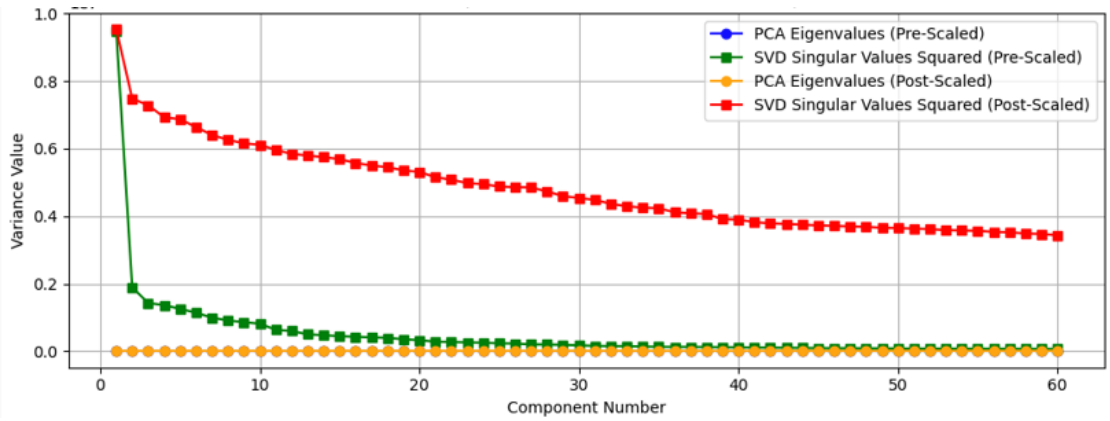


Figure 14.1: Per-component variance for PCA and SVD (pre- and post-scaling). Created using Python [15].

From the figure, it is evident that pre-scaled SVD captures most variance in the first few components, making it efficient for compression. In contrast, post-scaled SVD distributes variance more evenly, avoiding dominance by high-magnitude phonemes.

Cumulative Variance Explained

The cumulative variance for PCA and SVD is visualised in Figure 14.2.

Observations from the figure indicate that pre-scaled SVD reaches 80% variance at approximately 20 components, while post-scaled SVD requires around 45 components. Despite the slower variance accumulation, post-scaled SVD leads to better word retrieval accuracy, as discussed next.

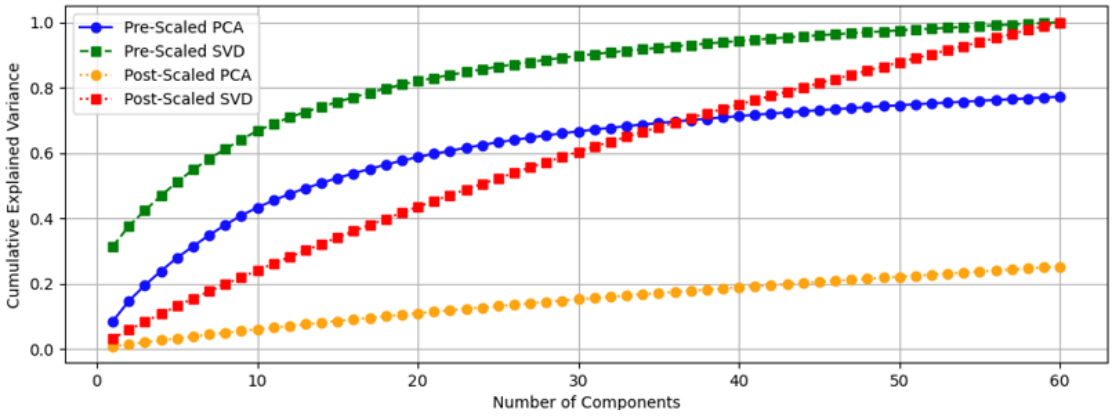


Figure 14.2: Cumulative variance explained for PCA and SVD (pre- and post-scaling). Created using Python [15].

Effectiveness of Similarity Measures in Phonetic Word Prediction

Phonetic similarity retrieval was evaluated using cosine similarity, Euclidean distance, and Manhattan distance. The results were compared using both pre-scaled and post-scaled datasets.

Pre-Scaled SVD Results

Cosine similarity retrieved phoneme variants with some structure but struggled with multi-syllabic words. Euclidean and Manhattan distances disproportionately matched shorter phoneme variants, making them ineffective for accurate phonetic spelling correction. Many matches were single-phoneme outputs, leading to errors in phonetic word prediction (Table 14.3).

Table 14.3: Phonetic Word Prediction (Pre-Scaled SVD)

Input	Method	Closest Match	Distance
ingre	Cosine	surly, swirly, surlier, surreal	≈ 0.55
	Euclidean	e, ea, E	≈ 1.32
	Manhattan	ER, Er, Ir	≈ 7.27
halow	Cosine	revers's, Rickey's, air	≈ 0.56
	Euclidean	A, AA, AR	≈ 0.44
	Manhattan	A, AA, AR	≈ 2.42
pitcha	Cosine	Quebec's, weep, quickie, keep	≈ 0.63
	Euclidean	E, e, ea	≈ 1.18
	Manhattan	A, a, eh	≈ 6.80

Post-Scaled SVD Results

A significant improvement was observed in post-scaled SVD, particularly for cosine similarity. Cosine similarity now retrieves multi-syllabic phoneme variants, indicating better phonetic structure preservation. Euclidean and Manhattan distances still fail to capture the phonetic structure, returning mostly short, truncated phoneme variants. The retrieved words are now semantically and phonologically closer to the intended correct spelling (see Table 14.4).

Table 14.4: Phonetic Word Prediction (Post-Scaled SVD)

Input	Method	Closest Match	Distance
ingre	Cosine	tinkering, ringings, printings	≈ 0.09
	Euclidean	e'en, T, Thai	≈ 4.17
	Manhattan	T, Te, Ty	≈ 22.97
halow	Cosine	Missourian, Rousseau's, pronoun	≈ 0.05
	Euclidean	Ron, rain, rein	≈ 3.83
	Manhattan	T, Te, Thai	≈ 20.36
pitcha	Cosine	chitchat, Rorschach, nature	≈ 0.01
	Euclidean	Ron, Wren, rain	≈ 3.95
	Manhattan	T, Te, Thai	≈ 21.63

Mathematical Interpretation of Results

The results suggest that post-scaled SVD improves the phoneme vector space for word retrieval due to balanced feature scaling. Cosine similarity measures the angular distance, meaning that it performs best when feature magnitudes are normalised. Euclidean and Manhattan distances depend on absolute magnitude, making them unsuitable when phoneme representations vary significantly in scale.

Mathematically, post-scaling transforms feature distributions to have unit variance, ensuring that phonetic distance is based on meaningful phoneme structure rather than raw frequency. Cosine similarity benefits from this transformation because it measures the relative orientation of phoneme vectors rather than their raw magnitude.

14.4 Discussion and Conclusion

Summary of Findings

This study explored the correction of phonetic spelling using dimensionality reduction techniques and similarity metrics. The primary goal was to improve the retrieval of phonetic words using feature transformation methods such as PCA and SVD. The performance of pre-scaled and post-scaled SVD was evaluated based on variance retention and phonetic similarity measures.

Key results demonstrated that:

- Pre-scaled SVD retained a higher cumulative variance in fewer dimensions but led to suboptimal word retrieval due to imbalanced feature scaling.
- Post-scaled SVD, despite requiring more components to reach the same variance threshold, significantly improved phonetic word prediction accuracy.
- Cosine similarity emerged as the most effective metric, outperforming Euclidean and Manhattan distances in retrieving multi-syllabic phoneme variants.

The shift from pre-scaled to post-scaled SVD improved the alignment of phoneme vector representations, leading to more accurate phonetic word retrieval. Although the Euclidean and Manhattan distances struggled with phonetic structure preservation, cosine similarity successfully captured phonological relationships by measuring angular similarity.

Implications and Contributions

The results highlight the importance of feature scaling in phonetic representation. The post-scaling process ensures that phoneme vectors are mapped in a way that preserves their relative phonetic structure, improving similarity-based retrieval. The findings contribute to enhanced phonetic spelling correction methodologies, improved handling of non-standard phoneme spellings, and a scalable approach to processing phonetic input using dimensionality reduction.

This research provides a foundation for further refinement of phonetic spelling correction models, particularly in applications where spelling errors are heavily influenced by phonetic transcription inconsistencies.

Future Work

Although this study demonstrates the effectiveness of post-scaled SVD and cosine similarity, several enhancements can be pursued. Integrating Hidden Markov Models (HMMs) could improve phoneme clustering by capturing sequential dependencies in spoken language. Further, exploring contextual word embeddings, for instance Word2Vec, BERT or transformer-based architectures, could refine phonetic similarity by incorporating broader linguistic patterns.

Expanding the dataset with larger and more diverse phonetic corpora would enhance model robustness and generalisation, ensuring better performance across various dialects and speech patterns. Furthermore, alternative dimensionality reduction techniques, such as t-SNE and UMAP, could be explored to improve the preservation of phonetic structures in a reduced space.

Deploying this methodology in a real-time phonetic spelling correction system and conducting user evaluations would provide insights into its practical effectiveness. Future iterations could incorporate feedback-driven refinements to optimise accuracy and usability in real-world applications.

Conclusion

This study demonstrated that post-scaled SVD, combined with cosine similarity, significantly enhances phonetic spelling correction. While pre-scaled SVD efficiently captures variance, its reliance on unnormalised feature magnitudes reduces retrieval accuracy. In contrast, post-scaling ensures balanced phoneme representations, leading to more precise word predictions.

These findings highlight the crucial role of feature scaling in phoneme vectorisation and confirm that cosine similarity is the most effective metric for phonetic retrieval. By preserving phonetic structure while reducing dimensionality, post-scaled SVD provides a strong foundation for real-world phonetic spell-checking systems.

Future research should refine phoneme embeddings, incorporating probabilistic models, contextual embeddings, or deep learning architectures to further enhance accuracy. Integrating this approach into real-time phonetic correction tools could provide practical support for individuals with phonetic spelling tendencies, improving accessibility and communication.

Context

This article was derived from research completed as part of a Deakin University Scholarship, and 2025 AMSI Summer Research Scholarship. Supervised by Dr. Simon James and Dr. Julien Ugon.

About the Author

Louisa Best, a data science major, is passionate about natural language processing, machine learning, and inclusive design. Her research investigates the role of machine learning in enhancing communication for people with cognitive and linguistic challenges.

Acknowledgements

With sincere thanks to Dr. Simon James (Unit Chair of SIT292) and Dr. Julien Ugon (Unit Chair of SIT192) for their steadfast support, mentorship, and guidance throughout this project. Our thanks also go to the Australian Mathematical Sciences Institute (AMSI) 2024-25 Summer Research Scholarship program for support received, and for offering the opportunity to back inclusive and innovative student research.

References

[1] Ruth Huntley Bahr, Stephanie Lebby, and Louise C Wilkinson. Spelling error analysis of written summaries in an academic register by students with specific learning disabilities: Phonological, orthographic, and morphological influences. *Reading and Writing*, 33(1):121–142, 2020.

[2] Gordon DA Brown and Richard PW Loosemore. Computational approaches to normal and impaired spelling. *Handbook of spelling: Theory, process and intervention*, pages 319–335, 1994.

[3] Rachel Sermier Dessemontet, Anne-Françoise de Chambrier, Catherine Martinet, Natalina Meuli, and Anne-Laure Linder. Effects of a phonics-based intervention on the reading skills of students with intellectual disability. *Research in Developmental Disabilities*, 111:103883, 2021.

[4] Emma Flint, Elliot Ford, Olivia Thomas, Andrew Caines, and Paula Buttery. A text normalisation system for non-standard english words. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 107–115, 2017.

[5] Uta Frith. *Cognitive processes in spelling*. ERIC, 1980.

[6] Nestor Garay-Vitoria and Julio Abascal. Text prediction systems: a survey. *Universal Access in the Information Society*, 4:188–203, 2006.

[7] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[8] Bridget Leutzinger. Phonics and spelling intervention for children with intellectual disabilities. *Public Access Theses, Dissertations, and Student Research from the College of Education and Human Sciences. 412*, 2022.

[9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[10] Ryo Nagata, Hiroya Takamura, and Graham Neubig. Adaptive spelling error correction models for learner english. *Procedia Computer Science*, 112:474–483, 2017.

[11] Steven C Pan, Timothy C Rickard, and Robert A Bjork. Does spelling still matter—and if so, how should it be taught? perspectives from contemporary and historical research. *Educational Psychology Review*, pages 1–30, 2021.

[12] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[13] Lawrence Philips. The double metaphone search algorithm. *C/C++ users journal*, 18(6):38–43, 2000.

[14] Lawrence Phillips. The double metaphone search algorithm, 2000.

[15] Python Core Developers. *Python: A dynamic, open source programming language*. Python Software Foundation, 2022. https://www.python.org/downloads/release/python-3110/.

[16] Arne Rubehn, Jessica Nieder, Robert Forkel, and Johann-Mattis List. Generating feature vectors from phonetic transcriptions in cross-linguistic data formats. *arXiv preprint arXiv:2405.04271*, 2024.

[17] Mariia Ryskina. *Learning Computational Models of Non-Standard Language*. PhD thesis, Carnegie Mellon University, 2022.

[18] Pavel Sofroniev and Çağrı Çöltekin. Phonetic vector representations for sound sequence alignment. In *Proceedings of the Fifteenth Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 111–116, 2018.

[19] Xiuli Tong, Catherine McBride-Chang, Hua Shu, and Anita MY Wong. Morphological awareness, orthographic knowledge, and spelling errors: Keys to understanding early chinese literacy acquisition. *Scientific Studies of Reading*, 13(5):426–452, 2009.

[20] Avraham Treistman, Dror Mughaz, Ariel Stulman, and Amit Dvir. Word embedding dimensionality reduction using dynamic variance thresholding (dyvat). *Expert Systems with Applications*, 208:118157, 2022.

[21] Ha Trinh, Annalu Waller, Keith Vertanen, Per Ola Kristensson, and Vicki L Hanson. Applying prediction techniques to phoneme-based aac systems. In *Proceedings of the Third Workshop on Speech and Language Processing for Assistive Technologies*, pages 19–27, 2012.

[22] Valeriy S Vykhovanets, Jianming Du, and Sergey A Sakulin. An overview of phonetic encoding algorithms. *Automation and Remote Control*, 81:1896–1910, 2020.

[23] Vilém Zouhar, Kalvin Chang, Chenxuan Cui, Nathaniel Carlson, Nathaniel Robinson, Mrinmaya Sachan, and David Mortensen. Pwesuite: Phonetic word embeddings and tasks they facilitate. *arXiv preprint arXiv:2304.02541*, 2023.

https://blogs.deakin.edu.au/maths/yearbook/