Modern Computation

A Unified Approach

S. S. Chandra
Ph.D

March 2020

Draft
Version: 0.5

“Computing Science is no more about computers
than Astronomy is about telescopes.”

Edsger W. Dijkstra (1930-2002)

Contents

List of Figures

List of Tables

To my family ...

CHAPTER 2

Mathematical Preliminaries

"God made the integers, all else is the work of man."

Leopold Kronecker (1823-1891)

Mathematics is the basis of everything that computer science is built upon. Indeed, the first "computer" was invented not as circuits or hardware, but as an idea and abstract concept 30 years before its practical realisation. Sir Alan Turing [1937] built this simple computer, a basic machine now called a Turing machine, out of pure logic and mathematical theorems. He then extended its scope to include stored instructions, introducing computers programs in the process, and the ability to run other machines, a concept now know as universality, in order to solve one of the Hilbert's problems[1]. These concepts are discussed briefly at the end of this chapter in section 2.6 on page 15 and in detail within later chapters.

In this chapter, we will cover the mathematical preliminaries required to understand these results and the use of theorems to construct logical ideas. These will be used as a basis to describe and design computers and their languages just as Turing [1937] did with his machines. To get a better fundamental understanding of the concepts, let us begin with most basic of all mathematical concepts, numbers.

2.1 Numbers

Counting is a fundamental concept we learn in our early childhood. Yet some of the most amazing results in science and some of the most basic questions remain unanswered about the numbers used for counting, namely the integers

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots \tag{2.1}$$

[1]A set of 23 important mathematical problems proposed by the renowned mathematician David Hilbert, the solutions of which became the defining achievements of twentieth century mathematics.

2.1.1 The Natural Numbers

We can create a collection of all possible integers and refer to them as a single entity called a set. This set of all possible positive integers is referred to as the natural numbers $\mathbb{N}$

$$\mathbb{N} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots\} \tag{2.2}$$

Then a number of objects, say apples on a tree, would be a number in this set. An orchid, which would be a collection of apple trees each having a number of apples, would be a subset of this set. For example, an apple tree could have 5 apples and an orchid $\{\, A \,\}$ could have 4 trees with

$$\{\, A \,\} = \{2, 3, 5, 6\} \tag{2.3}$$

Thus, $\{\, A \,\}$ is a subset of $\mathbb{N}$. For more compact notation, we can write the symbol $\in$ to represent if an element is part of a set. For example, we can write $2 \in \mathbb{N}$, since 2 is a natural number, but also $2 \in \{\, A \,\}$.

We can draw the set $\mathbb{N}$ as distances from the origin (i.e. zero) as shown in figure 2.1. In theory, we could count forever, so this number line representation of $\mathbb{N}$ goes on forever.

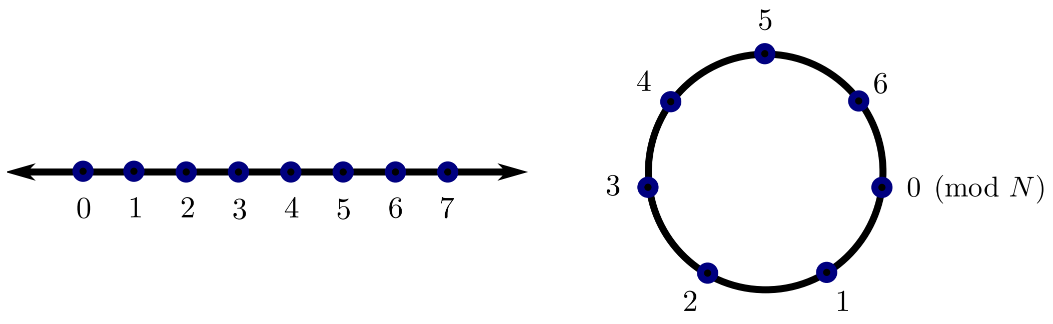


Figure 2.1: Two different ways to represent the natural numbers $\mathbb{N}$. The number line (left) and the finite circle (right).

Because of this, we say that the set of natural numbers $\mathbb{N}$ has an infinite number of elements.

2.1.2 Operations on Numbers

You may ask, what else can we do with this set, and indeed the number line representation, that is useful? Well, we can represent the process of counting by adding a unit repeatedly. In fact, we can construct every other number in $\mathbb{N}$ from zero just by repeatedly adding one. This is easily seen by observing that we 'hop' from one point on the number line to the one immediately on the right of this point until we stop counting. If we continue forever, then we get our number line of figure 2.1. In later chapters, we will use this simple, yet mundane operation to construct an entire universal computational system!

What if we use another number other than one? Let's say we use the number two. We would then obtain the set of even numbers, but we may define a more convenient operation than repeatedly adding a chosen number x called multiplication. If we have a number b and we wish to add it a times, then we write $c = a \cdot b$ or $c = ab$ for short. We can then create a grid representation of the operation such as

$$\begin{array}{ccc} c & c & c \\ c & c & c \end{array}$$

for $a = 2$ and $b = 3$. Then the number of c elements are evident from how many times it appears in the grid by relating the concept to how the area of a square is computed. The number c is referred to as the product of a and b, where the numbers a and b are called factors of c.

This poses two very important questions:

1. Can we create the number line (as we did for the addition of the unit) using products and their factors?

2. How can we undo the product of two numbers?

The later is the basis of all financial transactions because this property is an integral part of most secure encryption schemes. The former will lead us into the realm of the longest unsolved problem in all science - the distribution of prime numbers.

2.1.3 Prime Numbers

Let us begin by trying to create the number line of figure 2.1 on the facing page using the multiplication of integers. Starting with the set containing the number two as an element of our set $\{P\}$, since multiplication involving zero and one is trivial, gives us all the numbers that involve repeatedly multiplying two to itself.

We can define a more convenient operation for repeated multiplication as exponentiation or powers. The power of b raised to a power a can be defined as multiplying b a total of a times or simply written as b^a. Thus, making two an element of our set allows one to construct all powers of two written as 2^a, such as 4 and 8 etc. Our set $\{P\} = \{2\}$ now gives our resulting set of numbers

$$\{N\} = \{2, 4, 8, 16, 32, \ldots\}. \tag{2.4}$$

We have far from succeeded, since the next number 3 is missing, including all of its powers. By making $\{P\} = \{2, 3\}$ improves our resulting set $\{N\}$, since it contains not only powers of 3, but also numbers whose factors involve 2 and 3

$$\{N\} = \{2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, \ldots\}. \tag{2.5}$$

The next number missing from $\{N\}$ is 5 and by making $\{P\} = \{2, 3, 5\}$ improves our number set even further

$$\{N\} = \{2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, 32, \ldots\}. \tag{2.6}$$

Our set $\{N\}$ is slowly becoming complete with powers of 5 and all numbers with factors 2, 3 and 5 are joined to the number set. The same process can be repeated by making the missing numbers 7 and 11 so that $\{P\} = \{2, 3, 5, 7, 11\}$. However, the set $\{N\}$ remains incomplete with the numbers 13 and 17 missing despite filling in more missing numbers. We can continue this process, but we note that the set $\{P\}$ has some interesting properties.

The set $\{P\}_n$ is the set of prime numbers up to the integer n. These numbers have no factor other than itself and unity, while numbers with multiple factors referred to as composite. In a way, prime numbers, or simply 'primes', are the multiplicative building blocks of the natural numbers $\mathbb{N}$. The process described above is known as the Sieve of Eratosthenes, where the primes are sieved out from other numbers as those that do not have factors other than themselves. If you continue the process till all primes are found for $n = 32$, then

$$\{P\}_{32} = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31\}\,. \tag{2.7}$$

There is seemingly no pattern to the primes, and indeed no explicit pattern or formula for producing has been found. Given a number n, it is also difficult to predict how many primes there will be less than n. Significant progress has been made in determining an approximation for the number of primes less than n and is known as the Prime Number Theorem. Predicting the distribution of prime numbers is the Holy Grail of mathematics for which there is a million dollar prize [2].

But what about undoing a multiplication? Given numbers a, b and $c = ab$, how can be solve for say b? We need the concept of integer divisibility.

2.2 Divisibility

We introduce a new operation called division to initially undo the operation of multiplication, where c divide by a to re-obtain b would be written as

$$b = {}^{c}\!/_{a}. \tag{2.8}$$

Clearly, something strange happens in equation (2.8) when $a > c$. We no longer have an integer but a fraction or a number less than unity. Thus, if we want a number system that is self contained, i.e. a system where no new mysterious numbers appear without warning, we either need to expand our set of numbers or define a proper way of dividing only integers. In the next section, we define new number sets until we have a complete system, and in section 2.2.3 on page 8 we define a system where integer division is always possible under some constraints.

2.2.1 Rational Numbers

Let us begin making the system of $\{N\}$ more complete by adding the numbers ${}^{a}\!/_{b}$ for all the possible values of a and b to the set. This creates the new set of numbers called the rational numbers $\mathbb{Q}$ or simply rationals. It is clear that the set $\mathbb{N}$ is a subset of $\mathbb{Q}$, because $a \in \mathbb{N}$ and $b = 1$ for all elements of $\mathbb{N}$. The rationals are usually easy to spot when using decimal representation as those numbers that either terminate or repeat a sequence of digits after the decimal point.

For example, ${}^{1}\!/_{2} = 0.5$ and ${}^{1}\!/_{3} = 0.\bar{3}$, where the bar represents the digits that repeat. Also note that we can always 'convert' a rational number $c = {}^{a}\!/_{b}$ to an integer by multiplying by b, so that $cb = a$ by construction. These two reasons are why the integers and rationals are

[2] A millennium problem prize offered by the Clay Mathematics Institute.

sometimes used inter-changeably and why integers are called rationals in some mathematical texts, particularly when defining variables within equations.

The ancient Greek mathematicians thought that these numbers were the perfect numbers and that all of mathematics could be done using only integers, as our quote from Kronecker at the beginning of the chapter suggests. However, a simple construction involving triangles spoiled their perfect number system. As a side note though, another approach involving integers developed long after the ancient Greeks can indeed create a perfect number system that can even represent floating point numbers exactly. This system is known as the p-adic numbers, where p is prime, but its discussion is outside the scope of this book.

2.2.2 Irrational Numbers

Consider the simple act of constructing a right-angled triangle whose two sides are of length one. What is the length of the third side? This brings us to one of the most fundamental of mathematical results, Pythagorean theorem. Although we will discuss what a theorem actually means later in section 2.5, for now imagine that you have an equation that always holds true, namely that for *any* given right-angled triangle with sides a, b and c with c being the longest side or hypotenuse, then

$$c^2 = a^2 + b^2. \tag{2.9}$$

This is shown in figure 2.2(a). The simple construction of a right-angled triangle with unit

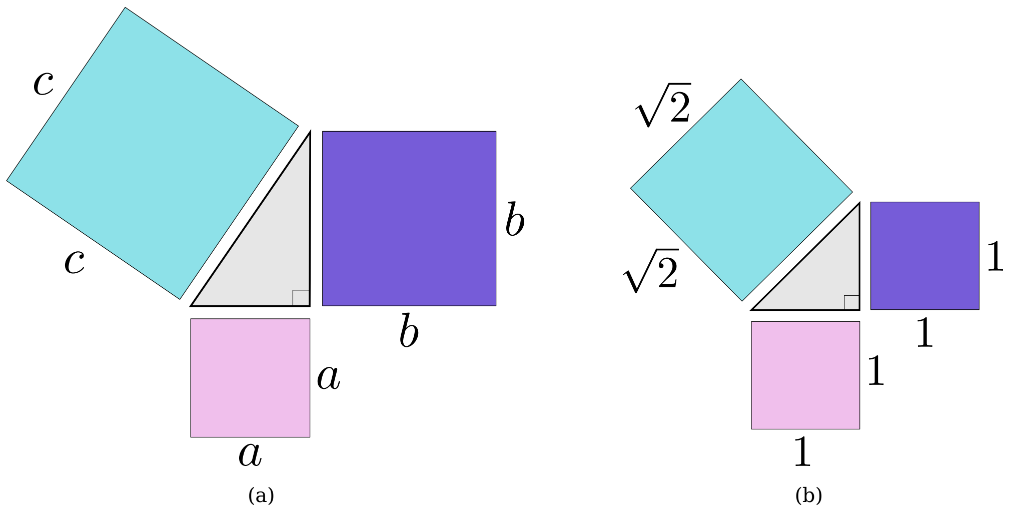


Figure 2.2: The Pythagorean theorem (a) for any right-angled triangle. The construction of a unit sided right-angled triangle (b) leading to the hypotenuse of $\sqrt{2}$.

lengths shown in figure 2.2(b) immediately causes a problem because $\sqrt{2}$ cannot be represented as a ratio of integers. In terms of decimal representation of $\sqrt{2}$, the digits neither terminate nor have a fixed repeating sequence throughout. In fact, the sequence of digits encountered is unique to $\sqrt{2}$. The ancient Greek referred to these numbers as irrational, because it did not fit into their vision for a perfect number system.

Placing the irrational numbers $\sqrt{2}$, $\sqrt{3}$ etc. together with the rationals $\mathbb{Q}$ creates the real number system $\mathbb{R}$. We finally have a system where we can add, subtract, multiple and divide without encountering any elements that is outside the realm of $\mathbb{R}$. This type of system, one that is 'closed' under all these operations is called a number field, or simply a field. Using a field in anything we do mathematically ensures that all arithmetic computations can be done, while not creating anything unexpected. However, many consider this to be an inelegant solution to the problem of constructing a field. In the next section, we construct a number field with only integers!

2.2.3 Congruences

Let us begin by using the number line representation of $\mathbb{N}$, and define some additional operators to help represent integer division. When an integer b divides a, denoted as $b \mid a$, then the distance from the origin a can be divided into b equal parts (see part (a) of figure 2.3). When

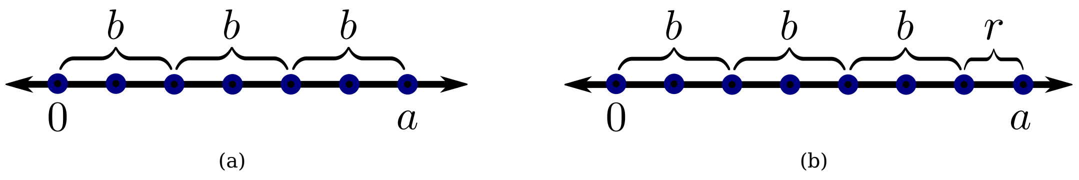


Figure 2.3: Illustration of integer division. (a) shows exact integer division. The integer a is a multiple of integer b so that $a = b \cdot c$. (b) shows integer division with a remainder r. The integer a has to be expressed as equation (2.10).

b does not divide a, denoted as $b \nmid a$, the nearest multiple q being the quotient of b is found smaller than a so that

$$a = b \cdot q + r. \tag{2.10}$$

The remainder r is the extra distance left after division (see part (b) of figure 2.3). Note that equation (2.10) is an equation of a line whose values are restricted to the realm of integers. Such equations are called Diophantine equations, named after the work of Diophantus [100] who first studied them.

In equation (2.10), one may ask two separate, but ultimately equivalent questions. What property will the numbers a and b always share? What is the result of repeatedly applying equation (2.10) to any of the remainders? The answer to both questions is the greatest common divisor (GCD), also called the highest common factor. The GCD of two numbers a and b, which will be denoted as $\gcd(a, b)$, can be found as the following using Euclid's algorithm [Euclid, 300BCE]. Consider the rectangle $a \times b$ as shown in part (a) of figure 2.4 on the facing page. Tile as many whole $b \times b$ squares into this rectangle. Next, tile the remaining $r_0 \times b$ rectangle into as many $r_0 \times r_0$ squares as possible. Repeat this using a $r_1 \times r_1$ square within the remaining $r_0 \times r_1$ square and so on. When the remaining rectangle is tiled exactly by a square, the length of this last square is the GCD of the numbers. This GCD square therefore, tiles the entire initial $a \times b$ rectangle. The tiling process is shown in part (b) of figure 2.4 on the next page.

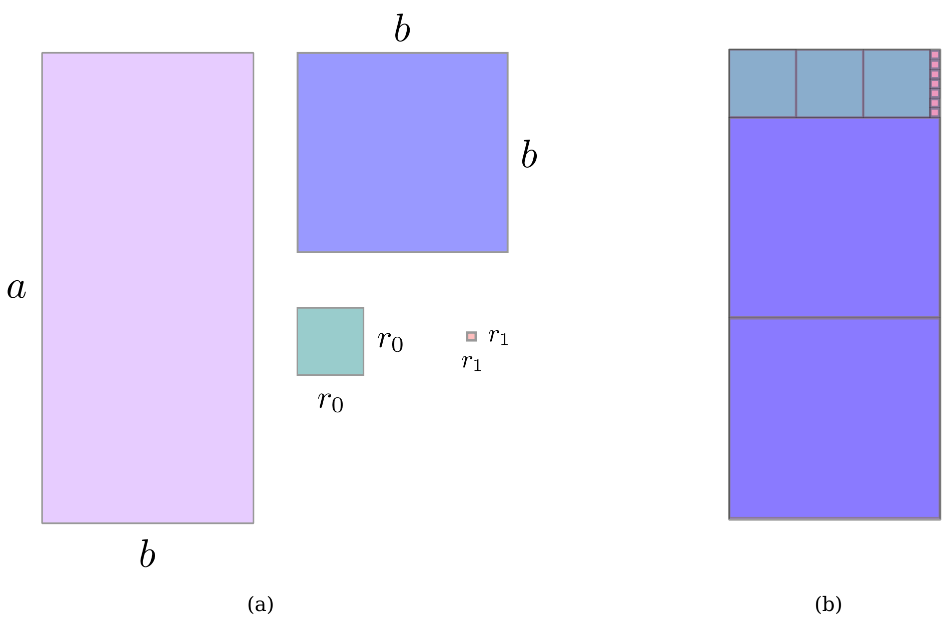


Figure 2.4: Euclid's Algorithm. (a) shows the rectangles used to tile the main rectangle $a \times b$. (b) shows the tiling of the main rectangle $a \times b$.

Gauss [1801, see Articles 1 & 2] introduced another way to view the remainder via the notion of a congruence and congruent integers. Then the integer a is congruent to integer b modulo M when $M \mid a - b$ (see figure 2.5). Alternatively, if we view the number line as a

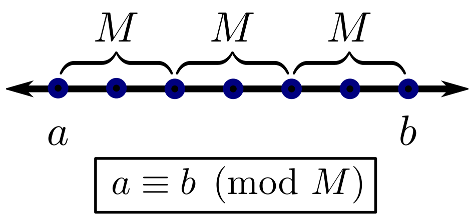


Figure 2.5: The congruence of integers. Integer a is congruent to integer b when the distance between them is a multiple of M.

circle with M points as shown in figure 2.1 on page 4, then the remainder is only the extra bit required to move around the circle ignoring the repeatedly cycles around the circle akin to clock arithmetic. It is analogous to 2 pm on a Thursday looking the same on a clock as 2 am or pm on a Friday.

We can then write linear congruences as

$$a \cdot m \equiv b \pmod{M}. \tag{2.11}$$

If M is prime, then $a \cdot m$ will go through all possible values $\{0, \dots, M-1\}$ uniquely in some order when a goes through $\{0, \dots, M-1\}$ since the $\gcd(m, M) = 1$ always for a fixed $m \in \mathbb{N}$ (see figure 2.6 on the following page). When $\gcd(m, M) = 1$, division by m is possible within

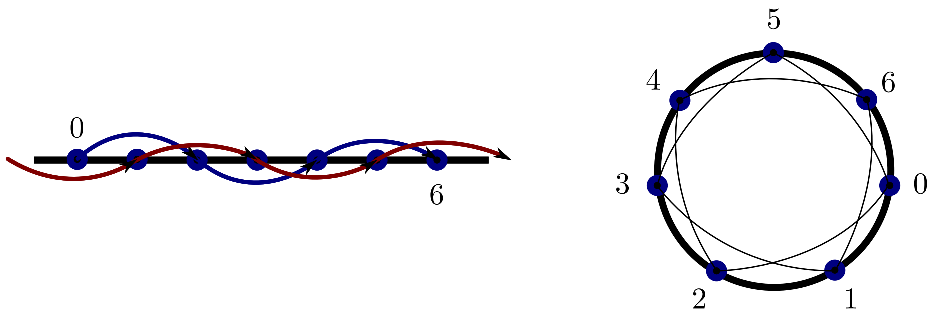


Figure 2.6: Linear congruence equation (2.11) when M is prime, shown as a number line (left) and its equivalent finite circle (right). The example shows $M = 7$ and $m = 2$, where equation (2.11) cycles through all possible values called residue classes $0, \ldots, M-1$ uniquely in some order when a goes through $0, \ldots, M-1$.

equation (2.11). This is done via a multiplicative inverse, so that one can determine a if only b is known, according to

$$a \equiv b \cdot m^{-1} \pmod M. \tag{2.12}$$

The act of division is replaced by multiplying b by the multiplicative inverse of m. We can find this inverse for any given modulus M and integer m via an extended version of Euclid's algorithm. This allows equation (2.11) to be a field, i.e. addition, subtraction, multiplication and division can be computed without leaving the set $\mathbb{N}$.

Congruences and modulo properties of composite numbers forms the basis of key-based encryption schemes such as RSA used by financial institutions around the world. The circle representation of figure 2.1 on page 4 is the how the encryption can be undone. A modulus M is found that is a product of two very large prime numbers, therefore a composite number, that is known only to you. Two powers e, d modulo M are found that act as public and private keys. Exponentiation by the public key e results in an encryption of the character represented as a number. This encryption is a point on the finite circle, albeit one with very many points. The decryption is computed via the exponentiation with the private key d that effectively traverses the finite circle 'back' to the original starting point on the circle, analogous to a multiplicative inverse but for exponentiation.

As long as the modulus M is large and composite, it is very difficult to find the private key d, therefore making the encryption 'secure'. The term secure is relative here, since with a super-computer one could compute the decryption given enough time. Fortunately (or unfortunately depending on your point of view) it would take the lifetime of the universe to crack it with the current computational power. However, a new paradigm of computation called quantum computation could change that and this will be covered in later chapters.

Now that we have defined numbers and number systems, we need to be able to compute and do things with them. This does not always involve only arithmetic but concepts of computation and structures as we shall see in the next section.

2.3 Functions

Given a number system, such as any of the ones defined previously $\mathbb{N}$, $\mathbb{Q}$ etc., we can define functions over numbers. A function is an object that produces an output based on the input provided. A function f can also be thought of a mapping from a to b when written as $f(a) = b$. The set of inputs to a function are also referred to as the domain D of the function and the set of outputs as its range R. We can express this as

$$f : D \to R. \tag{2.13}$$

We can also think of a function as a black box that gives us a transformation of the input to an output.

We have already seen examples of functions in previous sections, such as the GCD of two numbers a and b written as $\gcd(a, b)$. It returns the number that is the greatest common divisor of a and b. When the GCD of two numbers is unity ($\gcd(a, b) = 1$), the numbers are defined as being coprime to each other. Euler [1763] described the properties of a function which defines the number of coprime integers less than a number n. This function is denoted as $\phi(n)$ and is known as Euler's Totient or phi function with $\phi(n) : \mathbb{N} \to \mathbb{N}$. For example, a prime number p is a number where $\phi(p) = p - 1$, i.e the $\gcd(j, p) = 1$ for all $0 \leqslant j < p$ and $j \in \mathbb{N}$.

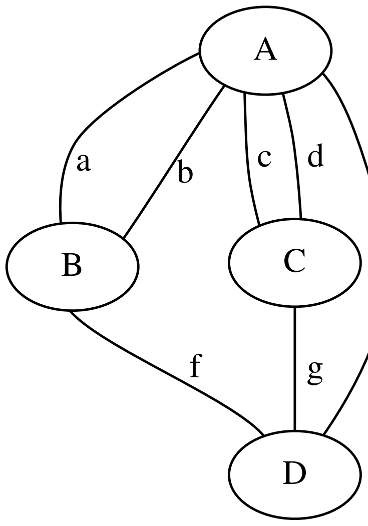
Functions do not have to take exactly one or two inputs however. It is common to use the notation $\{A\}_0 \times \{A\}_1 \times \{A\}_2 \times \ldots \times \{A\}_k$ of sets to represent a tuple $(a_0, a_1, a_2, \ldots, a_k)$ of inputs or arguments to a function. An example of tuples is how we use the Cartesian coordinates in two dimensional (2D) or 3D (x, y, z) is formed from $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$, which we use to plot arithmetic functions.

The concept of a function may seem simple enough and maybe even uninteresting, but in later chapters we will use it to construct machines capable of any possible computation you can devise! In the next section, we show a mathematical way to represent functions visually or more generally the structure of any arbitrary concept.

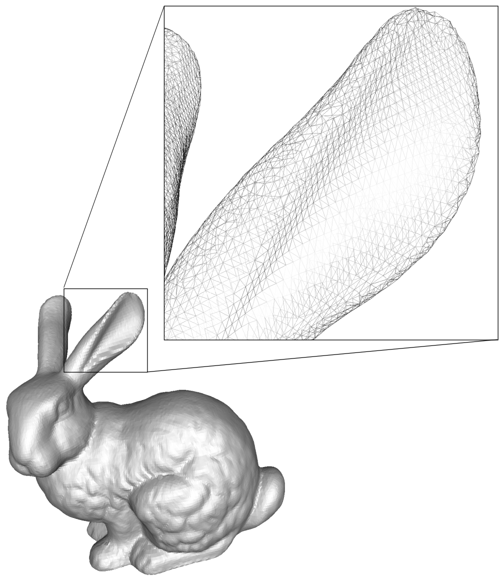
2.4 Graphs

A graph is a useful mathematical construct to help visualise and analyse connections between objects or concepts and represent the inter-dependencies among them. Although they can be used to represent functions or formulas, they are powerful enough to warrant their own field in mathematics called graph theory.

A graph is a structure made out of nodes or vertices V and edges E. Much like drawing a polygon such as a triangle, the vertices usually represent the 'corners' of the graph and the edges connect these vertices together that encode the relationship between the nodes depending on what the graph represents. Indeed, this polygon analogy is used as an efficient way to represent 3D models as 2D surfaces in computer graphics. Figure 2.7 on the following page shows examples of two models representing two different structures. Models in this form are represented as vertices, edges and cells with the latter required if the model is solid and not a wireframe.



(a)



(b)

Figure 2.7: Examples of graphs as (a) the Dollar game and (b) as a 3D model used in computer graphics.

The number of vertices in a graph is called its degree and the edges can be directed or undirected. In a directed graph, all the edges have a direction or point to the next vertex to traverse, which allows for more complex relationships and constraints to be encoded into a graph.

Graphs have many uses other than arithmetic and 3D models. The graph structure is the basis of how auto-merging is done by the version control software called Git. Git tracks content not files like many more traditional version control packages. The result is that Git can automatically merge changes to source files without user intervention, a feature that is critical to the fork and distributed version control workflow of software engineering.

Sophisticated mathematical algorithms are also available on graphs. One such algorithm is known as Dijkstra's shortest path algorithm, which finds the shortest paths between vertices on graphs. For example, if the nodes represent interconnected cities and towns, Dijkstra's algorithm determines the shortest route required to navigate from one city or town to another. Another algorithm is one that finds the minimal energy among nodes to separate nodes on a graph into two parts called graph cuts. We can use this algorithm to create a separation between two structures close together. These features make graphs a useful tool in visualising and analysing structures for a number of applications. We will use graphs to represent machines and their components to analyse how they process incoming input data and what states they occupy.

So far we have used a few results that seem to be very important and we have used the terms 'theorem' or algorithm and we have assumed that they always work. We have even made assumptions about numbers, but how do we know that they are always true? In the next section, we describe what a mathematical proof is and what the consequences are for science and computation.

2.5 Theorems and Proofs

We all know the definition of a fact: A concept or something that is true or proven to be true. A theorem is a mathematical fact, a result that is proven to be true using logic and mathematical results. Unlike facts in everyday life however, a theorem is as close to scientific fact or truth as possible that is beyond any doubt and *always* true. The term 'always' is important here, as the fact is true regardless of the size of numbers or limits and conditions involved. A proof is a series of infallible logical arguments that shows the theorem to be true. Any special conditions required by the fact are provided in the theorem itself and also shown in the proof. Let us examine theorems with a simple example involving even numbers, which are numbers that are multiples of two.

1 **Theorem (Product of Even Numbers)**
The product of any two even numbers is always an even number.

Proof: *Every even number n_i can be represented as $n_i = 2 \cdot k_i$, where $k, i \in \mathbb{N}$. The product m of two even numbers can then be written as*

$$
\begin{aligned}
m &= n_1 * n_2 && (2.14)\\
&= (2 \cdot k_1) \cdot (2 \cdot k_2) && (2.15)\\
&= 2 \cdot (2 \cdot k_1 \cdot k_2) && (2.16)\\
&= 2 \cdot k_3 && (2.17)
\end{aligned}
$$

But the term $2 \cdot k_1 \cdot k_2$ can be substituted as a natural number k_3, so that we get $m = 2 \cdot k_3$, which is also an even number. Since k_1, k_2 and k_3 can be any natural number $\mathbb{N}$, the substitution is always possible resulting in an even number for all and any k_1 and k_2. ◄

The above is an example of an algebraic proof, where if we can equate the left-hand side with the right, our proof is complete. The text following the equations are not really necessary, but for the benefit of the reader. Notice that the theorem applies to any and *all* even numbers regardless of size. This cannot be emphasized enough, there are no counter examples possible and absolute nature and finality of the theorem are indicative of all mathematical proofs. We will use the word 'proof' in certain times in the book and by this we will mean that the result is absolute and final with no doubt.

Not all proofs need be algebraic in nature however, or indeed be possible algebraically. There are many other types of proofs that are available including proof by construction and by induction. A proof by induction assumes you have a starting statement called as basis that can be extended to all possible cases or scenario by an induction. In a proof by construction, logical steps in creating mathematical object, such as a polygon or shape, make the proof self-evident from these steps. More information about these types of proofs can be found in Sipser [2013, Chapter 0]. In the subsequent sections, we will discuss some of the other major ways to prove mathematical results.

2.5.1 Visual Proofs

One of the easiest proofs to understand are those that visually represent the proof as a figure and the theorem is almost self-evident. These types of proofs are the hardest to construct though and generally not utilised very often. An example of such a proof is the proof of the Pythagorean theorem using figures of triangles as shown in figure 2.8. A more common type

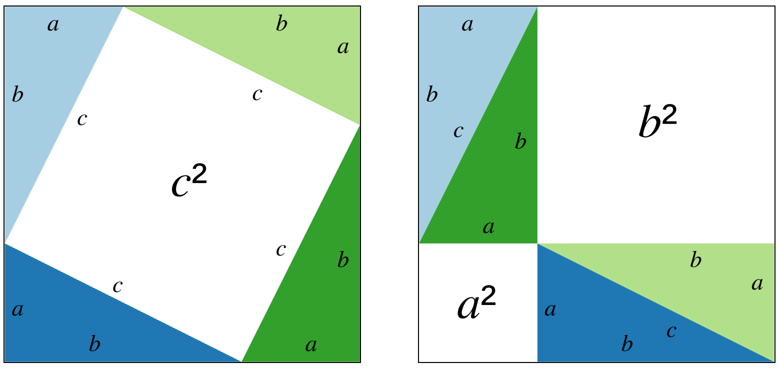


Figure 2.8: A visual proof of the Pythagorean theorem. The total square minus the area of the triangles gives c^2 on the left and a^+b^2 on the right. Original figure by William B. Faulk.

of proof is via the use of creating contradictions from your initial assumption.

2.5.2 Proof by Contradiction

Often we would like a certain assumption to be true and we would like to prove it. It is usually tricky to prove that something is true because either there are too many aspects to prove correct or it does not provide us with anything glaringly obvious to show that the proof is done. What would be nice is if we set up a set of logical statements and it led to an obvious outcome that completes the proof for us.

Proof by contradiction is one of those techniques, whereby we make an assumption, set out a set of logical statements and see if it leads to a obvious erroneous outcome like a contradiction to our initial assumption. A popular way to use this approach is when we assume that the opposite of what we would like to prove is true and if if end up with a contradiction, then the opposite must be true so it proves the statement that we wanted to prove.

For example, in the case of prime numbers, we would hope that there are an infinite number of primes because there are an infinite set of numbers in $\mathbb{N}$. To prove this, we assume that there are in fact a finite number of primes and check to see if we encounter a contradiction, see theorem 2 on the facing page.

2 **Theorem (Infinitude of Primes)**
There are an infinite number of primes in $\mathbb{N}$.

Proof: *We begin by assuming the contrary, there are a finite number of primes p_i, where $i < N$ and N is some large number, i.e. our set $\{P\}$ is finite. Then let us multiply all the known finite primes together to create the product*

$$m = p_0 \times p_1 \times p_2 \times \ldots \times p_N \tag{2.18}$$

This is the largest number we can represent using our known set of primes $\{P\}$. But then we can always add unity, so that $m' = m + 1$. Clearly m' is not a product of the known primes, since it is bigger than m and $p_i \nmid m'$ for $i < N$. But m' is on the number line like any other and therefore must be a product of primes. Thus, there must be another prime number that is not in our set $\{P\}$. We can repeat this process ad infinitum and we arrive at a contradiction. Therefore there must be an infinite number of primes. ◀

This is a well known proof, since a similar proof was known to Euclid [300BCE], but it also relates to an important concept of completeness, which we will discuss in the next section.

2.6 Incompleteness

In previous sections, we have constructed mathematical systems, such as the natural numbers $\mathbb{N}$, where we defined its set of elements and its operations. There were a few rules in this system called axioms that we took for granted about the operations in $\mathbb{N}$, a few of which include axioms such as

1. multiplication and addition are commutative, i.e. $ab = ba$ or $a + b = b + a$,
2. associative law applies, i.e. $c(a + b) = ca + cb$.

These axioms, though seemingly obvious, are important for the system to function properly. In general, any formal mathematical system consists of a set of axioms that define the rules for the system and allow for the creation of new theorems. These axioms govern the way the system behaves and define everything else within it, including the theorems that are possible.

For example, consider plane geometry that we use whenever we construct geometric figures such as circles and triangles on the page. This geometry is actually governed by a set of axioms initially defined by Euclid [300BCE], which is why it is usually referred to as Euclidean geometry. These axioms include how circles, lines and points are defined, and how lines interact, but the exact details are not important here. The fact that a fixed set of axioms is required for this system to function is important for the idea of completeness. Euclid's axioms are sufficient to perform any operation within the geometry, so Euclidean geometry is considered to be complete. Note that defining a system is not always unique however as Klein [1893] found a more general set of axioms that unified both Euclidean and non-Euclidean (i.e. curved) geometries with a single set of axioms.

In an important result that turned mathematics upside down, Gödel [1931] proved as a series of theorems that a system that is complete cannot be consistent and vice versa. We mentioned that a system is complete if the axioms govern all of the possible theorems. A formal system is consistent if all the theorems can be either proved or disproved, i.e. there are no mathematical inconsistent results possible. In other words, the incompleteness theorems of Gödel [1931] show that a consistent formal system has to be incomplete.

To understand why this is the case, let us examine our Theorem 2 on the previous page on the infinitude of the primes. When the set of primes $\{P\}$ is finite, we can only create a finite set of numbers up to the product of all the primes in our set. Likewise, by having a finite set of axioms in our formal system, we can only create a finite set of theorems and therefore there will always be theorems that cannot be proved using these axioms, just like there will be numbers which we cannot create as a product of the primes in our limited set $\{P\}$.

In the following chapters, we will discuss how representing mathematics and formal systems as a 'code' of symbols enacted on by 'machines', can prove the same result and much more leading to the creating of computers in a formal sense. This result by Turing [1937] is the basis of state-based computation and forms the theoretical foundation of most digital computers we use today.

CHAPTER 3

Turing Machines

"The idea behind digital computers may be explained by saying that these machines are intended to carry out any operations which could be done by a human computer."

Alan M. Turing (1912-1954)

Imagine that you could reduce all known computations, anything you know a computer can do and even arithmetic, down to just a few fundamental operations of reading, writing and moving symbols around on pieces of paper and nothing more. It would be simple enough to even solve any computation by hand given enough patience. With such a simple machine, we could not only study computation and prove its properties, but also help design a physical realisation of such a computation machine. Alan Turing [Turing, 1937] introduced just such a machine in order to study the "Entscheidungsproblem" (or the decision problem) and in the process developed a theory of computation. These machines are known as Turing machines in his honour and eventually led to the design the first digital computers by John von Neumann and others.

3.1 Computers

But what are computers? We associate computers with digital devices and electronic hardware because this is the current generation of computing we use today. However, the electronic computer is a fairly recent invention with the Colossus constructed in 1944 at Manchester, England and the ENIAC constructed in 1945 in the United States. Before then, a computer was a specialised human being that computed results for scientists and engineers by hand! In fact, the computations for the Manhattan project that eventually devised the Hydrogen bomb used to end World War II utilised women computers extensively. The design of the workflow for these human computers were sophisticated enough to detect errors and one of the major problems Nobel laureate Richard Feynman had to solve for this project to ensure calculations could be done in time [Feynman, 2005]. We will use this idea of a human computer to establish if and how we can transition to an automatic computing engine (ACE) as Turing envisioned it[1] or a computer as we know it today.

Imagine that you have a hypothetical "computer" that is defined by everything inside a closed off room. This room has two small letterboxes, one for the input and another for the output. From the outside, it is not possible to see the inner workings of the room and the only way to interact with this "computer" is via letters through these letter receptacles.

Let our room initially house a human and that person computes whatever is requested via the input letters. Immediately you would recognise that the "language" used to communicate with the "computer" is very important and has to be predefined in order for the device to understand the input and for us to understand its output. More on this definitions later when we construct our ACEs. Without loss of generality, we may assume that the device is only used for arithmetic calculations, so that a simple computation 2+3 provided to the "computer" would be easily calculated by the person in the room and return the result 5. The person inside the room would use his or her knowledge of arithmetic to compute the result.

Now let us consider providing an instruction booklet to the human that describes the method for all types of supported arithmetic step-by-step, such as addition in our case above. The human is told to only compute the result following this booklet. The human would examine the input and based on its contents, select the appropriate instructions, modify the symbols until a solution is obtained. From this scenario, a two practical points arise:

1. The language and the representation of the inputs now need to match the instructions more formally, since the they must operate on valid known inputs to function properly.

2. It is clear that the human would require working paper and pencil/pen to keep an accounting of which part of the input is currently being processed, where in the booklet one is currently located and the value(s) of the intermediate result.

In other words, our "computer" requires a formal language and working paper or some memory in addition to the instruction booklet.

If we examine our 2+3 example using this booklet approach, we can define a unary representation for numbers and instructions depending on the arithmetic operators supported and cells to write our representation. A unary representation is used in cartoons and movies, where the protagonist commonly counts the number days in captivity via a series if ones. In our example, the input would look like We have chosen to use the '+' symbol, but it could have

1	1	+	1	1	1

been any symbol as long as it corresponds to the instructions. To obtain our sum, observe that we simply need to move all the three 1's after the + symbol one cell to the left, overwriting the '+' symbol and return the result as this will be the unary representation of the answer 5.

1	1	1	1	1	_

[1]We use the term ACE to refer to the concept of an automated computer and loosely on the ACE computer specifications designed initially by Turing.

Thus, the instructions to a human would be along the lines of the following for addition:

1. Start at the first element.
2. Travel right, denoted by the symbol R until the operator symbol is located in the input.
3. If the operation is addition, move all the 1's after the + symbol one cell to the left overwriting the + symbol.
4. Return the result.

Note that the act of moving symbols across the cells will involve the human using the working paper to keep track of the intermediate results. The act of executing the instructions above, the human will need to keep track of the state of the computation so far, i.e. moving cells, returning the result etc. Finally note that the human will need to look at and place their attention or awareness on one cell at a time to do the moving of cells.

It is important to point out that the instructions we are talking about is what we call a computer program (or instruction sets) in modern terminology. This program represents an algorithm or a collection of algorithms that can solve the problem for which we seek a solution. Also, not only is the input is a series of symbols, but the instructions above can also be represented as symbols resulting in the computation being reduced to nothing but "codes" and a manipulation of this code. Therefore, every instruction in this program or algorithm can be seen as a decision that is represented as a code. Turing envisioned all mathematics in this way, so that any problem or theorem is a unique code that can be manipulated to prove important results, such as the halting problem that will be discussed later.

We are nearly done creating our ACE. In our "computer" above, the human is not doing very much other than the mechanical tasks required by the detailed instructions provided. Thus, we now we replace the human with an (electro)mechanical device that will keep track of the awareness of the computation and one that can read/write on cells just like the human did. This device is very simple to construct as it simply read/writes cells and only follows the instructions provided as codes. For convenience of notation and description, we shall call this device the read/write head and the group of all cells as the tape or array. Though this is terminology that dates back to Turing's time of ticker tape machines, we can associate it with sequential access memory such as hard disks or magnetic tape drives. We may also refer to it as an array for a more modern interpretation because it is possible to represent one dimensional (1D), 2D, 3D and even n-dimensional arrays via 1D C-style arrays quite easily and therefore possible to take this representation into account in the instructions.

Now we have completed our ACE. By construction, it is the simplest computational machine capable of computing everything. It was Turing's work that first introduced this concept of the stored computer program in such a formal way. It also makes the bold assertion that *any* computation that a human can do can be reduced down to algorithms! We shall discuss these issues at the end of the chapter. For now, we will examine how Turing used these machines to prove that there exist programs that may never halt or arrive at a fixed solution.

3.2 The Halting Problem

We noted in the previous section that our Turing machine uses the concept of a stored program to compute. The machine repeatedly executes the instructions in this program until a solution is reached. But what if a solution is never reached? How do we know that a solution is even possible for a given computation? Does there always exist a solution to every problem? In other words using the terminology of Turing machines, can we determine if a problem or computation will ever complete or halt? Just as for any other problem, we have already shown that we can create a Turing machine to compute a solution of this problem for us.

Let us create a Turing machine that computes whether or not a given problem is guaranteed to halt or not. This machine has an important property however, since its input is another Turing machine. These types of general machines are called universal Turing machines and Turing [1937] introduced them to prove the halting problem. The input of this machine is a Turing machine and the output is whether or not the computation will halt or not. We note that this machine T_H will halt if and only if the input machine halts, producing the output True. If the machine does not halt, we will need to let it loop until it is finished. Using the same argument above, we can run this universal Turing machine T_H inside another Turing machine T_D to detect and decide that the original Turing machine, the one input to T_H, is looping and tell us that it will not halt by returning True if T_H is looping.

We may test this setup with any normal Turing machine and it seems all is fine and well. That is until you realise that the input to T_H must be all Turing machines for our assertion that every program will halt, including our Turing machine T_D. Setting T_D as the input to itself causes a contradiction to occur. When T_D is supposed to halt it will loop, but at the same time when T_D will loop it is supposed to halt. Clearly, it cannot be in both states at the same time. This self-referential idea causes many problems in mathematical logic, including the question introduced by Bertrand Russell asking whether $\{R\}$, the set of all sets that do not contain themselves, contains $\{R\}$ itself? If $\{R\}$ is a set of all sets that that does not contain itself, it must in $\{R\}$, which is a contradiction.

The consequence of this theorem is that there will be particular computations, where it will be impossible to know if solutions exist and that programs will halt. This extends the incompleteness theorems Gödel [1931] to decidability, so that not only are there potentially theorems that cannot be proven but there will be problems that are not decidable either.

Till now, we have discussed Turing machines and the ideas behind their use in proving theorems. In the remaining sections of this chapter, we will construct these machines and design programs for them. In order to do this, we first investigate simpler machines that are possible, which do not compute everything, but are still useful for very simple specialised devices. These machines are devices that do not have any memory.

3.3 Finite State Machines

Consider having to design a device or system that only needs to decide whether to accept an input or reject it. Equivalently, you would like a system that produces a yes/on or no/off decision. In such a scenario, a machine that has a finite number of states and one that processes the input one element at a time is appropriate. This type of device that has no memory (in the sense of a tape or array) and is called a Finite State Machine (FSM).

3.3.1 State-based Computation

One of the simplest FSMs is the well known on/off switch that is capable of turning things on or off. We can represent a FSM as a state diagram using a directed graph that represents transitions between states as edges such as that of figure 3.1. In this diagram, we can keep

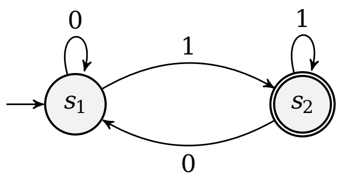


Figure 3.1: A graph representation of the FSM that is a simple on/off switch.

track of how the computation progresses by following the arrows in the diagram depending on the input elements encountered one at a time. We start at an initial state s_1 and only end successfully if we end up in the accept state s_2, indicated by the double circle. For example, a string $w = 1101$, we can express the resulting computation as a series of state transitions:

1. Start computation in state s_1
2. Read first character in w, transition from s_1 to s_2 because it is a 1
3. Read next character in w, transition from s_2 to s_2 as it is 1
4. Read next character in w, transition from s_2 to s_1 as it is 0
5. Read next character in w, transition from s_1 to s_2 as it is 1
6. Reading is complete and we are in the accept state s_2
7. Accept the computation

When in the accept state, we have completed our computation successfully by processing the entire input string and therefore stop the machine. If we however end up in any other state, we reject the computation as a failure. Writing the above list of transitions as sentences is cumbersome, so there are more efficient representations such as the state diagram of figure 3.1 or as a transition table below.

	0	1
s_1	s_1	s_2
s_2	s_1	s_2

Table 3.1: Transition table for the FSM in figure 3.1.

We may also write the transitions symbolically also using the notation δ to represent a transition function, where he δ symbol is usually used to represent change in many areas of mathematics. The arguments to $\delta(s, a)$ would be the character being processed and the current state and it would return the new state. To summarise, the FSM in diagram 3.1 on the preceding page as transitions are

$$\delta(s_1, 0) = s_1, \qquad \delta(s_1, 1) = s_2 \tag{3.1}$$
$$\delta(s_2, 0) = s_1, \qquad \delta(s_2, 1) = s_2. \tag{3.2}$$

Although the transition table above defines how the computation will progresses for the string w, we should ask for what inputs are accepted by the machine in general and which ones are rejected. In this machine, it is clear that any binary string ending in 1 will be accepted, but this is not always so easy to determine for more complex machines. We will describe the significance of strings that are accepted by FSMs in the next section.

Formally, each FSM firstly consists of pre-defined set of states $\{S\} = \{s_1, s_2, \ldots, s_n\}$ for number of states n, which is used during the computation. This includes three important states that all FSMs have: an initial state s_0, an accept state s_{yes} and a reject state s_{no}. Secondly, we need the set of transitions or instructions δ for the FSM that defines what state the machine should go into when in a particular state and it reads a certain character from the input string. In notation, we mean that $\delta : \{S\} \times \{A\} \to \{S\}$, i.e. state paired with character to another state. Lastly, we need to formally define the set of strings acceptable by FSMs and we discuss this in the next section.

3.3.2 Regular Languages

As we have ascertained in the previous section, the language of any machine input needs to be carefully defined to match the instructions and the FSM is no exception. First, we need to define the symbols that we will use for the input and the computation to apply on. The set of all possible unique symbols or characters is called the alphabet $\{A\}$. The input is then a string of symbols concatenated together from the alphabet. For example, a binary FSM would have the alphabet $\{A\} = \{0, 1\}$ and one of the possible strings as $w = 1101$ as we have seen before. We can then define the set of all finite strings of $\{A\}$ as $\{A\}^*$. The set $\{A\}^*$ is an infinite set of all possible strings via the alphabet $\{A\}$, each of finite length. An example of $\{A\}^*$ for the alphabet $\{A\} = 0, 1$ would be

$$\{A\}^* = \{\epsilon, 0, 00, 1, 11, 01, \ldots\}, \tag{3.3}$$

where ϵ indicates the empty string, i.e. a string of length zero.

In order to make our notation more compact, it is possible to generalise our transition function to accept strings w not just characters of the alphabet, so that we can write $\delta^*(s, w)$. We can then define the transitions recursively by repeatedly breaking down the string w to the string $w = ua$, where u is a substring of w including all characters except a, which is the last character of w so that $a \in \{A\}$, i.e. one of the unique characters of the alphabet. This gives us the notation $\delta(\delta^*(s, u), a)$. The notation is based around the last character of the string because it will decide whether the machine's final state will be an accept state or not. Formally, we can define the function recursively as

$$\delta^*(s, w) = \begin{cases} s, & \text{if } w = \epsilon \\ \delta(\delta^*(s, u), a), & \text{if } w = ua \text{ for } u \in \{A\}^* \text{ and } a \in \{A\} \end{cases} \tag{3.4}$$

We can then repeat our "breakdown" of the substring u in this way until the empty string remains, in which case do nothing more and return. This is equivalent to following the state diagram until the entire input string is processed and the accept state is reached.

If we define the set of final acceptable states as $\{S\}^{yes}$, then the language of the machine M can then be defined as

$$L(M) = \{w \in \{A\}^* \mid \delta^*(s_1, w) \in \{S\}^{yes}\}. \tag{3.5}$$

Any language which is accepted by a FSM is known as a regular language. Conversely, a FSMs can only recognise a language if it is regular.

We may now put all these aspects together to formally define FSMs. A machine $M = (\{S\}, \{A\}, \delta, s_0, \{S\}^{yes})$ is a FSM if it accepts an input string $w \in \{A\}^*$ of length n from a series of states $r_0, r_1, \ldots, r_n$ in $\{S\}$ such that

1. $r_0 = s_0$, i.e. starting from the initial state,
2. $\delta(r_i, a_i) = r_{i+1}$ for $0 \leqslant i < n-1$ and $w = a_0 a_1 \ldots a_n$,
3. $r_n \in \{S\}^{yes}$, i.e. ending up in an accept state.

In other words, starting from a state r_0, which is the initial state s_0, we process each character of w to the next state up to the last state. The last state must be an accept state for the FSM M to recognise the language L formed from the alphabet $\{A\}$ and transitions δ.

3.3.3 Regular Operations

Having defined regular languages, it is important to consider the types of operations between languages allowed that preserve the regularity of the resulting language. In this text we will explore the result of combining two regular languages, namely their union and intersection. To determine if the result of combining two languages is regular, we may utilise the corresponding FSMs M_1 and M_2 of these languages to process an example input w via both machines simultaneously and establish if a new FSM M_3 is definable from the resulting operation.

3 Theorem (Regular Operations)
The operation of a union of two languages $L_1 \cup L_2$ with the same alphabet $\{A\}$ is also a regular language L_3.

Proof: *Consider the corresponding FSMs M_1 and M_2 of the two languages having the states $\{S\}_1 = \{s_1, s_2\}$ and $\{S\}_2 = \{r_1, r_2\}$. Simulating a new machine M_3 that runs these two machines simultaneously implies that the machine M_3 is in a pair of states from both $\{S\}_1$ and $\{S\}_2$ at the same time at any one given moment. Given that the cardinality of $\{S\}_1$ and $\{S\}_2$ is both 2, the states of M_3 can only have the following four possible pair of states*

$$\{S\}_3 = \{(s_1, r_1), (s_2, r_1), (s_1, r_2), (s_2, r_2)\} \tag{3.6}$$

But the pairs of states in above set $\{S\}_3$ are themselves a unique combination of pairs and can mathematically substituted with different dummy variables q_1, q_2, q_3 and q_4. Since the alphabet of the two languages is the same, the alphabet and set of strings $\{A\}^*$ of M_3 must be the same as well.

We can apply the same logic to the transitions functions of both machines δ_1 and δ_1 to create a new function δ_3 that describes the paired transition of both machines as the input is processed from states q_i to q_j. For example given a character a from input w, we could have $\delta_3((s_1, r_1), a) = (s_1, r_2)$ in our example using $\{S\}_1$ and $\{S\}_2$ above assuming that $\delta_1(s_1, a) = s_1$ and $\delta_2(r_1, a) = r_2$.

Lastly, we may define our start and final states as suggest a combination of M_1 and M_2 states, since we are running both states at once. Thus $\{S\}_3^{yes} = \{S\}_1^{yes} \times \{S\}_2^{yes}$ and likewise for the initial state. Therefore, if we extend our sets $\{S\}_1$ and $\{S\}_2$ to be set of states for any arbitrary FSMs, we can see that we will always be able to define a new machine M_3 that fulfils the criteria for a FSM and hence is also a regular language. ◀

A similar argument to theorem 3 on the previous page can be made for the intersection of languages. In the next section, we explore how parallel and branched processing can be done via FSMs.

3.3.4 Non-deterministic FSMs

In our example of figure 3.1 on page 21, what would we do if there were multiple transition edges from one state to another for the same character encountered? We would have two possible transitions from a state to another and how do we decide which one to take? In our previous FSM had no choices between edges to follow because there were only single edges between states and the computation is unambiguous, i.e. $\{S\} \times \{A\} \to \{S\}$. We call these machines deterministic FSMs and machines with multiple edges or transitions (i.e. branches) between states for the same character are known as non-deterministic FSMs.

Computations involving non-deterministic FSMs follow all possible paths equally, so that when multiple edges are encountered, parallel processes are spawned for each edge. In other words, a non-deterministic FSM is composed of multiple deterministic FSMs with the all possible computations conducted depending on the input string. Any computation process that eventually accepts the input results in the non-deterministic FSM accepting the input also. It may seem that non-deterministic FSMs are more powerful, but they can only do what FSMs can do, though it may be easier to handle differing inputs. We need to add more elements to FSMs to make them more powerful.

3.4 Turing Machines

We saw that FSMs have an alphabet $\{\,A\,\}$, associated states $\{\,S\,\}$ and transition rules $\delta : \{\,S\,\} \times \{\,A\,\} \to \{\,S\,\}$. It does not however, have memory for working as a human would need to do arbitrary computations. This would also require keeping track of the "awareness" within that memory. Thus, we define a more powerful machine that has a tape for memory and a read/write head for keeping track of awareness, which is known as a Turing machine.

3.4.1 Memory

The tape is a 1D array that can be thought of as sequential access memory. The read/write head moves left or right along the array and points to a cell in the tape that corresponds to the current computation taking place. The tape may hold the input and output strings, any intermediate results or working and even the program as we shall see later in this section. Figure 3.2 shows an illustration of the tape and head.

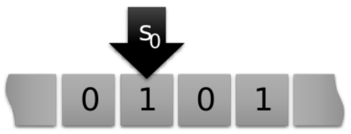


Figure 3.2: An illustration of a Turing machine with its tape and read/write head. The tape extends to infinity on both sides.

For theoretical reasons, we may consider the tape to be infinite in length. Unlike FSMs, the head also permits us to back along the input and mark on the tape. In fact, we are permitted to write any symbol we like as long as we define a tape alphabet $\{\,A\,\}_T$ that is a superset of symbols permitted on the tape with the input alphabet $\{\,A\,\}_i$. For example, for the binary alphabet $\{\,A\,\}_i = \{0, 1\}$, we could have the tape alphabet $\{\,A\,\}_T = \{0, 1, \star\}$, where we use the $\star$ symbol to represent intermediate results or working. Thus, we can write the working or intermediate results, as well as the input on the tape with unrestricted access. This allows us to conduct more powerful operations via more powerful transition functions.

3.4.2 Programs

With memory available for working and ability to read/write and move the head, we have more degrees of freedom with our transitions. Indeed, now our transitions have the form $\delta : \{\,S\,\} \times \{\,A\,\}_T \to \{\,S\,\} \times \{\,A\,\}_T \times \{L, R\}$, where L and R represent the left and right movement of the head.

Finally, the Turing machine also has an initial state s_0 like an FSM, but only has single accept and reject states, s_a and s_r respectively, because we have the ability to complete the computation instantly without having to process the entire input strin since we are free to move back and forth along the input. An example of such a computation that halts immediately is a Turing machine that checks the input contains at least a single 1 in the input string. As you can imagine, once a single 1 is encountered, regardless the length of the string and how much of the string has been processed, further computation is not required and the machine should accept and halt. Thus, we can formally define the concept of a Turing machine.

1 Definition (Turing Machine)
A Turing machine is a seven tuple $T = (\{S\}, \{A\}_i, \{A\}_T, \delta, s_0, s_a, s_r)$, *where the sets* $\{S\}, \{A\}_i$ *and* $\{A\}_T$ *are all finite and*

1. $\{S\}$ *is the set of all ppossible states,*
2. $\{A\}_i$ *is the input alphabet without the blank symbol,*
3. $\{A\}_T$ *is the tape alphabet which includes the blank symbol,*
4. $\delta : \{S\} \times \{A\}_T \to \{S\} \times \{A\}_T \times \{L, R\}$ *is the transition function,*
5. $s_0 \in \{S\}$ *is the initial state,*
6. $s_a \in \{S\}$ *is the accept state,*
7. $s_r \in \{S\}$ *is the reject state with* $s_r \neq s_a$.

The Turing machine would function as the following:

1. We assume that the tape is infinite in both directions.
2. Begin with the input placed in the middle of the tape with the head in the left most cell of the input. The rest of the tape is considered empty or blank.
3. The input alphabet $\{A\}_i$ does not have the blank symbol, so it is safe to assume that the end of the input is marked with a blank symbol.
4. The computation begins by using the tape alphabet $\{A\}_T$ and following the transition function until either the accept or reject state is reached.
5. When either of s_a or s_r is reached, the machine halts. Otherwise, the machine continues forever without halting.

Some implementations of Turing machine tape is one-sided, so that it is only infinite to the right. Turing [1937] does not mention how the tape should be implemented and indeed it makes very little difference to the computing power of the machine. In this one-sided scenario however, if the head tries to move off the tape, it merely stops at the closest cell of the tape and does not move.

3.4.3 Configurations

To track the progress and to debug Turing machine processes, it is often useful to include a notation for the current "configuration" of the machine. This is most commonly done by showing the current state of the tape that is in use and marking the current head position with the state name. Figure 3.3 shows an example of the configuration of a Turing machine in the state s_7. This notation allows us to write a plain text description of the machine configuration

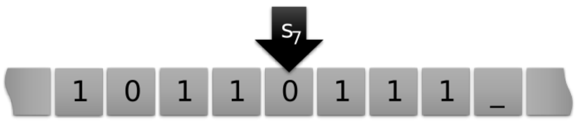


Figure 3.3: An example of a Turing machine configuration showing the current state, tape contents and read/write head position. This configuration can be written in plain text as $1011s_70111$.

easily for each step and follow the computation especially for debugging purposes.

We will close this section by showing an example of Turing machines that uses most of the concepts above. Consider a Turing machine whose task is to determine if the left side of the input is the same as the right side, where the hash symbol $\#$ is used as the separator. For example, the input string $011\#011$ should be accepted by the Turing machine. The algorithm of the Turing machine would be something like:

1. Check if the first character of the input is a hash, if it is then accept and halt,
2. Otherwise, cross of the first character with a symbol for crossing, say $\star$,
3. Enter state designated for the character and move right until the separator is found, e.g. with s_1 for 0 and s_2 for 1, then $\star 11\# s_1 011$,
4. If the first character encountered after the hash and any crosses matches the state of the encountered character, cross off and continue, else reject and halt,
5. Move left until the first cross is encountered after the hash and go back to initial state, e.g. $\star s_0 11\# \star 11$,
6. Repeat until no non-crossed characters are left, e.g. $\star\star\star s_0 \# \star\star\star$,
7. If only crosses are left either side of the hash, then accept and halt.

In the next section, we will look at ways to try increase the power of the Turing machines if possible and explore the most universal of all Turing machines.

3.5 Universal Turing Machines

3.6 Hypercomputation

CHAPTER 5

λ-Calculus

"The purpose of computing is insight, not numbers."

Richard Hamming (1915-1998)

In a previous chapter on Turing machines, we saw that Alan Turing tried to conceptualise the simplest form of a computer. He envisioned replacing a human with an electro-mechanical read/write head, some instructions and a memory for intermediate results that also kept track of the states of computation. This state-based approach serves us well in providing an intuitive understanding of a computer and computation. We also saw that the instructions and indeed the machine itself were a form of code, just as Gödel thought of theorems as unique integers that encoded all of mathematics.

In this chapter, we will describe an alternative approach that will show that all of computation can be deconstructed and derived from just a single concept. One simple concept that is required to construct all necessary logic, loops, numbers and recursion necessary to create a system that is still Turing complete. In much the same way that Turing 'built' an abstract encoding machine that represented the simplest computer, we will follow in the footsteps of Alonzo Church [1932], who constructed an abstract encoding scheme that can represent computation itself, without any assumption of a physical machine, based on the concept of a function. It allows us to solve computational problems with simpler language and notation than Turing machines, all the while better supporting parallel computations, high-level programming and being equivalent in computing power to Turing machines. This universal theory of computation based on functions is called λ-Calculus [Church, 1932], where the symbol λ is the Greek symbol lambda.

5.1 Functions

Recall that in our mathematics chapter (see section 2.3) we defined a function as a mapping between a domain and a range through the use of variables. For example, a function f representing the quadratic mapping of variable x can be written as

$$f(x) = x^2 \tag{5.1}$$

where the x in the parentheses is the argument to the function f and the function body defined after the equal sign. The notation above is not as compact and succinct as possible for encoding computations. We would essentially like to be able to write programs based on functions and having extraneous parentheses and equal signs will lengthen and complicate the our encodings.

Church [1932] introduced the λ notation to functions to allow long strings of functions to be chained or composed together that is both minimal and compact. We remove the parentheses and equal signs because they are implied, introduce the λ symbol to denote the beginning of a function and use the period symbol to denote the chaining or composition of all elements. For example, the simplest expression is $\lambda x.\ x$, where λ denotes function definition with arguments and the symbols after the period as the body of the function. Here we replace x with itself and effectively do nothing to x. We can define this as the identity function

$$I := \lambda x.\ x \tag{5.2}$$

Another example is the quadratic mapping above that can be represented as the following in his notation

$$\lambda x.\ x^2 \tag{5.3}$$

The functions defined in this way are *always* unary, i.e. only take one argument. We can then handle multiple arguments as in the following example of addition of the two variables x and y by nesting functions

$$\lambda x.\ \lambda y.\ x + y \tag{5.4}$$

5.1.1 Reductions and Normal Forms

This nesting has an operator precedence in which arguments, i.e. the terms with the λ symbol, are processed from left to right. The terms following the period after the last argument denotes the function bodies, which as simplified from right to left one at a time. For example, this can be seen in equation (5.4) as indicated by parentheses for clarity $(\lambda x.\ (\lambda y.\ x + y))$. If we wish to apply this computation as $x = 2$ and $y = 3$, we would write

$$\begin{aligned} (\lambda x.\ (\lambda y.\ x + y)3)2 &= (\lambda y.\ 2 + y)3 \\ &= 2 + 3 \\ &= 5 \end{aligned} \tag{5.5}$$

The process of simplification of λ-expression is called β-reduction. where the symbol β is the Greek symbol beta. The final result of a β-reduction that cannot be simplified further is called the β-normal form. We need not only require numbers to simplify expressions. For example, the following expressions can be β-reduced as well

$$(\lambda x.\ xx)(\lambda y.\ y) = (\lambda y.\ y)(\lambda y.\ y) \tag{5.6}$$

as well as

$$(\lambda x.\ x\ (\lambda x.\ x))y = y(\lambda x.\ x) \tag{5.7}$$

The only trap that we need to be careful of is the difference between what variables are bound to the function (are related to the arguments of the function) and what variables are free from them (do not appear in the arguments and appear from "nowhere"). Consider the y in the example

$$(\lambda x.\ xy) \tag{5.8}$$

The y is not input to the function via arguments and is thus a *free variable*, in contrast to x that is a *bound variable*. It is therefore quite easy to mix up free and bound variables of the same name. For example, the expression below has y in two different contexts

$$(\lambda x.\ (\lambda y.\ xy))y \tag{5.9}$$

The y on the far right is free but the y in the parentheses is bound. In these cases, we are able to rename variables, since the variables within functions are dummy variables and have no physical meaning other than notation. In our example, we can rename the y in the function as t to get

$$(\lambda x.\ (\lambda t.\ xt))y \tag{5.10}$$

which results in the following β-normal form

$$(\lambda t.\ yt) \tag{5.11}$$

Sometimes, the notation can be made more compact by concatenating the λ arguments since it is also implied, so that the equation (5.4) reduces to the following

$$\lambda xy.\ x + y$$

This is an example of Currying, in honour of Haskell Curry who formalised the notation of λ-Calculus [Curry and Feys, 1958]. For simplicity and easier interpretation, we will keep the notation more verbose as in equation (5.4) throughout this book.

5.1.2 Pure Functions

We also make one more requirement of the functions when using them within λ-Calculus; the functions must be pure, i.e. they have no internal state and no side effect(s). This means that functions only take inputs and produce outputs as described by the notation and do not change the inputs and any other variables or produce additional output, such as output to a terminal screen. In other words, operations such as 'print' are not allowed as it changes the state of 'outside' function elements. Though not allowed in λ-Calculus, programming languages that support λ-Calculus may still allow these operations for practical reasons. This has the significant benefit of supporting massively parallel computations, since unsolicited memory/hardware accesses such as write operations, are kept to a minimum to reduce the chance of simultaneous access that can cause programs to crash and ensures that operations are always well defined.

However, the λ-Calculus as we have defined it so far only has the concept of a function inside it. There are no numbers, or loops or even numbers! In the next section, we will define a way to encode all that is required to make a Turing complete computation system.

5.2 Church Encoding

The main challenge that λ-Calculus initially has is that it is completely bare, only supporting the concept of a unary, pure functions and nothing else, not even numbers or concepts of true and false. Though we can handle multiple arguments through Currying, we need to build the remaining concepts into the calculus from the ground up. Church [1932] developed an encoding scheme to construct all the necessary elements to create a Turing complete system, though Turing was developing his Turing machines independently at the same time. This process of encoding functionality into the notation of λ-Calculus is now known as Church encoding. We will not only show that the entire Turing complete functionality is possible, but also give examples of how elegant and simple it is to build multipliers and self interpreters within λ-Calculus.

5.2.1 Church Numerals

The first step is to create the concept of numbers within λ-Calculus and then define a method of counting between them. Numbers can be encoded by simply applying a function repeatedly n times depending on the number n being represented. For example, we can represent the number 1 as a function $\mathcal{N}_1$ as taking an input function f and variable x and applying f exactly once to x as

$$\mathcal{N}_1 := \lambda f.\ \lambda x.\ f(x) \tag{5.12}$$

Likewise for other numbers 2 and 3 as

$$\mathcal{N}_2 := \lambda f.\ \lambda x.\ f(f(x)) \tag{5.13}$$

$$\mathcal{N}_3 := \lambda f.\ \lambda x.\ f(f(f(x))) \tag{5.14}$$

and so on, noting that we can define $\mathcal{N}_0$ as applying f zero times

$$\mathcal{N}_0 := \lambda f.\ \lambda x.\ x \tag{5.15}$$

The actual form of the function f is not important right now and we can use the identity I as the function by default. These numbers are known as Church numerals.

We can then adopt the concept of the natural numbers $\mathbb{N}$ as defined by Peano and his so called Peano numbers to develop counting within the calculus. In Peano numbers, we define the natural numbers $\mathbb{N}$ recursively as the successor function $\mathcal{S}$, i.e. the "what comes after" function. For example, the successor of 0 is 1, i.e. $\mathcal{S}(0) = 1$, and $\mathcal{S}(1) = 2$ etc. Thus, we can write that $3 = \mathcal{S}(\mathcal{S}(\mathcal{S}(0)))$ and likewise for any number required by using the $\mathcal{S}$ function recursively. Therefore, we can count in the λ-Calculus by developing the successor "program" or function that takes as input a Church numeral to compute the successor from and the function to use to give us an output Church numeral as

$$\mathcal{S} := \lambda n.\ \lambda y.\ \lambda x.\ y(nyx) \tag{5.16}$$

The successor $\mathcal{S}$ function takes three arguments because we are effectively telling it to apply an additional function y to the numeral that is input to generate the next numeral, which itself requires two arguments. To test, let's apply this to $\mathcal{N}_0$, our numeral for zero. We substitute it into the successor function to find that

$$\begin{aligned}
\mathcal{S}_0 &= \mathcal{S}(\lambda f.\ \lambda x.\ x) \qquad (5.17)\\
&= \lambda n.\ \lambda y.\ \lambda x.\ y(nyx)(\lambda f.\ \lambda x.\ x)\\
&= \lambda y.\ \lambda x.\ y((\lambda f.\ \lambda x.\ x)yx)\\
&= \lambda y.\ \lambda x.\ y((\lambda x.\ x)x)\\
&= \lambda y.\ \lambda x.\ y(x) \qquad (5.18)
\end{aligned}$$

which is the definition of $\mathcal{N}_1$, since the dummy name y is equivalent to our initial name f in equation (5.12). Computing the successor of $\mathcal{N}_1$, we see the purpose of the y argument in the $\mathcal{S}$ function

$$\begin{aligned}
\mathcal{S}_1 &= \mathcal{S}(\lambda f.\ \lambda x.\ f(x)) \qquad (5.19)\\
&= \lambda n.\ \lambda y.\ \lambda x.\ y(nyx)(\lambda f.\ \lambda x.\ f(x))\\
&= \lambda y.\ \lambda x.\ y((\lambda f.\ \lambda x.\ f(x))yx)\\
&= \lambda y.\ \lambda x.\ y((\lambda x.\ y(x))x)\\
&= \lambda y.\ \lambda x.\ y(y(x)) \qquad (5.20)
\end{aligned}$$

It is there to rename the function f in the numeral to y in order to apply an additional y to compute the next numeral as we've defined in the calculus.

5.2.2 Arithmetic

Having defined numbers and counting in the calculus, we can now compute arithmetic operations such as add and multiply. In terms of multiplication, we simply use the Church numerals directly, so that multiplication of numerals n and k can be defined as

$$\mathcal{B} := \lambda n.\ \lambda k.\ \lambda f.\ n(kf) \qquad (5.21)$$

Then the multiplication of $\mathcal{N}_2$ and $\mathcal{N}_2$ is as follows

$$\begin{aligned}
(\lambda n.\ \lambda k.\ \lambda f.\ n(kf))\mathcal{N}_2\mathcal{N}_2 &= (\lambda k.\ \lambda f.\ \mathcal{N}_2(kf))\mathcal{N}_2 \qquad (5.22)\\
&= \lambda f.\ \mathcal{N}_2(\mathcal{N}_2 f)\\
&= \lambda f.\ (\lambda f.\ \lambda x.\ f(f(x)))((\lambda f.\ \lambda x.\ f(f(x)))f)\\
&= \lambda f.\ (\lambda f.\ \lambda x.\ f(f(x)))(\lambda x.\ f(f(x)))\\
&= \lambda f.\ \lambda x.\ (\lambda x.\ f(f(x)))((\lambda x.\ f(f(x)))(x))\\
&= \lambda f.\ \lambda x.\ (\lambda x.\ f(f(x)))(f(f(x)))\\
&= \lambda f.\ \lambda x.\ f(f(f(f(x)))) \qquad (5.23)
\end{aligned}$$

which is $\mathcal{N}_4$ as expected.

In terms of addition with respect to the successor function, we note that adding 2 to 2 is equivalent to computing the successor of 2 two times, which we can write as $2\mathcal{S}_2$ as

$$\begin{aligned} 2\mathcal{S}_2 &= (\lambda s.\ \lambda z.\ s(s(z)))\ \mathcal{S}\ (\lambda s.\ \lambda x.\ s(s(x))) && (5.24)\\ &= (\lambda z.\ \mathcal{S}\ (\lambda s.\ \lambda x.\ s(s(x)))(\mathcal{S}\ (\lambda s.\ \lambda x.\ s(s(x)))(z))) \\ &= (\lambda z.\ \mathcal{S}_2(\mathcal{S}_2(z))) && (5.25)\end{aligned}$$

which we can simplify as we have done for the multiplication example above.

5.2.3 Boolean Algebra

The next step is to create the concepts of true and false, known as Boolean types that forms the basis of Boolean algebra. Often in conditional statements, we need to make a decision based on a value or expression. For example, we may have statement that "If water in my cup is cold, I will do this, but if it is not I will do that". In these expressions, the true part of the if statement comes first, i.e. we choose to do the first task based on if the expression is true and the second part when the expression is false. We can mimic this in λ-Calculus by taking two arguments and defining that if the first is chosen, it represents true and if the second is chosen, then it is false. In other words, we may define true $\mathcal{T}$ and false $\mathcal{F}$ as the following in λ-Calculus

$$\begin{aligned} \mathcal{T} &:= \lambda x.\ \lambda y.\ x && (5.26)\\ \mathcal{F} &:= \lambda x.\ \lambda y.\ y && (5.27)\end{aligned}$$

This may seem arbitrary, but Boolean algebra forms a key part in digital computers as binary values are easily modelled after true and false values. We can then perform simple logical operations on Boolean values known as AND, OR and NOT, thus verifying whether our definitions for $\mathcal{T}$ and $\mathcal{F}$ are consistent.

A NOT operation takes a Boolean value and negates it, so that true becomes false and false becomes true. Consider now how we defined true and false, it selects one of the two inputs depending on the Boolean value. The NOT operation toggles the value so we need to select the opposite value based on the current argument. The inputs are Booleans p and q, i.e. selector functions, so we can then define the NOT as an opposing selector function

$$\text{NOT} := \lambda p.\ p\mathcal{F}\mathcal{T} \qquad (5.28)$$

In the body, we can see the Boolean p will select either $\mathcal{F}$ if it is true and $\mathcal{T}$ if it is false according to our convention for true and false. Likewise, we can define the AND operation as the selector function that requires both p and q are both true, so we can define it as

$$\text{AND} := \lambda p.\ \lambda q.\ pq\mathcal{F} \qquad (5.29)$$

In the body, we can see if p is false, return false because both Booleans need to be true and the value of q is not necessary to make the decision, otherwise return the value of q that will decide the final value of the operation. Finally, we can also build the OR operation based on similar principles as

$$\text{OR} := \lambda p.\ \lambda q.\ p\mathcal{T}q \tag{5.30}$$

We select true if p is true, if it is not true then perhaps q is true, so return q. The value of q will decide the final result.

These Boolean operators and Church numerals now allow us to do conditional statements. For example, we can develop conditional statements such as one to check if a number is zero

$$\text{ISZERO} := \lambda n.\ \lambda x.\ n(\lambda x.\ \mathcal{F})\mathcal{T} \tag{5.31}$$

Remember that a Church numeral takes input a function and an argument and the number it defines is the number of times the function is applied to the argument, The expression $\lambda x.\ \mathcal{F}$ is a function that will always return false, so that the ISZERO operation above is only true when the input Church numeral does not apply this function to the argument and that can only happen for the Church numeral for zero.

5.3 Combinators

At this point, notice how each λ expression is merely different combinations of the arguments x and y or whatever argument variables we have used. We combine the arguments to obtain the different expressions and operations. For example, the identity I occurs within the definition of $\mathcal{F}$ and MUL $\mathcal{B}$ occurs in other expressions such as the successor.

Just as for Boolean algebra and linear algebra, where we define a minimal set of operators that can replicate all of the operations necessary in that system, we can define a non-redundant or minimal set of combinators, i.e. λ expressions without free variables that combine arguments in different ways. In this way, we can use combinators as the atoms of λ-Calculus and problem can be solved or algorithms constructed like molecules from these atomic structures just as atoms from the periodic table in chemistry. This modular approach was pioneered by Haskell Curry [Curry and Feys, 1958] and he provided distinct names for these expressions and reformulated λ-Calculus based on them.

Let us begin by constructing Boolean algebra in this way. We can rewrite Boolean λ expressions using combinators by defining the Kestrel combinator

$$\mathcal{T} := \mathcal{K} := \lambda x.\ \lambda y.\ x \tag{5.32}$$

then the expression for false becomes

$$\mathcal{F} := \mathcal{K}I \tag{5.33}$$

because we know that $\mathcal{K}Ix = I$, but I is a function that takes an argument, so we can write $\mathcal{K}Ixy = Iy = y$, which is equivalent to $\mathcal{F}$. Sometimes the $\mathcal{K}$ combinator is considered a constant or 'const' operator (such as in a language like Haskell), because it will always return the first argument regardless of the second or even whether $\mathcal{K}$ is defined without the second argument.

We can likewise define the Cardinal combinator and redefine the NOT function as

$$\text{NOT} := \mathcal{C} := \lambda f.\ \lambda x.\ \lambda y.\ fyx \tag{5.34}$$

that flips its arguments before applying the function f, so that we get $\mathcal{CK} = \mathcal{KI}$ etc. We can also refine the OR operator to use the self application combinator called the Mockingbird combinator

$$\text{OR} := \mathcal{M} := \lambda f.\ ff \tag{5.35}$$

by noting that OR can be written from equation (5.30) as $\lambda p.\ \lambda q.\ ppq$, since if p is true then we have completed the evaluation. For arithmetic, we can do the same for MUL and in fact we have already by defining MUL as the Bluebird combinator $\mathcal{B}$ in equation (5.21).

Haskell Curry [Curry and Feys, 1958] and others showed that all of λ-Calculus can be created by just using the combinators $\mathcal{B}$, $\mathcal{C}$, $\mathcal{K}$ and $\mathcal{I}$. Although the details of this formulation is out of scope of this book, we will use some of these results to create one of the most famous results in computer science, the $\mathcal{Y}$ combinator.

5.4 The Y-Combinator

In previous sections, we showed that we can compute conditional branches (via Boolean algebra) and arithmetic via Church numerals. However, we need to augment at least one more operation to λ-Calculus to make it Turing complete: the ability to infinitely loop. One way to obtain infinite loops is to do computations recursively until a stopping condition is reached. The surprising result is that in a system where only unary, pure functions exist, we can still create the notion of recursion using the Mockingbird combinator as a starting point.

Notice that for $\mathcal{M}$, we apply the input function f to itself. When we apply $\mathcal{M}$ to itself, we get an expression that repeatedly calls itself *ad infinitum*, so that we get an infinite recursion but nothing that is useful because it has no stopping condition. Interestingly, the self application of the Mockingbird to itself is related to the halting problem we discussed in a previous chapter. In general, we cannot know if any λ expression will always have a β reduction and halt or continue forever.

What we need is a recursive combinator that takes in a function f, but calls f again with the recursive combinator only if the base condition is not satisfied. If the base condition is satisfied, we would like to stop calling it itself and allow β reduction. This is exactly what the $\mathcal{Y}$ combinator allows us to do and is defined as

$$\begin{aligned} \mathcal{Y} &:= \lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx)) && (5.36)\\ &:= \lambda f.\ \mathcal{M}(\lambda x.\ f(\mathcal{M}x)) && (5.37) \end{aligned}$$

We can see how this might work from a top level β reduction involving some combinator $\mathcal{R}$ as

$$\begin{aligned} \mathcal{YR} &= (\lambda f.\ (\lambda x.\ f(xx))(\lambda x.\ f(xx)))\mathcal{R} && (5.38)\\ &= (\lambda x.\ \mathcal{R}(xx))(\lambda x.\ \mathcal{R}(xx)) \\ &= \mathcal{R}((\lambda x.\ \mathcal{R}(xx))(\lambda x.\ \mathcal{R}(xx))) \\ &= \mathcal{R}(\mathcal{YR}) && (5.39) \end{aligned}$$

To demonstrate this amazing structure in a practical sense, we will show how to build the factorial function $n!$ for the number n. Consider the recursive form of the factorial function

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n(n-1)!, & \text{for } n > 0. \end{cases} \tag{5.40}$$

The usual recursive form of the factorial function in most procedural languages, such as Python or C/C++ would be

```
int factorial(int number)
{
   if (number == 0)
      return 1;
   else
       return number * factorial(number - 1);
}
```

We can attempt to convert the above procedural 'function' directly to λ-Calculus for which we might (incorrectly) write

$$\text{FAC} = \lambda n.\ \text{if ISZERO}(n)(1)\ \text{MUL}(n)(\text{FAC}(\text{PRED}(n))) \tag{5.41}$$

Instead, because functions in λ-Calculus are anonymous, we can write the above correctly using the $\mathcal{Y}$ combinator while setting $\mathcal{Z}$ for ISZERO as

$$\begin{aligned} \text{FAC} &= \mathcal{Y}(\lambda f.\ \lambda n.\ \mathcal{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))) && (5.42) \\ &= \mathcal{Y}\mathcal{R} && (5.43) \end{aligned}$$

and set $\mathcal{R} = \lambda f.\ \lambda n.\ \mathcal{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))$. For example, we can compute $1!$ using the above expression as

$$\begin{aligned} \text{FAC } 1 &= \mathcal{Y}(\mathcal{R})\ 1 && (5.44) \\ &= \mathcal{R}(\mathcal{Y}(\mathcal{R}))\ 1 && (5.45) \end{aligned}$$

Expanding the λ expressions

$$\begin{aligned} \text{FAC } 1 &= \lambda f.\ \lambda n.\ \text{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))(\mathcal{Y}(\lambda f.\ \lambda n.\ \text{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))))\ 1 \\ &= \lambda n.\ \text{Z}(n)(1)\ \text{MUL}(n)((\mathcal{Y}(\lambda f.\ \lambda n.\ \text{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))))(\text{PRED}(n)))\ 1 \end{aligned}$$

Now we evaluate the 1 as the argument λn and reduce the $\mathcal{Z}$, since 1 is not zero and so compute the false part of the statement

$$\begin{aligned} \text{FAC } 1 &= \text{MUL}(1)(\mathcal{Y}(\lambda f.\ \lambda n.\ \text{Z}(n)(1)\ \text{MUL}(n)(f(\text{PRED}(n)))))(\text{PRED}(1)) \\ &= \text{MUL}(1)(\mathcal{R}(\mathcal{Y}\mathcal{R}))(\text{PRED}(1)) \\ &= \text{MUL}(1)(\mathcal{R}(\mathcal{Y}\mathcal{R}))\ 0 && (5.46) \end{aligned}$$

Notice how the expression remaining is similar to equation (5.45) except requiring the evaluation with zero. The zero will trigger the base case and return a 1, allowing the recursion to reduce to the final result.

The $\mathcal{Y}$ combinator shows that even a simple system based on a compact notation system can encode not only recursion, but in conjunction with other combinators, computation itself can be reduced to just a handful of fixed modular combinations of functions and their arguments.

CHAPTER 6

Quantum Computation

"Until computers and robots make quantum advances, they basically remain adding machines: capable only of doing things in which all the variables are controlled and predictable."

Michio Kaku (1947-)

We have seen from previous chapters that the notion of computation can be reduced to the encoding of symbols. Computation is then the evolution of the symbols to another set of symbols in a definite, predictable and prescribed way. For example in Turing machines, we maintain a series of fixed states and a tape of symbols to do the computation. In these previous computation models, we are not concerned with how long it takes and that we have infinite time, instead we are concerned with whether computations are possible or not and whether the computation will complete or halt.

In an ideal computational system however, we would like to compute as much as possible within the shortest time possible. If the computation has multiple branches, we would like to explore all branches, since we are likely not to know which is the correct one. We could do many computations at once if we run multiple sub-computations to speed things up, i.e. compute lots of different chunks of the problem in parallel. This might be a kin to using multiple heads in a Turing machine and, as we saw in previous chapters, this does not make our computer more powerful in terms of computation capability. But what if we could compute all possible branches and chunks all at once and in a single computation? Is such a system even possible? Even if it is possible, how would you even design and build such a computer? In this chapter, we will show that such as computer is indeed possible and discuss the theory behind them. More details about quantum computation can also be found in Moore and Mertens [2011, Chapter 15].

6.1 Quantum Mechanics

The development of quantum mechanics (QM) comes from the simple goal of trying to understand the fundamental constituents of matter and being able to make predictions of physical systems in nature. Traditionally, we have understood all natural phenomenon before the 20th century in one of two main ways: waves or particles.

6.1.1 Wave or Particle?

We experience sound and ripples in ponds and lakes as waves, where vibrations in material or fluids are carried across space and time as pressure or density changes. We know these wave properties so well, we use them intuitively to create music with different instruments that utilise vibrations via many different mechanisms such as strings, membranes and cavities. The music we create are made up of specific set of waves that form the fundamental building blocks of all music called harmonics. These harmonics are the simplest set of waves that are possible on a string tied at both ends and are called standing waves as a result as they resonate and continue to "ring" out as you pluck the string. Trying to create waves of other types results in sound that simply dies out too fast to be heard or just sound unpleasant. Interestingly, these harmonics only exist when the total number of maximum displacements from the resting position (so called toughs) in the wave is a whole number (see Figure 6.1).

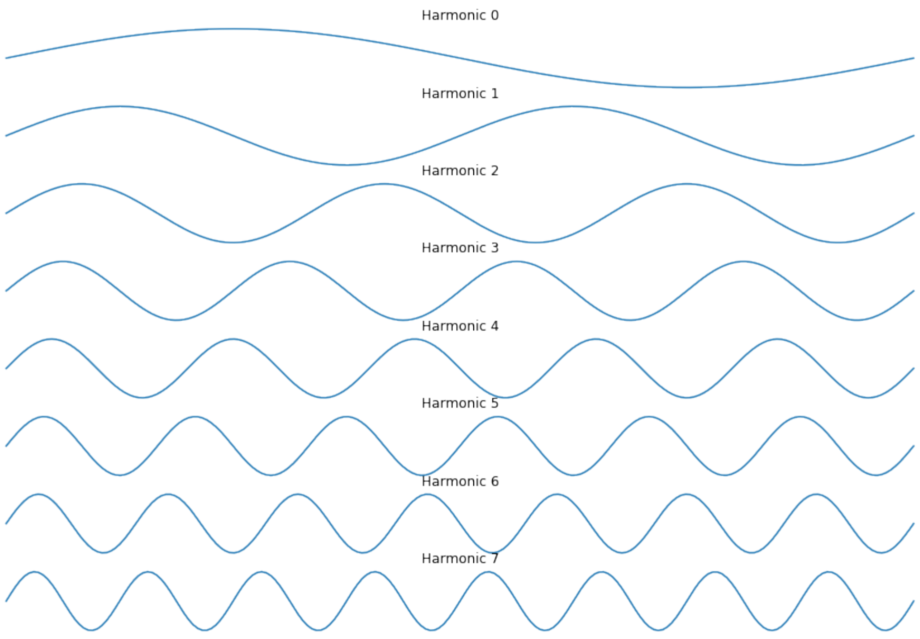


Figure 6.1: The standing waves that make the harmonics of a musical system.

Waves also exhibit another important property called superposition. Superposition is the process of creating a linear combination or weighted sum of elements, so that a superposition of a set of harmonics is a weighted sum of those harmonics. For example, the superposition $\mathbf{r}$ of two vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ can be written as $\mathbf{r} = a\,\mathbf{e}_1 + b\,\mathbf{e}_1$. In fact, this is how we write vectors in 2D to represent position, where the vectors $\mathbf{e}_1$ and $\mathbf{e}_2$ are the basis vectors for the x and y dimensions.

We also observe the particle or rigid body interaction in nature by way of collisions of these bodies, such as billiard balls and objects falling in gravity. These particles or objects exhibit a change in motion via exchanges of momentum $p = mv$, where m is the mass of the particle and v it's velocity. We also imagine planets around our sun and other celestial objects with the same ideas.

Therefore based on these experiences, we ask the obvious questions: Does all matter exist as waves or as particles at the fundamental level? What experiments confirm either hypothesis? The answer to those questions are that matter exists as neither of them (or in a form that is effectively both of them at once, depending on your point of view) and experiments confirm that matter can have both properties! This relatively new form leads to unexpected behaviour that we have no intuition for, which is usually what makes QM a challenging theory to learn.

6.1.2 Wavefunctions

All fundamental matter in the universe exists in a quantum state or a superposition of quantum states that can be described by a wavefunction ψ. A wavefunction can be thought of as a wave packet, a bundle of harmonics weighted and summed together to form an envelope of localised energy. Figure 6.2 shows an example of two wavefunctions interacting.

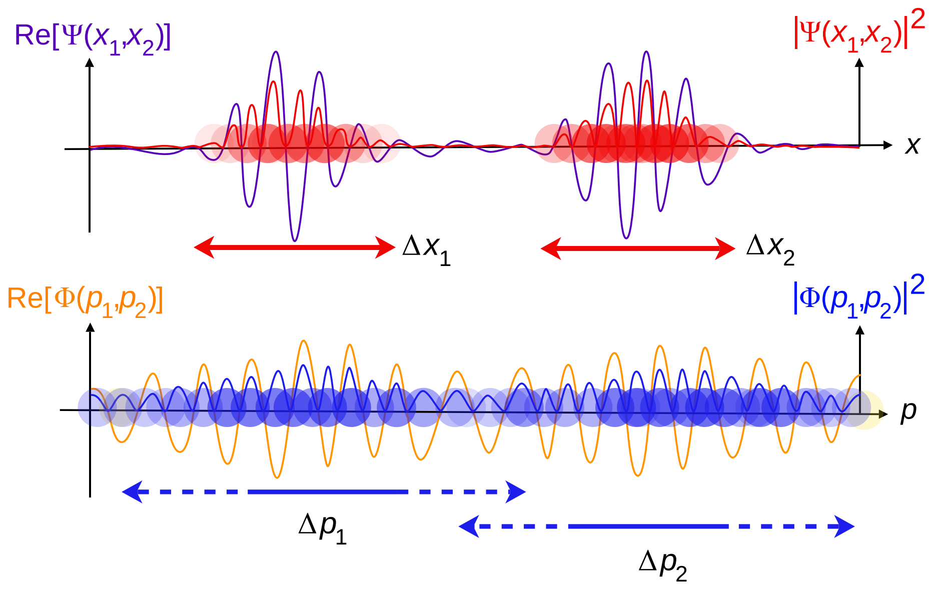


Figure 6.2: An example of the wavefunctions ψ_1 and ψ_2 as wave packets while interacting and their corresponding momenta. Original figure by Maschen.

The wavefunction model of matter explains how matter can behave as a particle or as a wave. Consider the set of waves that are superimposed to create the wavefunction. Since the constituents are waves and will exhibit wave-like properties, the summation will also. But also consider the spatial extent Δx of the wavefunction, it can be seen to effectively represent a particle of finite size. Indeed the length of the wavefunction

$$P(x) = |\psi|^2 \tag{6.1}$$

represents the probability of the particle location $P(x)$, that is the chances of finding the particle at that location upon measurement. This is often referred to as the probability density as it relates to how 'tightly packed' the packet is with respect to space (and time if the wavefunction happens to be time dependent as well). For example, one of the simplest wavefunctions are the Gaussian wave packets, whose probability density is the normal distribution. It effectively says that the particle will be found in space according to the normal distribution when a measurement is made.

The interesting concept here is the 'measurement' being made. In the macroscopic scale, making a measurement has inconsequential effect on the outcome of the measurement itself. Measuring the length of a table is hardly affected by the act of the measurement. However, at the quantum scale (typically less than an Angstrom, which is 10^{-10}m), the act of using light to 'see' the electron can cause the electron to change state or even be ejected from the matter being observed.

In QM, there are one of two consequences of making a measurement. The accepted consequence until recently has been that the wavefunction collapses into one of the outcomes that is possible according to the probability density. The collapse is the outright destruction of the quantum state to produce this outcome. This leads to a famous paradox called the Schrödinger's cat.

Imagine a cat inside an opaque box. Inside the box is a radioactive substance, whose decay is known to be a quantum process that emits an alpha particle (via quantum tunnelling, a concept that will be discussed later) according to its probability density. A detector is present on the far side of the box that effectively triggers the poisoning of the cat if an alpha particle is measured by the detector. How does one know if the cat is alive or dead without opening the box? In essence, the cat is in a superposition of states, simultaneously alive and dead until a measurement is made. Opening the box will trigger the collapse of the system into one of the possible states, i.e. the cat will be found dead or alive. This is often the conundrum that QM systems create when trying to conduct experiments and quantum computation will be no exception.

6.1.3 Uncertainty Principle

We can however note another weird phenomenon of the wavefunction representation of matter. Observe what happens when we add more and more frequencies to the wave packet. It will become more and more localised having a smaller and smaller Δx. Note that the energy of the wavefunction is directly proportionate to the different frequencies present. Using the wave nature of sound and the harmonics, having higher and higher frequencies, we must induce more and more energy into the string in order for it to vibrate faster. Thus, as we add more frequencies, it increases the ΔE, which we will show later as being equivalent to an increase in momenta Δp. Conversely, if we only use a single frequency, i.e. use a single sine curve to represent a wavefunction, we know the energy exactly as there is only a single frequency, but as it is well known that a sine curve propagates to infinity in both the positive and negative x directions, so that it is as poorly localised as possible. This give and take nature of QM is known as Heisenberg's uncertainty principle and it broadly states that

$$\Delta x \Delta p \geqslant \frac{\hbar}{2} \tag{6.2}$$

or equivalently

$$\Delta E \Delta t \geqslant \frac{\hbar}{2} \tag{6.3}$$

Here $\hbar = {}^{h}\!/_{2\pi}$, pronounced 'h bar' and $h = 6.626 \times 10^{-34}$ m^2 Kg/s and is known as Planck's constant. It is this constant that is the very definition of QM, appearing in nearly all expressions describing quantum phenomenon found in nature by QM.

6.1.4 Planck's Constant

At the beginning of the 20th century, a new phenomena was observed call the photo-electic effect. Classical physics predicted that shining light on the metals would induce a current proportional to the intensity of the light if light was modelled as a wave phenomena. The more the light incident on the metal, the more number of electrons would be excited and larger the current.

It turned out that the current was experimentally found and verified to be dependent on the frequency of the light and not the intensity. No matter how bright the light, current did not flow unless there was a minimum frequency of the light, a frequency dependent on the metal involved. Eventually it was found that the energy of the electron E given in electron volts (eV) was proportional to the incident light frequency f as

$$E = hf \tag{6.4}$$

where h is the same Planck's constant. In fact, these experiments of this photo-electric effect gave the first estimates of the constant and represented the slope of the line plotting the E and frequency f, which was always the same regardless of the type of metal.

Einstein [1905] pointed out that this could only occur if light behaved as a particle having a 'quanta' or prescribed amount of energy that is transferred to the electron allowing current to flow. It would be twenty years later until Schrödinger [1926] would show that both wave and particle natures of light could be explained using wavefunctions and use it to predict the energy levels of Hydrogen. Throughout QM, one will see Planck's constant as a recurring theme that captures the quantum nature of matter. Schrödinger [1926] work would eventually be generalised to produce a more elegant mathematical framework, known as Dirac's notation, to explain the remaining quantum phenomena including to construct quantum computing.

6.2 Dirac's Notation

We have discussed how all matter is in a superposition of quantum states ψ or most precisely $\psi(x,t)$, since the wavefunction is a function of space and time. For what follows, we will ignore the time aspect of the wavefunction, so that it is $\psi(x)$. In general, we will represent the state $\psi(x)$ has a complex-valued vector and operations between vectors will include inner products, projections and outer products. For example, the inner product of two states ψ_1 and ψ_2, i.e. the projection of one state onto another, would be

$$\int_{-\infty}^{\infty} \psi_1(x)^* \psi_2(x) dx \tag{6.5}$$

as in general ψ_1 and ψ_2 are complex valued and $*$ represents complex conjugation. Note that complex conjugation is the operation of negation of all imaginary parts to allow complex numbers to be projected back into the real number line. Since the norm of the quantum state is the probability density, we also write

$$\int_{-\infty}^{\infty} \psi(x)^* \psi(x) dx = \int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1 \tag{6.6}$$

The above notation can get very complicated when dealing with a large number of states and many operations. It is thus desirable to give these states a special name when designing a compact mathematical notation for them.

Dirac [1930] designed just such a notation that we will utilise in this chapter to discuss quantum states in general. If we define a state vector a, then the *ket* vector that represents a is denoted as $|a\rangle$ and its corresponding *bra* that represents a^* is denoted as $\langle a|$. Then we can represent superposition of state vectors $|a\rangle$ and $|b\rangle$ to give another state vector $|c\rangle$ as

$$|c\rangle = \alpha_1 |a\rangle + \alpha_2 |b\rangle \tag{6.7}$$

for arbitrary complex numbers α_1 and α_2. Thus, the inner product reduces to the simple "braket" statement

$$\langle a|b\rangle := \int_{-\infty}^{\infty} a(x)^* b(x) dx \tag{6.8}$$

and that

$$\langle a|a\rangle = 1 \tag{6.9}$$

In linear algebra, we can write any vector space via a set of basis vectors $\{e_i\}$, where $i = 1, \dots, n$ and n is the dimension of the space. The basis vectors are generally of length unity and orthogonal to each other. Orthogonality means the vectors are linearly independent and thus have no common components. In other words, the basis vectors satisfy the conditions that the norm is unity $\langle e_i|e_i\rangle = 1$ and the inner (or dot) product $\langle e_i|e_j\rangle = \delta_{ij}$, where δ_{ij} is the Dirac delta function defined as

$$\delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j. \end{cases} \tag{6.10}$$

This function simply states that the only vector that has a common component to a vector like e_i is the vector e_i itself. This set of basis vectors is often also referred to as an orthonormal set. We can simply write basis vectors compactly using direct notation for $|e_i\rangle$ as $|i\rangle$ so that

$$\langle i|j\rangle = \delta_{ij} \tag{6.11}$$

When we make measurements of physical systems, the wavefunctions of these systems are sampled discretely, so that our state vectors are discrete also, so that we can use linear algebra to describe them. The state vectors can then be defined in terms of basis vectors as a discrete superposition. For example, the two state vectors

$$|a\rangle = \sum_{i=0}^{n} \alpha_i |e_i\rangle , \qquad |b\rangle = \sum_{i=0}^{n} \beta_i |e_i\rangle \tag{6.12}$$

and likewise

$$\langle a| = \sum_{i=0}^{n} \alpha_i^* \langle e_i| , \qquad \langle b| = \sum_{i=0}^{n} \beta_i^* \langle e_i| \tag{6.13}$$

The resulting *kets* are column vectors and *bras* are row vectors

$$\langle a| = (\alpha_1^*, \ldots, \alpha_n^*) , \qquad |a\rangle = \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} \tag{6.14}$$

This changes our inner product definitions to discrete ones

$$\begin{aligned} \langle a|b\rangle &= \sum_{i=0,j=0}^{n} \alpha_i^* \beta_j \langle e_i|e_j\rangle \\ &= \sum_{i=0}^{n} \alpha_i^* \beta_i \end{aligned} \tag{6.15}$$

This also means that all the operations involving the state vectors are vector and matrix multiplications. Thus, changing states from one state to another is done by the use of operators

$$\boldsymbol{\Omega} |a\rangle = |b\rangle \tag{6.16}$$

and the operators are matrices that define the act of measurement or computation. We can see this from the equation (6.16) and (6.12) that

$$\sum_{j=0}^{n} \boldsymbol{\Omega}\, \alpha_j |j\rangle = \sum_{j=0}^{n} \beta_j |j\rangle \tag{6.17}$$

We contract the *kets* by multiplying with a *bra* $\langle i|$ both sides resulting in

$$\sum_{j=0}^{n} \langle i| \boldsymbol{\Omega}\, \alpha_j |j\rangle = \sum_{j=0}^{n} \langle i| \beta_j |j\rangle \tag{6.18}$$

which can be further simplified by taking out the scalars

$$\sum_{j=0}^{n} \langle i| \boldsymbol{\Omega} |j\rangle \alpha_j = \sum_{j=0}^{n} \langle i|j\rangle \beta_j \tag{6.19}$$

We can now define the matrix elements

$$\boldsymbol{\Omega} := \Omega_{ij} := \langle i| \boldsymbol{\Omega} |j\rangle \tag{6.20}$$

allowing us to finally write

$$\sum_{j=0}^{n} \Omega_{ij}\alpha_j = \beta_i \tag{6.21}$$

where summation over the same indices j result in a contraction from a matrix to a vector, leaving the vector with index i.

Finally, an important operation in linear algebra is the outer product or tensor product, which can be represented in Dirac notation as

$$P = |a\rangle \langle b| = a \otimes b \tag{6.22}$$

The result is a matrix P which becomes a projection operator with components $|a\rangle$ and $\langle b|$. In quantum computing, it is often more convenient to represent the outer product of k n-dimensional vectors as a n^k-dimensional vector instead to represent a superposition. Later we shall use this property to create k-bit systems, whose states are 2^k-dimensional vectors.

6.3 Qubits

We will begin by formulating classical or conventional computing using a operator like approach via linear algebra.

6.3.1 Classical Bits

Consider a simple single bit system that consists of just two states 0 and 1, we can write this system as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{6.23}$$

We can view this notation as representing a vertical switch with states $|0\rangle$ and $|1\rangle$ representing the on and off positions of that switch. We can operate on these states by using operators or gates represented as matrices and their application to states as matrix multiplication. For a 1-bit system there are only four operations possible:

1. Identity

$$\begin{aligned} I|0\rangle &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= |0\rangle \end{aligned} \tag{6.24}$$

$$\begin{aligned} I|1\rangle &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ &= |1\rangle \end{aligned} \tag{6.25}$$

2. Negation

$$\begin{aligned}
N|0\rangle &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= |1\rangle \qquad (6.26) \\
N|1\rangle &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= |0\rangle \qquad (6.27)
\end{aligned}$$

3. Set State to $|0\rangle$

$$\begin{aligned}
0\,|0\rangle &= \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= |0\rangle \qquad (6.28) \\
0\,|1\rangle &= \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
&= |0\rangle \qquad (6.29)
\end{aligned}$$

4. Set State to $|1\rangle$

$$\begin{aligned}
1\,|0\rangle &= \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= |1\rangle \qquad (6.30) \\
1\,|1\rangle &= \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= |1\rangle \qquad (6.31)
\end{aligned}$$

Now consider a 2-bit system, so that we have the four possible states $|00\rangle$, $|10\rangle$, $|01\rangle$ and

$|11\rangle$. Using our notation, we can write them as

$$\begin{aligned} |00\rangle &= |0\rangle\langle 0| \\ &= \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} && (6.32) \\ |01\rangle &= |0\rangle\langle 1| \\ &= \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} && (6.33) \\ |10\rangle &= |1\rangle\langle 0| \\ &= \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} && (6.34) \\ |11\rangle &= |1\rangle\langle 1| \\ &= \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} && (6.35) \\ & && (6.36) \end{aligned}$$

where $|a\rangle\langle b|$ is the outer product (also known as a tensor product $a \otimes b$) of the states a and b discussed in section 6.2. In addition to our operators for the single bit system, we introduce another operator called the conditional NOT or CNOT

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{6.37}$$

that only flips or applies NOT to one of the bits when one of the chosen bits is in the $|1\rangle$ state.

We can then apply this to our two bit system states to obtain

$$\begin{aligned}
C|00\rangle &= C|0\rangle\langle 0| \\
&= \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}1\\0\\0\\0\end{pmatrix} = \begin{pmatrix}1\\0\\0\\0\end{pmatrix} \\
&= |00\rangle \qquad (6.38)\\
C|01\rangle &= C|0\rangle\langle 1| \\
&= \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \\
&= |01\rangle \qquad (6.39)\\
C|10\rangle &= C|1\rangle\langle 0| \\
&= \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}0\\0\\1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\end{pmatrix} \\
&= |11\rangle \qquad (6.40)\\
C|11\rangle &= C|1\rangle\langle 1| \\
&= \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}0\\0\\0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\1\\0\end{pmatrix} \\
&= |10\rangle \qquad (6.41)
\end{aligned}$$

We are now ready to generalise our formalism to qubits.

6.3.2 Quantum Bits

A qubit is a is a bit that is in a superposition of two states (say on and off or 1 and 0), rather than exclusively in either one of two states as in a classical or binary bit. In other words, a qubit is in a continuum of the two possible states 0 and 1. A qubit still collapses into a binary bit, but like all quantum systems, the final state has a probabilistic outcome given by its probability density. In our Dirac notation, the qubit can be represented as

$$|q\rangle = \begin{pmatrix}\mu\\ \nu\end{pmatrix} \qquad (6.42)$$

where μ and ν are arbitrary complex numbers, so that $\langle q|q\rangle = 1$ and has a probability density of $|\mu|^2 + |\nu|^2$. In other words, the qubit has a $|\mu|^2$ probability of collapsing into state 0 and $|\nu|^2$ probability of collapsing into state 1.

Notice that the classical bit defined in equation (6.23) is a subset of a qubit, but now that states 0 and 1 are replaced by complex numbers. Some example qubit states include

$$\begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \qquad \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}, \qquad \begin{pmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{2} \end{pmatrix} \tag{6.43}$$

The first of these has a probability density of $1/2 + 1/2$, which implies that the qubit state is equally in both the on and off states simultaneously. In contrast to a classical bit, $|0\rangle$ has a probability of 1 to collapse to the state 0 and probability of 0 to collapse to the state 1, while $|1\rangle$ has a probability of 0 to collapse to the state 1 and probability of 1 to collapse to the state 1.

Thus, qubits can hold multiple bit states at once during a computation before they collapse when we measure them. A series of qubits then can do parallel computations, so that each thread of that computation, and its corresponding intermediate results, is represented as one of the many superimposed states held in the qubits. In other words, each thread of the parallel computation exists as a single harmonic across the qubits in a cacophony of harmonics created by superposition that represents the entire computation.

To create multiple qubits, we follow the same process constructed in section 6.3.1. A two qubit system would have

$$\begin{aligned} |qr\rangle = |q\rangle\,\langle r| &= \begin{pmatrix} \mu_1 \\ \nu_1 \end{pmatrix} \otimes \begin{pmatrix} \mu_2 \\ \nu_2 \end{pmatrix} \\ &= \begin{pmatrix} \mu_1\mu_2 \\ \mu_1\nu_2 \\ \nu_1\mu_2 \\ \nu_1\nu_2 \end{pmatrix} \end{aligned} \tag{6.44}$$

so that $|\mu_1\mu_2|^2 + |\mu_1\nu_2|^2 + |\nu_1\mu_2|^2 + |\nu_1\nu_2|^2 = 1$. For example, for the states

$$|q\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \qquad |r\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \tag{6.45}$$

will produce the following state

$$\begin{aligned} |qr\rangle = |q\rangle\,\langle r| &= \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \end{aligned} \tag{6.46}$$

which has the probability density of $\frac{1}{4}+\frac{1}{4}+\frac{1}{4}+\frac{1}{4}=1$. In other words, there is a $\frac{1}{4}$ probability that the system will collapse to one of $|00\rangle$, $|10\rangle$, $|01\rangle$ or $|11\rangle$ states.

The operators that we have constructed so far work in exactly the same way as for the formulation in section 6.3.1. There are however, several operators that only make sense in the quantum realm, which then allows us to build new types of circuits that can exploit the quantum weirdness for computation.

6.4 Quantum Circuits

An important step in building qubits is to be able to convert between classical and quantum states.

6.4.1 Operators

The Hadamard operator H allows us to create the superposition of states of a qubit with its form

$$H=\begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \tag{6.47}$$

For example, we can convert the states $|0\rangle$ and $|1\rangle$ to superpositions

$$H|0\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}\begin{pmatrix}1\\ 0\end{pmatrix}=\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix} \tag{6.48}$$

$$H|1\rangle = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}\begin{pmatrix}0\\ 1\end{pmatrix}=\begin{pmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{pmatrix} \tag{6.49}$$

Operators applied in this way take a quantum state present on one point on the unit circle to another point on the same circle. Generally, the vectors and operators are in the set of complex numbers $\mathbb{C}$, so that states are actually points on the Bloch sphere, which is a unit 2-sphere, i.e. like a surface of a ball.

However, quantum operators must have special mathematical properties to ensure that total probability is preserved, that is it does not change the norm of the state vector though it may change its direction. It follows that a quantum operator U must satisfy

$$UU^{\dagger}=I \tag{6.50}$$

where the † represents the Hermitian conjugate $U^{\dagger}=(U^{T})^{*}$. It therefore also implies that $U^{\dagger}=U^{-1}$, i.e. the operator U is self-adjoint or it is its own inverse when in Hermitian conjugate form. Equivalently, the column vectors of the matrix representing U must be orthonormal and matrices of this form are therefore referred to as unitary matrices. We can see this in the presence of the negative value in the Hadamard operator H that gives it this property. All operations in a quantum computer are simply rotations or reflections applied by unitary operators in a high dimensional vector space.

Thus, because the Hadamard operator is unitary and self-adjoint, we can convert a qubit superposition, like the ones presented in equations (6.48) and (6.49), back into a classical state simply by applying the operator H again. This is indicative of many quantum computing operators and algorithms, allowing one to convert between probabilistic and deterministic output, and represents the notion of reversible computing.

6.4.2 Circuits

To build quantum computing circuits, we represent each qubit with a ‘wire’ and the operators as boxes or symbols on these wires. For example, we write the following to represent a qubit and the Hadamard operator H acting on that qubit as shown in figure 6.4.2. The result of this

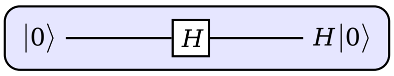


Figure 6.3: A quantum circuit for producing a superposition of states in equation (6.48).

operation will be the same as that shown in equation (6.48). We can also represent composition of operators U_2U_1 and the tensor product of operators $U_1 \otimes U_2$ of any two valid quantum operators U_1 and U_2 as boxes in series and parallel respectively as shown in figure 6.4.2. We

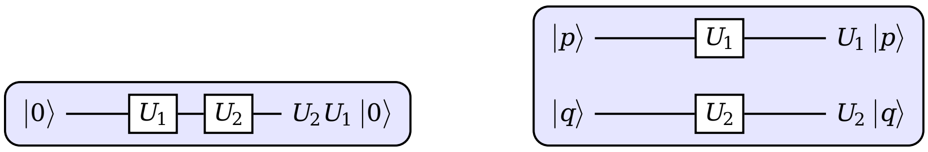


Figure 6.4: A quantum circuits for producing a composition and tensor products of operators.

can then create the four circuits for our the four operations on a single qubit using these diagrams.

However, out of the four operations on the single bit, the operations of Set $|0\rangle$ and Set $|1\rangle$ do not have a self adjoint, i.e. they do not obey equation (6.50) and are not reversible while the other two operations (identity and negation) are reversible. The Set operations overwrite the result regardless of the input that makes it impossible to recover the initial input without further information. We need to create a larger matrix that can still encode this operation and obey equation (6.50), thereby making these operations reversible. We can do so by using an additional qubit to encode the output (top wire) and input qubit (bottom wire), so that we can tell what happened and reverse the operation. This means all our four operations will have to be constructed using two wires and encode the operations accordingly. Once we use two wires, the Set operations are fairly simple and straight forward (see figure 6.4.2 on the next page). They simply involve two non-interacting wires assuming the output qubit is always $|0\rangle$ initially. By default, such a setup is already a Set $|0\rangle$ operation and we can use a simple NOT operator X to obtain the Set $|1\rangle$.

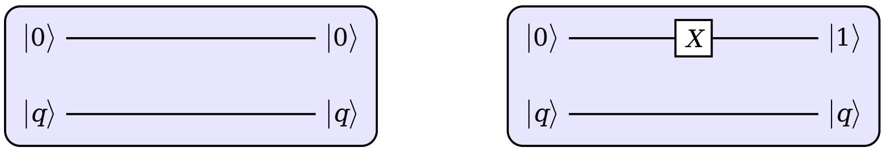


Figure 6.5: A quantum circuits for producing Set $|0\rangle$ and $|1\rangle$ operations.

To implement a simple identity operation, we require the use of the CNOT operator to create the circuit as shown in figure 6.4.2 on the left. The dark circle is the control bit and it decides

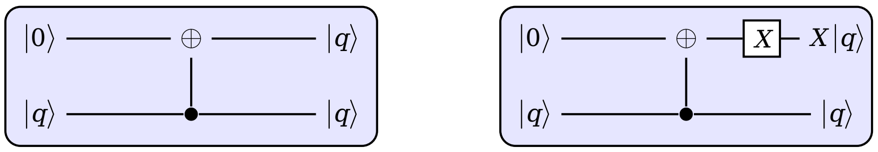


Figure 6.6: A quantum circuit for producing an identity and negation operations on states.

whether the output bit is negated or not. Thus, the circuit function as an identity operation by only flipping the output qubit to $|1\rangle$ when input bit is also $|1\rangle$, otherwise it remains as $|0\rangle$, which is also the same as the input in this case. The negation operation shown in figure 6.4.2 on the right is simply the negated identity result using the X (NOT) operator on the output qubit.

The circuits presented so far represent the basic notations required for build quantum algorithms. Now we have all the necessary machinery to build the algorithms that will allow us to harness the main power of quantum computing: exponential speed ups.

6.5 Deutsch’s Oracle Problem

Imagine now that we have a problem of trying to determine which of the four operations exist as a black box that we know nothing about. It can take inputs and produce outputs, but we are not told anything about the operator that resides inside the box. We are allowed to provide inputs and measure outputs, i.e. for inputs and output states $|0\rangle$, we may get the arbitrary states $|p\rangle$ and $|q\rangle$ (see figure 6.5). Can we determine what the operator is within the box with

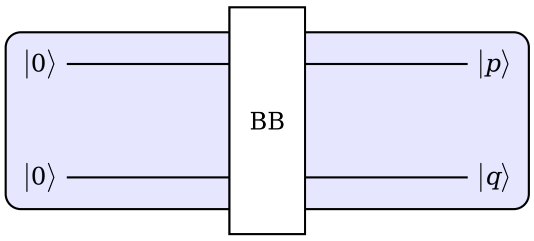


Figure 6.7: Deutsch’s oracle problem with an black box (unknown) operator.

as few queries as possible?

Classically, for a single bit of information, we would require a total of 2^1 queries into the black box to determine what operation was actually applied. For a series of n bits, a total of 2^n queries would be required. This is because for every bit input to the box, the bit could've have been negated or set to a state and the operation would be indistinguishable from each other. The only way to know is to try as many bits as uniquely possible to determine the operator in the box. However, as we shall see, with a quantum computer, we can always determine the operator present with only a single query resulting in an exponential speed up! For the purposes of this book, we will examine the single bit black box and solve the problem using our quantum circuits from the previous section.

Consider the four operations on a single bit: identity, negation, set $|0\rangle$ and $|1\rangle$. The last two are constant operations that do not take the input state into account. We can see this as the quantum circuit for these wires that do not interact unlike the first two operators that involve the CNOT gate. We can thus divide the operators up into two categories: variable and constant operators. Our goal is to use superposition of qubits to separate out the two types of operators. The variable type of operator should interact with the output and input bits to produce a different tensor product state than for those constant operators that do not interact.

Now consider the quantum circuit for Deutsch's problem as show in figure 6.5, where the M operator represents making the measurement. We can insert the relevant quantum circuits

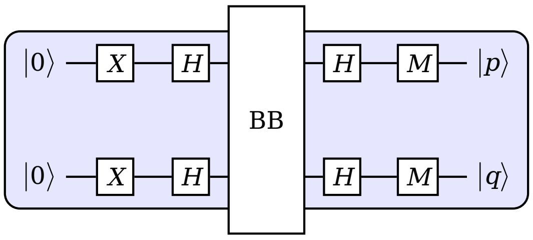


Figure 6.8: A quantum circuit for Deutsch's problem with the black box (unknown) operator.

corresponding to the four possible operations in place of the black box in figure 6.5 and analyse the outcomes. For Set $|0\rangle$ (left) and Set $|1\rangle$ (right) operators as shown in figure 6.5, we can replace the black box and evaluate what happens. Essentially, the Set $|0\rangle$ and Set $|1\rangle$ operations

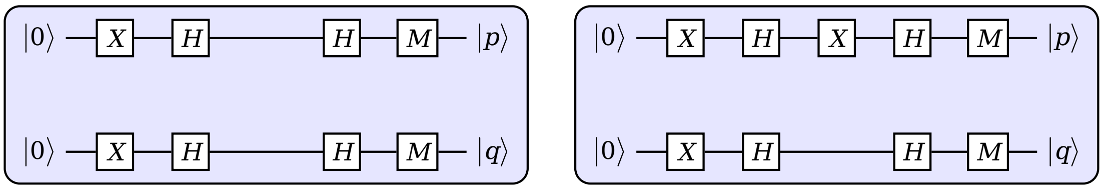


Figure 6.9: Quantum circuits for Deutsch's problem as Set $|0\rangle$ and $|1\rangle$ operators.

pass through without any changes as our previous circuit constructions have shown. We can write out the (tensor) product state of the these circuits to see the outcomes. Top and bottom wires of the Set $|1\rangle$ will result in

$$\begin{aligned}
HHX|0\rangle &= X|0\rangle \\
&= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\
&= |1\rangle \qquad (6.51)
\end{aligned}$$

The product state is then just $|11\rangle$ as seen from equation (6.40). The bottom wire in Set $|1\rangle$ is the same as equation (6.51) and the top wire is simply

$$\begin{aligned}
HXHX|0\rangle &= HXH|1\rangle \\
&= HX\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = HX\begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} \\
&= H\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix} = H\begin{pmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \\
&= \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}\begin{pmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} \\
&= |1\rangle \qquad (6.52)
\end{aligned}$$

Since the probability density of the result becomes $0^2 + (-1)^2$ is the same as $|1\rangle$, the resultant state is $|1\rangle$. Thus when using the constant operators, the result is always $|11\rangle$.

Now examine the result if the black box operator is replaced with the variable operators: identity and negation as shown in figure 6.5. In both instances we use a CNOT gate, so we must

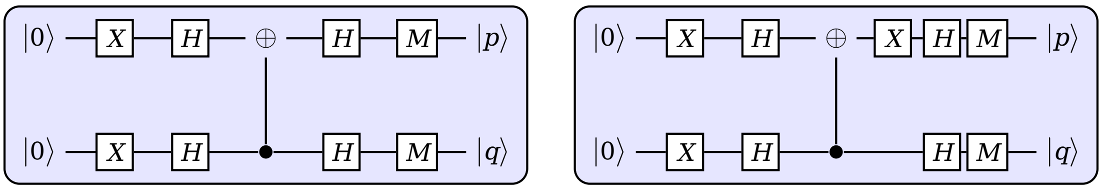


Figure 6.10: Quantum circuits for Deutsch's problem as variable operators identity and negation.

use the product states $|00\rangle$ etc. throughout. Since both X and H to both qubits independently, we can simplify the expression so that only the CNOT operation is left to apply. Thus, we can write the identity operation as

$$\begin{aligned}
CHX|00\rangle &= C\left(\begin{pmatrix}\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\end{pmatrix}\otimes\begin{pmatrix}\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\end{pmatrix}\right) = C\begin{pmatrix}\frac{1}{2}\\-\frac{1}{2}\\-\frac{1}{2}\\\frac{1}{2}\end{pmatrix}\\
&= \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}\frac{1}{2}\\-\frac{1}{2}\\-\frac{1}{2}\\\frac{1}{2}\end{pmatrix} = \begin{pmatrix}\frac{1}{2}\\-\frac{1}{2}\\\frac{1}{2}\\-\frac{1}{2}\end{pmatrix}\\
&= \frac{1}{2}\begin{pmatrix}1\\-1\\1\\-1\end{pmatrix} = \left(\begin{pmatrix}\frac{1}{\sqrt{2}}\\\frac{1}{\sqrt{2}}\end{pmatrix}\otimes\begin{pmatrix}\frac{1}{\sqrt{2}}\\-\frac{1}{\sqrt{2}}\end{pmatrix}\right)\\
&= |01\rangle
\end{aligned} \tag{6.53}$$

Likewise apply a similar analysis for the negation circuit, we simply need to handle the additional NOT gate on the output bit. The result is similar to equation (6.52). Therefore, the resultant state remains $|01\rangle$ as with the identity case. Thus, we have shown that we can classify the unknown operator into one of the two categories as predicted but with only a single query. If the resulting state from input and output states $|0\rangle$ is $|11\rangle$, then it is a constant operator and if the resulting state is $|01\rangle$ it is a variable operator.

The above can be extended to n bits and is known as the Deutsch-Josza problem [Deutsch and Jozsa, 1992]. Although this problem is out of scope for this book, it can be seen that the superposition of states introduced by the Hadamard gate is the key in separating out the two categories. We can note that the different between constant and variable operations is the CNOT gate, so we can also view it as amplifying the difference between the categories and minimising the similarities between the categories. This is similar to how Shor [1994] constructed his algorithm for factoring large numbers, by using periodicity of roots of unity, we can cancel out the undesired categories to give us the answer we seek. These and other algorithms are set to benefit greatly when qubits are made more stable, so that quantum computers can become more readily available.

CHAPTER 7

Summary

"Computer science is not as old as physics; it lags by a couple of hundred years. However, this does not mean that there is significantly less on the computer scientist's plate than on the physicist's: younger it may be, but it has had a far more intense upbringing!"

Richard Feynman (1918-1988)

We can summarise all the ideas presented in this book as a single idea: computation can be encoded into many different forms, each with its uses, but all are equivalent. This is a powerful result, allowing us to both navigate difficult problems and limitations by shifting to another encoding scheme, one that suits the problem better.

7.1 State-based Computation

Gödel [1931] showed that mathematical theorems can be encoded as integers. Turing [1937] extend his result to show that computation can be encoded as symbols on a tape, equivalently as integers as well. His notion of computation comes from the simplest concept of a computer, one that replaces a human computer with a machine that has a working area or memory for storing intermediate results and an electro-mechanical head that shifts the awareness of the computation from location to location on this memory. These Turing machines allow for universal computing by keeping track of a finite number of states and an infinite amount of memory. Turing's state based approach to computation is easily realisable via digital circuits that have given rise to the modern day computers we use today.

However, we saw that this type of computation need not only be implemented via digital circuits, but with simple rules using cells. Indeed, we could build cellular circuits that could be used to construct Turing machines and therefore be capable of universal computation. In fact, the back and forth of a Turing machines head while executing a program's instructions and completing a computation could also be mimicked by the jittering of an ant around a 2D grid that is known as Langton's Ant. The simple rules of this ant is also capable of universal computing and shows us how computation can be in fact constructed just about anywhere!

7.2 Functional Computation

Although Turing thought of computer as a machine that did encoding, a view that become the de facto standard for modelling computation, Church [1932] envisioned of the most compact encoding scheme directly that does not require any machine! An entity in its own right, the λ-Calculus brings an encoding scheme that beings together all the elements required by universal computation from just the concept of a function.

Instead of thinking about how a computation could be implemented on machines, which can result in long set of instructions that shift symbols around on a tape, the idea of functional computation is to focus on the problem and reduce it to a pure, precise arrangement function symbols. The result is an elegant representation of most concepts in computation that can be used as a starting point to create programs that have a more mathematical basis and therefore less bugs.

It is this approach to focus on solving the computational problem at hand directly, rather than thinking about how to convert the problem so that it can be solved on a machine that has become the shift in focus in recent times. With the computational power now available to us, it is time to dispense the low level programming and solve problems as they are meant to be solved.

7.3 Quantum Computation

But will our computers continue to become more and more faster according to Moore's law, where the speed of computers double roughly every 18 months? Surely we will reach the speed limit at some point in the not to distant future? Quantum computing could provide the way forward by allowing massively parallel computing by way of qubits in multiple states at once. Algorithms have already been developed that could change the way our financial institutions function and disrupt the way we live and work. However, the technology to create a large number of stable qubits is still under development. Perhaps, in the not to distant future, we will have quantum supremacy, where quantum computers are stable enough for regular use. Till then we will have to rely on our digital circuit based computers to delivery the computing power that we desire.

Abbreviations

1D	one dimensional	19
2D	two dimensional	11
3D	three dimensional	iv
GCD	greatest common divisor	8
ACE	automatic computing engine	18
FSM	Finite State Machine	iv
QM	quantum mechanics	41

Bibliography

Berlekamp, E., J.Conway, R.Guy, 1982. Winning Ways for your Mathematical Plays. Vol. 2.

Church, A., 1932. A set of postulates for the foundation of logic. Annals of Mathematics 33 (2), 346–366.
URL https://doi.org/10.2307/1968337

Curry, H. B., Feys, R., 1958. Combinatory Logic, Volume I. North-Holland.

Deutsch, D., Jozsa, R., 1992. Rapid solution of problems by quantum computation. Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences 439 (1907), 553–558.
URL https://doi.org/10.1098/rspa.1992.0167

Diophantus, 100. Arithmetica. Springer-Verlag Berlin and Heidelberg GmbH & Co. KG (December 31, 1982).

Dirac, P. A. M., 1930. The Principles of Quantum Mechanics. Clarendon Press.

Einstein, A., 1905. Über einen die erzeugung und verwandlung des lichtes betreffenden heuristischen gesichtspunkt. Annalen der Physik 322 (6), 132–148.
URL https://doi.org/10.1002/andp.19053220607

Euclid, 300BCE. The Elements.

Euler, L., 1763. Theoremata Arithmetica Nova Methodo Demonstrata. Novi Commentarii Academiae Scientiarum Petropolitanae 8, 74–104.

Feynman, R., 2005. The Pleasure of Finding Things Out: The Best Short Works of Richard P. Feynman. Helix Books.

Gardner, M., 1970. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". Scientific American (223), 120–123.
URL http://www.jstor.org/stable/24927642

Gauss, C. F., 1801. Disquisitiones Arithmeticae. Yale Univeristy Press.

Gödel, K., Dec 1931. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. Monatshefte für Mathematik und Physik 38 (1), 173–198.
URL https://doi.org/10.1007/BF01700692

Klein, F., Mar 1893. Vergleichende betrachtungen über neuere geometrische forschungen. Mathematische Annalen 43 (1), 63–100.
URL https://doi.org/10.1007/BF01446615

Langton, C. G., 1986. Studying artificial life with cellular automata. Physica D: Nonlinear Phenomena 22 (1), 120 – 149, proceedings of the Fifth Annual International Conference.
URL https://doi.org/10.1016/0167-2789(86)90237-X

Moore, C., Mertens, S., 2011. The Nature of Computation. Oxford University Press, Inc., New York, NY, USA.

Schrödinger, E., Dec 1926. An undulatory theory of the mechanics of atoms and molecules. Phys. Rev. 28, 1049–1070.
URL https://doi.org/10.1103/PhysRev.28.1049

Shor, P. W., 1994. Algorithms for quantum computation: Discrete logarithms and factoring. In: Proceedings 35th annual symposium on foundations of computer science. IEEE, pp. 124–134.
URL https://doi.org/10.1109/SFCS.1994.365700

Sipser, M., 2013. Introduction to the theory of computation / Michael Sipser., 3rd Edition. Cengage Learning, Andover.

Turing, A. M., 1937. On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society s2-42 (1), 230–265.
URL https://doi.org/10.1112/plms/s2-42.1.230