How to Program a Calculator

Numerical analysis of common functions

Jake Darby

Abstract

This document will discuss and analyse various numerical methods for computing functions commonly found on calculators. The aim of this paper is to, for each set of functions, compare and contrast several algorithms in regards to their effiency and accuracy.

Contents

1 Introduction

For many thousands of years all calculations that a Human might want performing had to be done by hand. For simple calculations such as addition, subtraction and multiplication this was not such an issue, but as society evolved we wanted to know the answer to increasingly hard questions. The Greeks' sought to find a value for π, and ended up with the bounds that $\frac{223}{71} < \pi < \frac{22}{7}$, which while sufficient for their needs is not sufficient for ours in the modern era.

At the same time many functions were being studied to find solutions, often arising from practical concerns. For instance finding the square root of any arbitrary number has been important to architects since the time of the ancient Babylonian mathematics. Similarly long studied have been the periodic trigonometric functions due to their relation to triangles, or exponential functions due to their use in finance such as interest on loans.

The difficulty of these methods is that there is typically no simple way of getting an exact answer, if in fact one is available. Over time methods were developed that would allow a person to calculate an approximate answer to their problem, given enough time and patience. Such methods tended to be long and tedious work, which even lead to the profession of a human computer from the early 17th century until the 20th century; who would be hired to perform these long tedious calculations.

By the time of the Renaissance period people had started to build early mechanical calculators to help in these endeavours. Such calculators were typically capable of only addition and subtraction, which could be used to implement multiplication and division if one so wished. Later these machines became more elaborate, capable of multiple simple functions, or designed to perform one more complicated function. A famous example would be Charles Babbage's difference engine which was a large mechanical calculator that would tabulate polynomial functions developed in the early 1800s.

Finally in the 20th century electronic computers were created and soon replaced both mechanical and human calculators. Such electronic machines had many benefits over both their human and mechanical counterparts, and soon it became common place to use electronic computers to perform mathematical computations. Nowadays computers have become faster and smaller, and the average person's phone outstrips the entire computing power of NASA during the Apollo missions.

However despite the speed of the calculations these modern computers still need to be instructed in how to evaluate the functions asked of it. This document will take some common functions that any calculator will answer in the blink of an eye accurate to around 10 significant digits, and explore how they may be computed. In particular this document will be comparing the speed at which these computations can be performed versus the accuracy of their results.

1.1 Code and Computers used

During this project I will be discussing the implementation of various algorithms. I will be implementing these algorithms in the C programming language, using the C11 standard.

I chose the C programming language to implement my algorithms in because, once it compiles to binary machine code, the programs produced tend to be very efficient. This is partly due to the low-level of C programming, having relatively close control over direct CPU actions; however this does come at the cost of losing higher functionality that many other programming languages offer. A second reason for the efficiency is due to C's long history, originally being developed in 1969-1970, which has lead to several very efficient compilers.

I will be implementing most programs using C's built in primitive types, typically `int`, `unsigned int` and `double`. On a computer an `int` is an integer that can represent both positive and negative bits using twos compliment, this gives an `int` using n bits a minimum value of -2^n and a maximum value of $2^n - 1$. Typically a computer will store an `int` as 32 bits, though some computers may use more or less bits. An `unsigned int` is very similar to an `int`, but does not represent negative values, and thus an `unsigned int` of n bits has a minimum value of 0 and a maximum value of $2^{n+1} - 1$.

If an integer of a specific number of bits is needed then the header `stdint.h` may be used which defines `int_N` and `uint_N` which respectively represent `int` of N bits and `unsigned int` of N bits; The typical values of N are 8, 16, 32 and 64.

In C a `double` is a floating point representation of a real value, that typically follows the IEEE 754 standard for double-precision binary floating points. This standard has:

- 1 bit for the sign of the number, s
- 11 bits for the exponent, e
- 52 bits for the significand, $b = b_0 b_1 b_2 \dots b_{51}$
- A value that is understood to be:

$$(-1)^s \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) \times 2^{e-1023}$$

This gives a `double` value a precision of around 15-17 significant decimal digits. While this is good for most applications, there are applications where we may want even more precision than this. To solve this I will be implementing certain algorithms using the GNU Multiple Precision Arithmetic Library (referred to as GMP) as well as GNU MPFR Library(referred to as GMP), which was built upon GMP to correct and optimise the original. These libraries allow the use of arbitrary precision real values, given enough memory space, as well as integers longer than C's standard integer types can hold.

An important point to note that will be useful later on is that due to the storage structure of C's `double`s and the MPFR `mpfr_t`s which also use a floating point representation. In the storage of the significand both data types work such that the value of b is in the range $[\frac{1}{2}, 1)$. This is useful as it means that if we have a stored value x, then it is very easy to extract $\alpha \in [\frac{1}{2}), \beta \in \mathbb{Z}$ such that $x = \alpha \cdot 2^{\beta}$. The value of this observation will be in restricting the range over which functions need to be evaluated later in the document.

I will be compiling and testing all of my code on a benchmark machine running a light version of Ubuntu 14.04, using the GNU C Compiler. The specifications of the machine, that may impact performance are:

- An Intel i5-4690K processor running at 4GHz. This processor uses a 64 bit architecture.
- 8Gb of DDR3 RAM
- A modern chipset on the motherboard

2 General Definitions and Theorems

This section will list some general definitions and theorems which will be used throughout the document. This will not be an exhaustive or in depth view of such concepts but merely an overview to allow easier reading of the material going forwards.

2.1 Methods

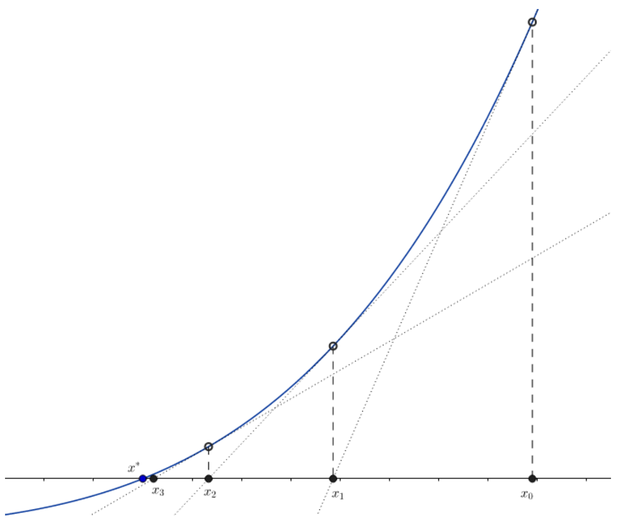
In this document we will look at various functions, such as root functions, trigonometric functions, among others. Despite the variety of functions being analysed there are several methods that are useful for more than one function, or are worth analysing before their use.

2.1.1 Newton-Raphson Method

The Newton-Raphson Method is named after Sir Isaac Newton and Joseph Raphson. It is a method that takes a continuously differentiable function f and it's derivative f', as well as an initial guess x_0, to create successively more accurate solutions to x where $f(x) = 0$.

The motivation of the method can be seen in figure **??**, where we take an initial guess x_0 of the root x^*. The tangent to the curve above x_0 is then found, and has the equation $y = f'(x_0)(x - x_0) + f(x_0)$, by setting $y = 0$ and solving for x we find x_1. By repeating this process and starting from a good enough x_0 we hope to find successively closer approximations to x^*.

Figure 2.1.1: Demonstration of Newton-Raphson Method



The specific definition of the Newton-Raphson method that I will be using in this document is below:

Definition 2.1.1.1. Given $f \in \mathcal{C}(\mathbb{R})$, f' being the derivative of f, and $x_0 \in \mathbb{R}$; then we define:

$$x_{n+1} := x_n - \frac{f(x)}{f'(x)} \forall n \in \mathbb{N}$$

The Newton Raphson method is not suitable for all problems and there are in fact many cases in which it behaves poorly. One such case is when $f'(x_n) \approx 0$ as the value of x_{n+1} will be very close to x_n and thus $f'(x_{n+1}) \approx 0$. Further bad choices of x_0 can lead to the method diverging or entering cycles between two points indefinitely, however we will see that we do not need to be concerned with these issues for our uses of the method.

2.1.2 Taylor Series Expansion

The Taylor Series formulation was created by Brook Taylor in 1715, based off of the work of Scottish mathematician James Gregory. The Taylor Series describes a method of representing any infinitely differentiable function as an infinite power series.

Definition 2.1.2.1. Given $f : \mathbb{R} \to \mathbb{R}$ which is infinitely differentiable on an open ball B centred at $a \in \mathbb{R}$, we define the Taylor Series of f on B to be:

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

It was shown that on the open ball B from the above definition we have that $f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$, i.e. a function is equal to it's Taylor polynomial on the ball for which it is defined. We can then, use this fact to define a polynomial that will approximate our function f at $x \in \mathcal{I} \subset \mathbb{R}$

Definition 2.1.2.2. Given $f : \mathbb{R} \to \mathbb{R}$ which has a Taylor Series of $\sum_{n=0}^{\infty} c_n x^n$, we define the Taylor Polynomial of degree $N \in \mathbb{N}$ to be

$$p_N(x) := \sum_{n=0}^{N} c_n x^n = c_0 + c_1 x + c_2 x^2 + \cdots + c_N x^N$$

A commonly used type of Taylor series is the McClaurin series which is a Taylor series in a ball around $a = 0$. Thus a McClaurin series has the form:

$$\sum_{n=0}^{N} \frac{f^{(n)}(0)}{n!} x^n$$

Some examples of simple McClaurin Series are:

$$\frac{1}{1-x} = \sum_{n=0}^{\infty} x^n \qquad \forall x \in (-1,1)$$

$$(1+x)^k = \sum_{n=0}^{\infty} \binom{k}{n} x^n \qquad \forall x \in (-1,1), k \in \mathbb{N}$$

2.2 Errors

The error of an approximation x_n for some x^* is a measure of how much x_n differs from x^*. We will use the error of approximations to discuss the convergence of methods as well as describing their accuracy.

There are several ways of evaluating the error of an approximation which each have their own uses. The error measures that we will use in this document are detailed below:

Definition 2.2.1. If we have a value v and it's approximation $\tilde{v}$, then the absolute error is

$$\epsilon := |v - \tilde{v}|$$

The absolute error is useful in guaranteeing a certain level of accuracy that a given implementation of a method will give; for instance if $epsilon < 10^{-3}$ then the approximation is accurate to at least 3 decimal places. Uses of absolute error in the document will use ϵ as their absolute error variable.
As the absolute error of an approximation is hard or impossible to accurately calculate during program execution, we want a way to estimate it. Typically our computations will produce a sequence of approximations $x_0, x_1, x_2, \ldots$, and thus we define the following:

Definition 2.2.2. If we have the sequence $(x_n)_{n\in\mathbb{N}}$, then the iteration error is defined as:

$$\delta_n := |x_n - x_{n-1}|$$

While it is often impossible to calculate ϵ_n it is very easy to calculate δ_n from the generated approximations. This estimate is best used when we know that the convergence is rapid, as in these cases δ_n is a good approximation of ϵ_n.

2.3 Convergence

definition
As our methods of approximating functions will typically generate a sequence of values $x_0, x_1, x_2, \ldots$ then we want to ensure that the approximations are approaching the correct value. We consider here what it means for a sequence to converge to a limit value, and some useful results for later chapters.

Definition 2.3.1. A sequence $(x_n \in \mathbb{R} : n \in \mathbb{N})$ converges to x uniformly if

$$\forall \tau \in \mathbb{R}_0^+ \exists N \in \mathbb{N} : \epsilon_n := |x - x_n| < \tau \forall n \in [N, \infty) \cap \mathbb{Z}$$

Remark 2.3.1.1. We will typically use the notation that $\lim_{n\to\infty} |x_n - x| = 0$, to denote that $(x_n : n \in \mathbb{N})$ converges to x.

Theorem 2.3.1. $(x_n \in \mathbb{R} : n \in \mathbb{N})$ *converges to* x *uniformly if and only if*

$$\forall \tau \in \mathbb{R}_0^+ \exists N \in \mathbb{N} : |x_n - x_m| < \tau \forall m, n \in [N, \infty) \cap \mathbb{Z}$$

Proof. For $\implies$:
Suppose that $(x_n : n \in \mathbb{N})$ converges to x uniformly. Then $\forall \tau \in \mathbb{R}_0^+ \exists N \in \mathbb{N} : |x_n - x| < \tau \forall n \in [N, \infty) \cap \mathbb{Z}$.
Thus suppose $N \in \mathbb{N}$ is such that $|x_n - x| < \frac{\tau}{2} \forall n \in [N, \infty) \cap \mathbb{Z}$.
Then if $n, m \geq N$ we see that

$$|x_n - x_m| \leq |x_n - x| + |x_m - x| \leq \tau$$

For $\Leftarrow$:
Omitted for brevity. □

We have shown now what it means for a value to converge to a limit, but not all sequences that approach a limit do so at the same pace. For example if we consider the sequences $x_n := 2^-n$ and $y_n := 10^-n$, then it is obvious that the limit of both sequences is 0, but y_n approaches the limit faster. This leads to the following definition of the rate of convergence.

Definition 2.3.2. If $(x_n \in \mathbb{R} : n \in \mathbb{N})$ is a sequence that converges to x, then it is said to converge:

- Linearly if $\lambda \in \mathbb{R}^+$ and
$$\lim_{n \to \infty} \frac{|x_{n+1} - x|}{|x_n - x|} = \lambda$$
- Quadratically if $\lambda \in \mathbb{R}^+$ and
$$\lim_{n \to \infty} \frac{|x_{n+1} - x|}{|x_n - x|^2} = \lambda$$
- Order $\alpha \in \mathbb{R}_0^+$ if $\lambda \in \mathbb{R}^+$ and
$$\lim_{n \to \infty} \frac{|x_{n+1} - x|}{|x_n - x|^\alpha} = \lambda$$

The higher the order of convergence of a sequence the faster it approaches it's limit, therefore we are looking for algorithms with high orders of convergence. Many regular series have linear convergence and quadratic convergence is typically very rapid, while orders above quadratic are hard to construct for useful functions.

A useful result is that, under the correct circumstances, the Newton-Raphson method can be shown to have quadratic convergence. The following proof assumes that $\epsilon_n := |x^* - x_n|$:

Theorem 2.3.2. *Let f be a twice differentiable function, x^* be a solution to $f(x) = 0$ and $(x_n : n \in \mathbb{N})$ be a sequence produced by the Newton-Raphson Method from some initial point x_0. If the following are satisfied, then $(x_n : n \in \mathbb{N}_0)$ converges quadratically to x^*:*

$\mathbf{NR}_1$**:** $f'(x) \neq 0 \forall x \in I := [x^* - r, x^* + r], \text{ where } r \in [|x^* - x_0|, \infty)$

$\mathbf{NR}_2$**:** $f''(x) \textit{ is continuous } \forall x \in I$

$\mathbf{NR}_3$**:** $M |\epsilon_0| < 1 \text{ where } M := \sup \left\{ \left| \frac{f''(x)}{f'(x)} \right| : x \in I \right\}$

Proof. By Taylor's Theorem with Lagrange Remainders we have that

$$0 = f(x^*) = f(x_n) + (x^* - x_n)f'(x_n) + \frac{1}{2}(x^* - x_n)^2 f''(y_n)$$

where $0 < |x^* - y_n| < |x^* - x_n|$.

Then we get the following derivation:

$$\begin{aligned}
& f(x_n) + (x^* - x_n)f'(x_n) = -\frac{1}{2}(x^* - x_n)^2 f''(y_n) \\
\implies & (\frac{f(x_n)}{f'(x_n)} - x_n) + x^* = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(x^* - x_n)^2 && \text{as } \mathrm{NR}_3 \implies f'(x_n) \neq 0 \\
\implies & x^* - x_{n+1} = -\frac{1}{2}\frac{f''(y_n)}{f'(x_n)}(x^* - x_n)^2 \\
\implies & \epsilon_{n+1} = \frac{1}{2}\left|\frac{f''(y_n)}{f'(x_n)}\right|\epsilon_n^2 && \text{by taking absolute values}
\end{aligned}$$

As NR_2 holds then M exists and is positive, and therefore we have:

$$\epsilon_n \leq M\epsilon_{n-1}^2 \leq M^{2^n-1}\epsilon_0^{2^n}$$

We now aim to show that we have convergence, i.e. $\lim_{n\to\infty} x_n = x^*$; to do this it suffices to show that $\lim_{n\to\infty} \epsilon_n = 0$.

Consider the sequence $(z_n := M^{2^n-1}\epsilon_0^{2^n} : n \in \mathbb{N}_0)$. We know that $0 \leq \epsilon_n \leq z_n \forall n \in \mathbb{N}_0$, so it then follows that if $\lim_{n\to\infty} z_n = 0$, then $\lim_{n\to\infty} \epsilon_n = 0$ by the Squeeze Theorem **??**.

Now as $M\epsilon_0 < 1$ by NR_3, then we see that:

$$\begin{aligned}
\lim_{n\to\infty} z_n &= \lim_{n\to\infty} (M\epsilon_0)^{2^n-1}\epsilon_0 \\
&= \epsilon_0 \lim_{n\to\infty} (M\epsilon_0)^{2^n-1} \\
&= \epsilon_0 \cdot 0 && \text{because } \lim_{n\to\infty} r^{m_n} = 0 \text{where } |r| < 1, m_n \geq n \forall n \in \mathbb{N} \\
&= 0
\end{aligned}$$

Now to show that this sequence converges quadratically we see that $\epsilon_{n+1} = \frac{1}{2}\left|\frac{f''(y_n)}{f'(x_n)}\right|\epsilon_n^2$, and therefore $\frac{\epsilon_{n+1}}{\epsilon_n^2} = \frac{1}{2}\left|\frac{f''(y_n)}{f'(x_n)}\right|$.

Because $|x^* - y_n| < |x^* - x_n|$ and $\lim_{n\to\infty} x_n = x^*$, then it follows that $\lim_{n\to\infty} y_n = x^*$. Therefore we see that

$$\lim_{n\to\infty} \frac{\epsilon_{n+1}}{\epsilon_n} = \frac{1}{2}\left|\frac{f''(x^*)}{f'(x^*)}\right| \in \mathbb{R}^+$$

Hence as the above limit exists and is positive then the sequence is quadratically convergent. □

2.4 Efficiency Metrics

Now that we have discussed how to measure the accuracy of our results by their errors, we wish to consider the efficiency method. There is typically a trade-off between accuracy and efficiency in that to gain a more accurate result, more calculations are required thus taking up more resources. In general though we will be using efficiency metrics to compare how efficient two different algorithms are at getting the same result.

There are two main ways in which we will measure the efficiency of an algorithm. The first of these methods is the theoretical complexity of the algorithm, which represents the number of steps/operations an algorithm needs to achieve it's goal. The complexity of an algorithm is denoted by the big O notation, which represents the order of the complexity, i.e. the highest order term in the number of operations required.

Typically the execution of an algorithm depends on the size of the input and so if we consider that an input has size n we can discuss different complexities. The first consideration is that if one algorithm takes $2n$ operations while another takes $20n$ operations, then both algorithms have a complexity of $\mathcal{O}(n)$.

A complexity of $\mathcal{O}(n)$ is not a bad complexity for an algorithm as the number of operations needed rises linearly with the size of the input. Complexities of $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$ and $\mathcal{O}(n!)$ are all poor complexities for an algorithm with the latter two becoming infeasible for larger n. On the other hand complexities better than $\mathcal{O}(n)$ include $\mathcal{O}(\log(n))$ and $\mathcal{O}(1)$, the latter of these is particularly significant as it means that the algorithm takes the same number of steps regardless of the size of the input.

The second method of assessing efficiency is timing of functions during execution. This method directly observes how long it takes a computer to perform the calculations for a given algorithm and can be used to empirically test the speed of two algorithms. One remark is that due to the speed of modern computers it is infeasible to time the execution of a single function, and one typically times the same algorithm with the same input being calculated multiple times to get accurate and measurable timings.

3 Root Functions

Root functions are a vital part of mathematics and have been used for millennia, originally studied for their useful relation to architecture root functions also have many modern day applications. The majority of this section will be dealing with the commonly used square root function $\sqrt{N}$, which always gives an irrational answer if N is not a square number.

We will consider several methods for approximating root functions, but for our purposes here we are only going to consider roots of $N \in \mathbb{R}_0^+$, this is because if $N \in \mathbb{R}^-$ then it follows that $\sqrt{N} = i\sqrt{|N|}$.

3.1 Digit by Digit Method

The first method we will examine is an old method, that has been observed in Babylonian Mathematics over 2000 years ago, which is used to accurately generate the square root of numbers one digit at a time. This method differs from others discussed as it generates each digit of the root with perfect accuracy, one at a time, thus in a theoretical sense this algorithm is the most accurate of the methods we will view; we will see however that this method is slow.

Now suppose we are looking for $\sqrt{N}$, then we know that $\sqrt{N} = a_0 10^n + a_1 10^{n-1} + a_2 10^{n-2} + \dots$ for some $n \in \mathbb{Z}$; it then follows that $N = (a_0 10^n + a_1 10^{n-1} + a_2 10^{n-1} + \dots)^2$. By expanding the quadratic value we get that

$$N = a_0^2 10^{2n} + (20a_0 + a_1)a_1 10^{2n-2} + (20(a_0 10 + a_1) + a_2)a_2 10^{2n-4} + \dots + (20\sum_{i=0}^{k-1} a_i 10^{k-i-1} + a_k)a_k 10^{2n-2k}$$

An observation should be made regarding the value of n that we use for the theorem. We could of course try different values of n, in some structured procedure, that will find the largest n such that $10^n \leq N$. However we can note that $log_{10}(\sqrt{N}) = \frac{1}{2}log_{10}(N)$, thus $10^{\frac{1}{2}log_{10}(N)} = \sqrt{N}$. Using this information, and the fact that $n \in \mathbb{Z}$, we can have $n := \left\lfloor \frac{1}{2}log_{10}(N) \right\rfloor$.

This allows us to get successive approximations of N where $N_0 = a_0^2 10^{2n}$, $N_1 = N_0 + (20a_0 + a_1)a_1 10^{2n-2}$, $N_2 = N_1 + (20(a_0 10 + a_1) + a_2)a_2 10^{2n-4}$. This will allow us to create an algorithm that will give successive approximations of $sqrtN = a_0 10^n + a_1 10^{n-1} + \dots$, more importantly each approximation will give us the exact next digit in the decimal representation of $\sqrt{N}$.

Thus we can have an iterative method to solve the problem, where at each stage we are trying to find the largest digit which satisfies the inequality $(20\sum_{i=0}^{k-1} a_i 10^{k-i-1} + a_k)a_k 10^{2n-2k} \leq N - N_{k-1}$. Thus we get the following pseudo-code, which outputs two sequences, one indicating the digits before the decimal point and one afterwards. I will use set notation to indicate the sequences, but in this case order is important and repetition is allowed.

Algorithm 3.1.1: Exact Digit by Digits Square Root

```
exactRootDigits($N \in \mathbb{R}_0^+, d \in \mathbb{N}$):
    $Digits_a := \emptyset$
    $Digits_b := \emptyset$
    $k := 0$
    $n := \left\lfloor \frac{1}{2}log_{10}(N) \right\rfloor$
    while $k < d$:
        $a_k := \max\left\{t \in [0,9] \cap \mathbb{Z} : \left(20\sum_{i=0}^{k-1} a_i 10^{k-i-1} + t\right) t 10^{2n-2k} \leq N\right\}$
        $N \mapsto N - \left(20\sum_{i=0}^{k-1} a_i 10^{k-i-1} + a_k\right) a_k 10^{2n-2k}$
        if $n - k < 0$:
            $Digits_b \mapsto Digits_b \cup \{a_k\}$
        else:
            $Digits_a \mapsto Digits_a \cup \{a_k\}$
        $k \mapsto k + 1$
    if $Digits_a = \emptyset$:
        $Digits_a := \{0\}$
    if $Digits_b = \emptyset$:
        $Digits_b := \{0\}$
    return $(Digits_a, Digits_b)$
```

This method has a computational complexity of $\mathcal{O}(d^2)$, as each loop requires the operations of summing k elements, and the loop is repeated for $k = 0 \to d$. We will see that by considering some changes to the algorithm we can change the complexity class to be $\mathcal{O}(d)$.

First we will note that line 5 is not an issue, as if we only care about the first significant digit of $\frac{1}{2}log_{10}(N)$, then this is $\mathcal{O}(|log(N)|)$. This can be seen as if we start from $n = 0$ we can either count up or down until a we find 10^{2n} at most or at least N, respectively. This obviously takes at most $|log_{10}(N)|$ steps, giving us our stated complexity. We will also assume that $\mathcal{O}(|log(N)|) \leq \mathcal{O}(d)$, as we have already seen that we can manipulate our input N to be within a reasonable range.
Second we note that on line 7 we calculate $\sum_{i=0}^{k-1} a_i 10^{k-i-1}$ for each value of t; we can reduce the complexity of this line by pre-calculating this value. However we can do even better if we consider that at step $k+1$ we are calculating $\sum_{i=0}^{k} a_i 10^{k-i} = a_k + 10\sum_{i=0}^{k-1} a_i 10^{k-i-1}$. Thus if we introduce $P_0 := 0$, and fore each k we calculate $P_{k+1} := 10P_k + a_k$, then we can reduce the complexity from $\mathcal{O}(k)$ to $\mathcal{O}(1)$.

This calculation of P_k, then carries over to reduce the complexity of line 8 to be $\mathcal{O}(1)$ instead of $\mathcal{O}(k)$. Combining this we can create the modified algorithm below:

Algorithm 3.1.2: Exact Digit by Digits Square Root version 2

```
1   exactRootDigits_v2($N \in \mathbb{R}_0^+, d \in \mathbb{N}$):
2       $Digits_a := \emptyset$
3       $Digits_b := \emptyset$
4       $k := 0$
5       $n := \left\lfloor \frac{1}{2}log_{10}(N) \right\rfloor$
6       $P_0 := 0$
7       while $k < d$:
8           $a_k := \max\left\{t \in [0,9] \cap \mathbb{Z} : (20P_k + t)\, t 10^{2n-2k} \leq N\right\}$
9           $N \mapsto N - (20P_k + a_k)\, a_k 10^{2n-2k}$
10          $P_{k+1} := 10P_k + a_k$
11          if $n - k < 0$:
12              $Digits_b \mapsto Digits_b \cup \{a_k\}$
13          else:
14              $Digits_a \mapsto Digits_a \cup \{a_k\}$
15          $k \mapsto k + 1$
16      if $Digits_a = \emptyset$:
17          $Digits_a := \{0\}$
18      if $Digits_b = \emptyset$:
19          $Digits_b := \{0\}$
20      return $(Digits_a, Digits_b)$
```

This method is useful, but can be difficult to implement as it requires high precision for the representation of the real value of N. In my implementation using C, I utilised the MPFR library to utilise high precision integers, but still encountered issues regarding loss of precision.

As an example the table below shows the number of digits of accuracy I was able to calculate for $\sqrt{2}$ using the above algorithm, compared to the number of bits of precision used in the calculations.

Bits of Precision	Maximum Accuracy
8	2
16	5
32	9
64	18
128	39
256	77
512	154
1024	308
2048	615
4096	1234
8192	2466

This data is highly structured and so we can hope to create a simple function that would allow us to calculate how much precision would be needed for a given number of digits of accuracy, at least for single digit inputs for N. We can see that the average ratio of Precision to Accuracy is 3.41259..., which ranges from 3.31928... to 4.0. From this we can draw a general trend that Digits of Accuracy $\approx$ 3.4 $\times$ Bits of Precision; thus if we take the more generous assumption that Digits of Accuracy 4 $\times$ Bits of Precision, we can use this to pre-determine the accuracy needed.

It should be noted that to ensure accuracy we should over-estimate the required precision, however if we overestimate the precision, then our calculations will be performed using unnecessarily large data structures and thus computation time will increase.

One particular use of this technique is to find an approximation of a square root to it's integer part, calculated in base 2. This algorithm is of note as we will see that it has a computation time of (1).

The algorithm uses the same basis as the base 10 version, for it's calculations, but due to the nature of being in binary several changes can be made for computational efficiency. To do this we will view the problem as follows: if we know some $r \in \mathbb{Z}_0^+$ which is our current approximation of our root, we are looking for some $e \in \mathbb{Z}_0^+$ such that $(r+e)^2 \leq N$. Expanding this out we get $r^2+2re+e^2 \leq N$, and if we keep track of $M = N-r^2$, we can test if $2re+e^2 \leq M$.

Now we can consider our choice of e, the most practical method is to test successive $e_m := 2^m$, where m is descending starting with $m = \max m \in \mathbb{Z}_0^+ : 4^m \leq N$. We can use an iterative formula to build up the integer square root, where we start with $r = 0, M = N$ and have $rr+e_m$ whenever $2re_m+e_m^2 \leq M$, stopping when $m < 0$. This is then implemented as follows:

Algorithm 3.1.3: Integer Square Root Algorithm

```
integerSquareRoot($N \in \mathbb{Z}_0^+$):
    $M := N$
    $m := \max m \in \mathbb{Z}_0^+ : 4^m \leq M$
    $r := 0$
    while $m \geq 0$:
        if $2r(2^m) + 4^m \leq M$:
```

```
7               $M \mapsto M - 2r(2^m) + 4^m$
8               $r \mapsto r + 2^m$
9           $m \mapsto m - 1$
10      return $r$
```

If we now consider an implementation of the above algorithm using an unsigned integer system with K bits, where $2|K$. We will use `res` to represent $2re_m$, which means at the start of the algorithm we will have `res = 0`; similarly we can use `bit` to represent e_m^2. As we know that K bits are used and $2|K$, it then follows that the largest power of 4 less than the maximum representable value ($2^K - 1$ is 2^{K-2}, which can be calculated as `bit = 1 << (K - 2)` using bit shift operations. Finally we will use `num` to represent M.

Now that we have discussed the set-up we can consider how to implement some of the steps above. First to implement line 3 we can simply keep dividing `bit` by 4 while `bit > num`, which can be efficiently implemented as `bit >> 2` by using bit shifts in place of division by powers of 2. The same technique can be used in place of line 9, which leads us to re-evaluating our usage of line 5. As we are using bit shifting and a bit shift that would take a number past 0 instead results in 0, we also know that $2|K$ and so eventually we will reach `bit == 1`, which represents $m = 0$; therefore we can use `bit > 0` as our stopping criteria on line 5.

Line 6 is easy to convert, given our definitions of `res`, `bit` and {num, as is line 7. All that remains is to consider how to update `res`, which has two different ways of being updated depending on whether `res + bit <= num`. If it is false that `res + bit <= num`, then we wish for `res` to represent $2re_{m-1}$; this is easily achieved if we consider that $2re_{m-1} = \frac{1}{2}(2re_m)$, which prompts the update `res = res >> 1`. For the second case, when `res + bit <= num` is true, we want `res` to represent $2(r + e_m)e_{m-1}$; to implement this we consider the following derivation:

$$\begin{aligned}
2(r + e_m)e_{m-1} &= \frac{1}{2} \cdot 2(r + e_m)e_m \\
&= \frac{1}{2} \cdot 2(re_m + e_m^2) \\
&= \frac{1}{2}(2re_m) + e_m^2
\end{aligned}$$

Using this above derivation we see that we can calculate this as `res = (res >> 1) + bit`. Below is a simple implementation of this in C using the unsigned 32 bit integer type `uint32_t`. A more commented and slightly modified version can be found in Appendix **??**, File **??**.

```c
uint32_t int_sqrt(uint32_t num)
{
        uint32_t res = 0, bit = (1 << 30);

        while (bit > num)
                bit = bit >> 2;

        while (bit > 0)
        {
                if (res + bit <= num)
                {
                        num = num - (res + bit);
                        res = (res >> 1) + bit;
                }
                else
                        res = res >> 1;

                bit = bit >> 2;
        }

        return res;
}
```

We should consider the final step of the loop, when `bit == 1`. In this case when `res` is updated we have `res` represent either $2(r+e_0)e_{-1} = r+e_0$, or $2re_{-1} = r$; thus the algorithm exits with the correct value.

Now that the algorithm is correctly constructed using simple unsigned integer addition, subtraction and bit shifting (which we can assume all have computational time of $\mathcal{O}1$), we can look at the worst case complexity of the algorithm:

- The complexity of the set up of variables is constant time.
- The worst case complexity would be to to have `bit <= num` at the start.
- The loop would execute 16 times for our 32 bit integers, and contains a single operation which is $\mathcal{O}(1)$ complexity.
  - The worst case within the loop is to have `res + bit <= num` for each iteration.
  - Within the first `if` branch there are a constant 4 operations.
  - Each loop has an additional operation operation to update `bit`.
  - This makes 5 operations per loop, giving $\mathcal{O}(1)$ complexity within the loops.

Therefore we see that the algorithm has $\mathcal{O}(1)$ time complexity, and even has the same in storage complexity. In particular our 32 bit example requires 163 operations, including assignments, comparisons and calculations. This means that the integer square root of any number up to 4294967295 can be calculated extremely quickly.

3.2 Bisection Method

The Bisection Method is a general method for approximating the zero, α, of a function, f, on a bounded interval, $I := [a, b]$, where f has the property $f(x)f(y) < 0 \forall (x, y) \in [a, \alpha) \times (\alpha, b]$; we may assume, without loss of generality, that $f(x) < 0 \forall x \in [a, \alpha]$.

The bisection method starts with initial bounds $a_0 = a, b_0 = b$, where the initial approximation for the root is $x_0 = \frac{1}{2}(a+b)$. We will consider pseudo-code of the iteration process, that uses $b_n - a_a < \tau$ or $f(x_n) = 0$ as exit criteria. Here τ is a tolerance threshold, and if the exit criteria is met it means that $|x_n - \alpha| \leq \frac{\tau}{2}$, while the other exit criteria means we have reached an exact solution.

Algorithm 3.2.1: General Bisection Method

```
bisectionMethod($a \in \mathbb{R}, b \in (a, \infty), f \in \mathcal{C}[a, b], \tau \in \mathbb{R}^+$)
    $a_0 := a$
    $b_0 := b$
    $x_0 := \frac{1}{2}(a + b)$
        $n := 0$
        while $f(x_n) \neq 0$ AND $b_n - a_n > \tau$:
        if $f(x_n) < 0$:
            $a_{n+1} := x_n$
            $b_{n+1} := b_n$
        else:
            $a_{n+1} := a_n$
            $b_{n+1} := x_n$
        $n \mapsto n + 1$
        $x_n := \frac{1}{2}(a_n + b_n)$
    return $x_n$
```

For our purposes we are trying to find the zero of $f(x) = x^2 - N$, which is a strictly increasing function on $\mathbb{R}_0^+$. If $N >= 1$, then $\sqrt{N} \in [0, N]$, while $N < 1 \implies \sqrt{N} \in [0, 1]$. It is obvious that our function has the required property, and thus we get the following method for finding the square root of N:

Algorithm 3.2.2: Bisection Method for Square Roots

```
bisectionSquareRoot($N \in \mathbb{R}_0^+, \tau \in \mathbb{R}^+$)
    $a_0 := 0$
    $b_0 := \max 1, N$
    $x_0 := \frac{1}{2}(a_0 + b_0)$
    $n := 0$
    while $x_n^2 - N \neq 0$ AND $b_n - a_n > \tau$:
        if $x_n^2 - N < 0$:
            $a_{n+1} := x_n$
            $b_{n+1} := b_n$
        else:
            $a_{n+1} := a_n$
            $b_{n+1} := x_n$
        $n \mapsto n + 1$
        $x_n := \frac{1}{2}(a_n + b_n)$
    return $x_n$
```

The implementation of this method is efficiently achieved in C using only addition, subtraction and multiplication by a constant. Before this method is implemented, however, we must first consider if and or when it converges to the correct answer. From an intuitive standpoint we would assume that if there is only one root in the interval, it would follow that we would converge to the root.

Proposition 3.2.1. $\lim_{n\to\infty} x_n = \sqrt{N}$ *for Algorithm 3.2.2*

Proof. To prove this statement it suffices to prove that $\sqrt{N} \in [a_n, b_n] \forall n \in \mathbb{N}$ and $\lim_{n\to\infty} |x_n - \sqrt{N}| = 0$.

Claim 1: $\sqrt{N} \in [a_n, b_n] \forall n \in \mathbb{N}$

Proof. $a_0 := 0 \implies a_0 \leq \sqrt{N}$
$b_0 := \max\{1, N\} \implies b_0 \geq \sqrt{N}$

Therefore it is obvious that $\sqrt{N} \in [a_0, b_0]$
Now suppose $\sqrt{N} \in [a_n, b_n]$ for some $n \in \mathbb{N}$
It should be noted that $a_n, b_n, x_n \in \mathbb{R}_0^+ \forall n \in \mathbb{N}$ as $a_0, b_0 \in \mathbb{R}_0^+$ and all the subsequent values are derived from these using only addition and multiplication by positive factors.
We then see that $x_n := \frac{1}{2}(a_n + b_n)$, and we consider the two cases that $x_n^2 - N \leq 0$ or $x_n^2 - N \geq 0$.

Case $x_n^2 - N \leq 0$**:** $a_{n+1} := x_n, b_{n+1} := b_n$
It is therefore obvious that $\sqrt{N} \leq b_{n+1}$.
Now we see that $x_n^2 - N \leq 0 \implies x_n^2 \leq N \implies x_n \leq N$ as all the values are non-negative.
Thus $\sqrt{N} \in [a_{n+1}, b_{n+1}]$.

Case $x_n^2 - N \geq 0$**:** $a_{n+1} := a_n, b_{n+1} := x_n$
It is therefore obvious that $\sqrt{N} \geq a_{n+1}$.
Now we see that $x_n^2 - N \geq 0 \implies x_n^2 \geq N \implies x_n \geq N$ as all the values are non-negative.
Thus $\sqrt{N} \in [a_{n+1}, b_{n+1}]$.

Hence $\sqrt{N} \in [a_n, b_n] \implies \sqrt{N} \in [a_{n+1}, b_{n+1}] \forall n \in \mathbb{N}$
As $sqrtN \in [a_0, b_0]$ then we see that $\sqrt{N} \in [a_n, b_n] \forall n \in \mathbb{N}$ ■

Claim 2: $\lim_{n\to\infty} |x_n - \sqrt{N}| = 0$

Proof. Let $n \in \mathbb{N}$ be arbitrary.
As $x_n := \frac{1}{2}(a_n + b_n)$ then we see that $|a_n - x_n| = |b_n - x_n| = \frac{1}{2}(b_n - a_n)$.
Now as $\sqrt{N} \in [a_n, b_n]$ it follows that $|\sqrt{N} - x_n| \leq \frac{1}{2}(b_n - a_n$.
As the modulus function is a mapping from $\mathbb{R}$ to $\mathbb{R}_0^+$, it is clear that $|\sqrt{N} - x_n|$ is bounded below by 0.
Now as for each $n \in \mathbb{N}$, either $a_{n+1} = x_n$ or $b_{n+1} = x_n$, we see that $b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n)$.
Further we can see that $b_n - a_n \geq 0 \forall n \in \mathbb{N}$ because $b_n \geq a_n$.
Therefore the sequence of $\frac{1}{2}(b_n - a_n)$ is a strictly decreasing sequence that is bounded below, by 0. Thus $\lim_{n\to\infty} \frac{1}{2}(b_n - a_n) = 0$
Therefore $\lim_{n\to\infty} |x_n - \sqrt{N}| = \lim_{n\to\infty} \frac{1}{2}(b_n - a_n) = 0$ ■

By using our two claims above we see that $\lim_{n\to\infty} x_n = \sqrt{N}$. □

The algorithm can be generalised to search for $\sqrt{k}N$, where $k \in [2, \infty) \cap \mathbb{Z}$. We can do this by using the integer power function discussed previously in section **??**. This gives the following algorithm:

Algorithm 3.2.3: Bisection Method for General Roots

```
kRootBisectionMethod($N \in \mathbb{R}_0^+, k \in [2,\infty) \cap \mathbb{Z}, \tau \in \mathbb{R}^+$)
    $a_0 := 0$
    $b_0 := \max 1, N$
    $x_0 := \frac{1}{2}(a_0 + b_0)$
    $n := 0$
    while $\text{intPow}(x_n, k) - N \neq 0$ AND $b_n - a_n > \tau$:
            if  intPow($x_n, k$) $- N < 0$:
                $a_{n+1} := x_n$
                $b_{n+1} := b_n$
            else:
                $a_{n+1} := a_n$
                $b_{n+1} := x_n$
            $n \mapsto n + 1$
            $x_n := \frac{1}{2}(a_n + b_n)$
        return $x_n$
```

The proof that this converges to the correct value is very similar to the proof for square roots.

We can now consider the accuracy that can be achieved by our algorithm, for our purposes we will be considering $\sqrt{N}$, though the same applies for $\sqrt{kN}$. We know that $\sqrt{N} \in [a_n, b_n] \forall n \in \mathbb{N}$, and in particular we know that either $\sqrt{N} \in [a_n, x_n]$ or $\sqrt{N} \in [x_n, b_n] \forall n \in \mathbb{N}$; therefore we know that $\epsilon_n := \left| x_n - \sqrt{N} \right| \leq \frac{1}{2}(b_n - a_n) \forall n \in \mathbb{N}$. Then as we know that $b_{n+1} - a_{n+1} = \frac{1}{2}(b_n - a_n)$, we know that $\epsilon_n \leq \frac{1}{2^n}(b_0 - a_0)$.

We can consider that $\forall N \in \mathbb{R}_0^+ \exists (r, k) \in [\frac{1}{4}, 1) \times \mathbb{Z} : N = r \cdot 2^{2k}$; using this we know that $\sqrt{N} = \sqrt{r} \cdot 2^k$. As we have the fixed initial bounds of $a_0 = 0$ and $b_0 = 1$, then if we are finding $\sqrt{r}$ we know that $\epsilon_n \leq \frac{1}{2^n} \forall n \in \mathbb{N}$. Hence we can calculate the precision of our current estimate beforehand for any $n \in \mathbb{N}$, and thus we can guarantee d significant digits of accuracy for $r \in [\frac{1}{4}, 1)$.

To get this accuracy must find $n \in \mathbb{N}$ such that $\epsilon_n \leq \frac{1}{10^d}$, to achieve this we must find $n \in \mathbb{N}$ such that $2^n \geq 10^d$. For example the following table indicates the required n, required for certain significant digits of accuracy.

d	$n : 2^n \geq 10^n$
1	0
5	15
10	30
20	64
50	163
100	329

Now usually finding r and k as above would be as hard as calculating the logarithm of N; however due to the way that C stores real numbers as either `double` or in the MPFR library, finding these values are actually fairly trivial. Both provide a functionality to find $(a, b) \in [\frac{1}{2}, 1) \times \mathbb{Z} : N = a \cdot 2^b$, and from this we merely require a simple comparison and division by 2 if b is not even. This leads to the following algorithm, which has the above maximum number of iterations for a required accuracy:

Algorithm 3.2.4: Bisection Method for Square Roots with fixed bounds

```
bisectionSquareRoot($N \in \mathbb{R}_0^+, \tau \in \mathbb{R}^+$)
    Let  $(r, e) \in [\frac{1}{2}, 1) : N = r \cdot 2^e$
    if  $2 \nmid e$:
        $r \mapsto \frac{r}{2}$
        $e \mapsto e - 1$
    $a_0 := 0$
    $b_0 := 1$
    $x_0 := \frac{1}{2}(a_0 + b_0)$
    $n := 0$
    while $x_n^2 - N \neq 0$ AND $b_n - a_n > \tau$:
        if $x_n^2 - N < 0$:
            $a_{n+1} := x_n$
            $b_{n+1} := b_n$
        else:
            $a_{n+1} := a_n$
            $b_{n+1} := x_n$
        $n \mapsto n + 1$
        $x_n := \frac{1}{2}(a_n + b_n)$
    return $x_n \cdot 2^{\frac{e}{2}}$
```

3.3 Newton's Method for Square Roots

If we consider our equation $f(x) = x^2 - N$, then we can see that it is differentiable on $x \in \mathbb{R}^+$ with $f'(x) = 2x$; we can therefore hope to use the Newton-Raphson method to approximate $x^* \in \mathbb{R}^+ : f(x^*) = 0$. Now it is obvious that if $f(^*) = 0$ then $x^* = \sqrt{N}$ and so the Newton-Raphson method should converge to the $\sqrt{N}$ provided we start at a suitable x_0.

The iterative step of Newton's method for square roots is $x_{n+1} = x_n - \frac{x_n^2 - N}{2x_n}$ which when implemented in C, requires the calculation of `x = x - (x*x - N) / (2*x)` each iteration, which requires 5 operations. However if we re-arrange our equation, we instead get $x_{n+1} = \frac{1}{2}x_n + \frac{N}{x}$, which when implements is `x = 0.5 * (x + N/x)`, which now uses only 3 operations.

We can then use the following pseudo-code as the basis of our implementations of the Newton-Raphson Method for Square Roots:

Algorithm 3.3.1: Basic Newton Method for Square Root

```
NewtonSquareRoot($N \in \mathbb{R}, x_0 \in \mathbb{R}, \tau \in (0,1)$):
    $n := 0$
    loop:
        $x_{n+1} := \frac{1}{2}(x_n + \frac{N}{x_n})$
        $\delta_n := |x_{n+1} - x_n|$
        if $\delta_n \leq \tau$:
            return $x_{n+1}$
        $n \mapsto n + 1$
```

Next we want to consider our initial estimate x_0; it is prudent to first consider when our initial estimate will converge to the correct root. By looking at a graph of the function, and in particular the tangents to the curve, it would seem reasonable to wonder if $\lim_{n\to\infty} x_n = \sqrt{N}$.

Proposition 3.3.1. *If $x_0 \in \sqrt{N}, \infty$ and $\{x_n : n \in \mathbb{N}\}$ is a sequence of approximations of $\sqrt{N}$ found via the Newton-Raphson Method, as detailed above, then:*

$$\lim_{n\to\infty} x_n = \sqrt{N}$$

Proof. Suppose $x_n > \sqrt{N}$, then

$$\begin{aligned}
x_{n+1} &= \frac{1}{2}\left(x_n + \frac{N}{x_n}\right) \\
&< \frac{1}{2}\left(x_n + \frac{N}{\sqrt{N}}\right) && \text{as}\sqrt{N} < x_n \implies \frac{1}{x_n} < \frac{1}{\sqrt{N}} \\
&= \frac{1}{2}\left(x_n + \sqrt{N}\right) \\
&< \frac{1}{2}(2x_n) \\
&= x_n
\end{aligned}$$

Therefore we see that $\{x_k : k \in [n, \infty) \cap \mathbb{Z}\}$ is a strictly decreasing sequence.
Now suppose that $x_n \geq \sqrt{N}$ and then, for a contradiction, assume that $x_{n+1} < \sqrt{N}$. We then see that:

$$\begin{aligned}
&\frac{1}{2}\left(x_n + \frac{N}{x_n}\right) < \sqrt{N} \\
\implies &x_n + \frac{N}{x_n} < 2\sqrt{N} \\
\implies &x_n^2 + N < 2\sqrt{N}x_n \\
\implies &x_n^2 - 2\sqrt{N}x_n + N < 0 \\
\implies &\left(x_n - \sqrt{N}\right)^2 < 0
\end{aligned}$$

This is a contradiction as $x_n, \sqrt{n} \in \mathbb{R} \implies \left(x_n - \sqrt{N}\right)^2 \geq 0$.
Therefore $x_n \geq \sqrt{N} \implies x_{n+1} \geq \sqrt{N}$.
Hence if $x_0 > \sqrt{N}$, then it follows that $\{x_n : n \in \mathbb{N}\}$ is a strictly decreasing sequence that is bounded below. Therefore by an elementary result from limit theory, we see that $\lim_{n\to} x_n = \inf\{x_n : n \in \mathbb{N}\}$. □

The most obvious choice for x_0 would be N, but we see that $N \in (0, 1)$, then $N < \sqrt{N}$. In this case, we could choose $x_0 = 1$ for the case that $N \in (0, 1)$. Therefore we can choose

$$x_0 := \begin{cases} N & : \quad N \in (1, \infty) \\ 1 & : \quad N \in (0, 1) \end{cases}$$

In our choice of x_0, we have so far left out the cases where $N \in \{0, 1$. In both of these case we already know the correct answer, namely $\sqrt{N} = N$ provided $N \in 0, 1$. Therefore we can exclude them from our calculations, as we can pre-asses the value of N, simply returning the correct answer if one of these cases is encountered.

This then leads to an updated version of the above pseudo-code:

Algorithm 3.3.2: Basic Newton Method for Square Root

```
NewtonSquareRoot( $N \in \mathbb{R}_0^+, \tau \in (0,1)$ ):
```

```
if $N \in \{0,1\}$:
    return $N$
if $N > 1$:
    $x_0 := N$
else:
    $x_0 := 1$
$n := 0$
loop:
    $x_{n+1} := \frac{1}{2}(x_n + \frac{N}{x_n})$
    $\delta_n := |x_{n+1} - x_n|$
    if $\delta_n \leq \tau$:
        return $x_{n+1}$
    $n \mapsto n+1$
```

An alternative would be to use the integer square root method discussed in Section 3.1 to improve our initial choice of x_0. We will start by showing, that for intervals $I \subset \mathbb{R}^+$, the first two criteria for quadratic convergence of the Newton Raphson method are met.

Proposition 3.3.2. *If $I \subset \mathbb{R}^+$ then NR_1 and NR_2 are satisfied for $f(x) = x^2 - N$*

Proof. $f(x) = x^2 - N \implies f'(x) = 2x \implies f''(x) = 2$
Now as $x \in \mathbb{R}^+ \forall x \in I$, then it is obvious that $f'(x) > 0$
Therefore $f'(x) \neq 0 \forall x \in I$, and so NR_1 is satisfied.
As $f''(x)$ is a constant function, then it is continuous on all of $\mathbb{R}$.
Hence $f''(x)$ is continuous $\forall x \in I$ and so NR_2 is satisfied. $\square$

Now the integer square root function will always produce a root that is at most a distance of 1 from $\sqrt{N}$; therefore we can consider $I = [\sqrt{N} - 1, \sqrt{N} + 1]$. Now if $N \leq 1$, then $I \mathbb{R}^+$ and so we cannot guarantee the satisfaction of NR_1. Therefore we can proceed with our analysis of the case that $N > 1$.

If $N > 1$ we need to find when we can satisfy NR_3. First, we remember that $M := \sup \left| \frac{f''(x)}{f'(x)} \right| : x \in I$ and $\epsilon_0 := \left| x_0 - \sqrt{(N)} \right|$. Then to satisfy NR_3, we must have that $M\epsilon_0 < 1$.

We can guarantee that $\epsilon_0 \leq 1$ because $x_0 \in I$ from the integer square root algorithm; therefore it suffices to find the situation where $M < 1$. As both f' and f'' are continuous and non-zero on I it follows that $M = \sup x^{-1} : x \in I = (\sqrt{N} - 1)^{-1}$. We then see that:

$$\begin{aligned} M < 1 &\iff \sqrt{N} - 1 > 1 \\ &\iff \sqrt{N} > 2 \\ &\iff N > 4 \end{aligned}$$

Therefore we can get the following new choice for x_0, and thus new pseudo-code:

$$x_0 := \begin{cases} 1 & : \quad N \in (0,1) \\ N & : \quad N \in (1,4] \\ intSqrt(N) & : \quad N \in (4,\infty) \end{cases}$$

Algorithm 3.3.3: Basic Newton Method for Square Root

```
NewtonSquareRoot($N \in \mathbb{R}_0^+, \tau \in (0,1)$):
    if $N \in \{0,1\}$:
        return $N$
    if $N < 1$:
        $x_0 := 1$
    else:
        if $N \leq 4$:
            $x_0 := N$
        else:
            $x_0 :=$ IntSqrt($N$)
    $n := 0$
    loop:
        $x_{n+1} := \frac{1}{2}(x_n + \frac{N}{x_n})$
        $\delta_n := |x_{n+1} - x_n|$
        if $\delta_n \leq \tau$:
            return $x_{n+1}$
        $n \mapsto n + 1$
```

If we consider any $N \in \mathbb{R}_0^+$, then $\exists a \in \left[\frac{1}{2}, 1\right), b \in \mathbb{Z} : N = a \times 2^b$. Finding this value would be a hard as finding the logarithm of N base 2, but due to the representation of numbers within C, both standard C and MPFR have functions that allow us to extract these two values with minimal computational expenditure.

This helps as we can then narrow our problem, to only finding $\sqrt{a} : a \in \left[\frac{1}{2}, 1\right)$, and then calculating

$$\sqrt{N} = \sqrt{a} \times 2^{\left\lfloor \frac{b}{2} \right\rfloor} \times \alpha \text{ where } \alpha = \begin{cases} 1 & : \ b \in 2\mathbb{Z} \\ \sqrt{2} & : \ b \in \mathbb{Z}^+ \setminus 2\mathbb{Z} \\ \frac{1}{sqrt2} & : \ b \in \mathbb{Z}^- \setminus \mathbb{Z} \end{cases}$$

We then get the following algorithm, which implements this:

Algorithm 3.3.4: Newton Method for Square Root v3

```
NewtonSquareRoot($N \in \mathbb{R}_0^+, \tau \in (0,1)$):
    Let $(a,b) :\in \left[\frac{1}{2}, 1\right) \times \mathbb{Z}$ s.t. $N = a * 2^b$
    $x_0 := 1$
    if $b \equiv 0 \bmod 2$:
        $\alpha := 1$
    else:
        if $b > 0$:
            $\alpha := \sqrt{2}$
        else:
            $\alpha := \frac{1}{\sqrt{2}}$
    $n := 0$
    loop:
        $x_{n+1} := \frac{1}{2}(x_n + \frac{a}{x_n})$
        $\delta_n := |x_{n+1} - x_n|$
        if $\delta_n \leq \tau$:
            return $\alpha \cdot x_{n+1} \cdot 2^{\left\lfloor \frac{b}{2} \right\rceil}$
        $n \mapsto n + 1$
```

We must first consider the fact that the algorithm requires the pre-calculation of both $\sqrt{2}$ and $\frac{1}{\sqrt{2}}$, to be able to calculate all values. However, it turns out we can use the algorithm itself to generate these values as $2 = \frac{1}{2} \cdot 2^2$, and as the exponent of 2 is even then the algorithm does not require $\sqrt{2}$ for this computation. Similarly $\frac{1}{2} = \frac{1}{2} \cdot 2^0$, which again is an even exponent. We can thus run our algorithm to find an arbitrarily accurate values for $\sqrt{2}$ and $\frac{1}{\sqrt{2}}$ to allow us to run the algorithm for other values.

With this observation can then consider $N \in \left[\frac{1}{2}, 1\right)$. As this is a small range and, as per our previous algorithm, we use an initial guess of $x_0 = 1$, then we can prove that our algorithm will converge quadratically to $\sqrt{N}$.

Proposition 3.3.3. *Algorithm 3.3.4, satisfies the criteria of Theorem 2.3.2, and thus has quadratic convergence to $\sqrt{N}$.*

Proof. To fulfil the criteria of Theorem 2.3.2, we must find and interval $I := [\sqrt{N}-r, \sqrt{N}+r]$ for some $r \geq \epsilon_0$.

Consider $\epsilon_0 = |\sqrt{N} - x_0| = 1 - \sqrt{N}$. We see that as $N \geq \frac{1}{2}$ then $\sqrt{N} \geq \sqrt{2}^{-}1$, and thus $\epsilon_0 \leq 1 - \sqrt{2}^{-}1$. Let us have $r := 1 - \frac{1}{\sqrt{2}}$, and I as defined above.

If we look at the lower bound of I, then we see that:

$$\begin{aligned}
\sqrt{N} - r &\geq \frac{1}{\sqrt{2}} - (1 - \frac{1}{\sqrt{2}}) \\
&= \frac{2}{\sqrt{2}} - 1 \\
&= \sqrt{2} - 1 \\
&> 0
\end{aligned}$$

Therefore we see that $I \subset \mathbb{R}^+$, and so by Proposition 3.3.2 we get that NR_1 and NR_2 are satisfied. It then remains to show that NR_3 is satisfied on I.

Now by the definition in Theorem 2.3.2, we have that $M = \sup\left\{\frac{1}{2}\left|\frac{f''(x)}{f'(y)}\right| : x, y \in I\right\}$. We know that I is bounded, $f''(x) = 2$ and $f'(x) = 2x$ meaning that $\frac{1}{2}\left|\frac{f''(x)}{f'(y)}\right| = \frac{1}{f'(x)}$ as $x \in \mathbb{R}^+$.

Therefore our problem is reduced to finding $\max\left\{\frac{1}{2x} : x \in I\right\}$, which is equivalent to finding $\min\{x : x \in I\} = \sqrt{N} - r$. Therefore by passing this information back up the chain we get that

$$M = \frac{1}{2(\sqrt{N} - r)}$$

Then we see that:

$$\begin{aligned}
M\epsilon_0 &= \frac{1-\sqrt{N}}{2(\sqrt{N}-r)} \\
&\leq \frac{1-\frac{1}{\sqrt{2}}}{2(\sqrt{N}-r)} \qquad \text{as } \sqrt{N} \geq \frac{1}{\sqrt{2}} \\
&\leq \frac{1-\frac{1}{\sqrt{2}}}{2(\frac{1}{\sqrt{2}}-r)} \qquad \text{as } \sqrt{N} \geq \frac{1}{\sqrt{2}} \\
&= \frac{1-\frac{1}{\sqrt{2}}}{2(\frac{2}{\sqrt{2}}-1)} \\
&= \frac{1-\frac{1}{\sqrt{2}}}{2\sqrt{2}(1-\frac{1}{\sqrt{2}})} \\
&= \frac{1}{2\sqrt{2}} \\
&< 1 \qquad\qquad \text{as } 2\sqrt{2} > 1
\end{aligned}$$

As we have confirmed that $M\epsilon_0 < 1$, then we have confirmed that NR_3 is satisfied on I, and so the algorithm converges quadratically to the desired root. □

Using the previous proposition we can, similar to our previous methods, consider how many iterations would be needed to reach a required tolerance. To start we consider that, as mentioned in the proof or Theorem 2.3.2, that $\epsilon_n \leq (M\epsilon_0)^{2^n-1}\epsilon_0$.

We know that $M\epsilon_0 \leq \frac{1}{2\sqrt{2}}$ and that $\epsilon_0 \leq 1 - \frac{1}{\sqrt{2}}$, giving:

$$\epsilon_n \leq \left(\frac{1}{2\sqrt{2}}\right)^{2^n-1}\left(1-\frac{1}{\sqrt{2}}\right)$$

Thus if we want to achieve a tolerance of $\epsilon_n \leq \tau$, then it suffices to find $n \in \mathbb{N}_0$ such that:

$$\left(\frac{1}{2\sqrt{2}}\right)^{2^n-1} \leq \tau$$

Then,

$$(2^n-1)\log\left(\frac{1}{2\sqrt{2}}\right) \leq \log\left(\frac{\tau}{1-\frac{1}{\sqrt{2}}}\right)$$

By noting that $\log(\frac{1}{a}) = -\log(a)$, then we get

$$(1-2^n)\log(2\sqrt{2}) \leq \log\left(\frac{\tau}{1-\frac{1}{\sqrt{2}}}\right)$$

Once this is rearranged we get the following inequality:

$$2^n \geq \frac{\log\left(\frac{2(\sqrt{2}-1)}{\tau}\right)}{\log(2\sqrt{2})}$$

By taking logarithms again and re-arranging we get that

$$n \geq \frac{\log\left(\frac{\log\left(\frac{2(\sqrt{2}-1)}{\tau}\right)}{\log(2\sqrt{2})}\right)}{\log(2)} = \log_2\left(\log_{2\sqrt{2}}\left(2\frac{\sqrt{2}-1}{\tau}\right)\right)$$

Now for an example, suppose we want to know how many iterations we need to perform to find $\sqrt{N}$ to within 10 decimal places, i.e. $\tau = 10^-10 = 0.0000000001$. We remember that $\sqrt{N} \in [\frac{1}{2}, 1)$, and then we will apply transformations to this value afterwards, therefore this is equivalent to finding 10 significant digits of accuracy for our square root (ignoring any loss of accuracy that may arise from multiplications afterwards).

Now in this case we want to find $n \in \mathbb{N}$ such that $n \geq log_2(log_{2\sqrt{2}}(2 \cdot 10^{10}(\sqrt{2}-1)))$. Using Wolfram Alpha to calculate this value we get that we need $n \geq 4.457144...$ and so we can take $n = 5$. This means that we could modify our algorithm and implementation to do 5 fixed iterations of Newton's Method to guarantee at least 10 decimal places of accuracy.

In terms of efficiency versus accuracy trade-off modifying the problem thus would improve it's efficiency by removing, now unnecessary, calculation and comparison of δ_n at each stage. However this does need a fixed guaranteed accuracy, and therefore such a program would no longer be suitable if we needed to calculate a square root accurate to 15 decimal places.

Below is a table that lists the minimum $n \in \mathbb{N}$ such that n satisfies our inequality, where our tolerance is 10^k for some $k \in \mathbb{N}$. This will give us the maximum number of iterations that must be performed for the required accuracy.

$k : \tau = 10^k$	n
5	4
10	5
100	8
1,000	12
1,000,000	22

3.4 Newton's Inverse Square Root Method

While the Newton's method discussed in the previous section is good, it has a small issue when it comes to performance, namely that division is slow for a computer to perform compared to multiplication. With this knowledge we would like to find a way of utilising Newton's method without having to perform any division operations.

If we consider $f(x) = N - \frac{1}{x^2}$ then if x^* is a solution to $f(x) = 0$ we see that $x^* = \frac{1}{\sqrt{N}}$. As $f'(x) = \frac{2}{x^3}$, then the Newton's Method, will give

$$x_{n+1} = x_n - \frac{N - \frac{1}{x_n^2}}{\frac{2}{x_n^3}} = x_n\left(\frac{3}{2} - \frac{N}{2}x_n^2\right)$$

where x_0 is a given initial guess. As can be seen this algorithm requires no division if we multiply by real constants rather than the division implied above.

We can then consider that, similar to Algorithm 3.3.4, any N can be represented as $a \cdot 2^b$ where $a \in \left[\frac{1}{2}, 1\right)$. This will, again allow us to narrow our problem to a known range of values, by using the following transformations.

$$\begin{aligned} N = a \cdot 2^b &\implies \tfrac{1}{N} = \tfrac{1}{a} \cdot 2^{-b} \\ &\implies \tfrac{1}{\sqrt{N}} = \tfrac{1}{a} \cdot 2^{\lfloor \frac{-b}{2} \rceil} \cdot \alpha \qquad \alpha := \begin{cases} 1 & : \ b \equiv 0 \mod 2 \\ \sqrt{2} & : \ b \equiv 1 \mod 2, b \in \mathbb{Z}^- \\ \frac{1}{\sqrt{2}} & : \ b \equiv 1 \mod 2, b \in \mathbb{Z}^+ \end{cases} \\ &\implies \sqrt{N} = N \cdot \tfrac{1}{\sqrt{a}} \cdot 2^{\lfloor \frac{-b}{2} \rceil} \cdot \alpha \end{aligned}$$

Therefore we only need to calculate inverse square roots for values of N in the range $[\frac{1}{2}, 1)$. Thus giving us the following algorithm:

Algorithm 3.4.1: Newton Inverse Square Root Method

```
NewtonInvSquareRoot($N \in \mathbb{R}_0^+, \tau \in (0,1)$):
    Let $(a,b) :\in \left[\frac{1}{2}, 1\right) \times \mathbb{Z}$ s.t. $N = a * 2^b$
    $x_0 := 1$
    if $b \equiv 0 \mathrm{mod}\ 2$:
        $\alpha := 1$
    else:
        if $b > 0$:
            $\alpha := \frac{1}{\sqrt{2}}$
        else:
            $\alpha := \sqrt{2}$
    $n := 0$
    loop:
        $x_{n+1} := x_n(\frac{3}{2} + \frac{a}{2}x_n^2)$
        $\delta_n := |x_{n+1} - x_n|$
        if $\delta_n \leq \tau$:
            return $N \cdot \alpha \cdot x_{n+1} \cdot 2^{\lfloor \frac{-b}{2} \rceil}$
        $n \mapsto n + 1$
```

With this method we can once again consider it's convergence properties, in particular does it satisfy the criteria for quadratic convergence in Theorem 2.3.2.

Proposition 3.4.1. *Algorithm 3.4.1 satisfies the criteria of Theorem 2.3.2, and thus has quadratic convergence to* $\sqrt{N}$.

Proof. We know that we only need to consider $N \in [\frac{1}{2}, 1)$, and therefore $\sqrt{N}^{-1} \in (1, \sqrt{2}]$. Also $x_0 = 1$ and so we see that

$$\epsilon_0 = |x_0 - \sqrt{N}^{-1}| = \sqrt{N}^{-1} - x_0 \leq \sqrt{2} - 1$$

Now let $r := \epsilon_0 = \sqrt{N} - 1$ and $I := [\sqrt{N}^{-1} - r, \sqrt{N}^{-1}]$. If we consider the lower bound of I we see that $\sqrt{N}^{-1} - (\sqrt{N}^{-1} - 1) = 1$, and in particular $0 \notin I$.

Next we know that $f(x) = N - x^{-2}$, and therefore we get $f'(x) = 2x^{-3}$, $f''(x) = -6x^{-4}$. It is obvious that $\nexists x \in \mathbb{R} : f'(x) = 0$, which means that $f'(x) \neq 0 \forall x \in I$ and so NR_1 is satisfied. Also as f'' is only discontinuous at $x = 0$ and $0 \notin I$, then $f''(x)$ is continuous $\forall x \in I$, meaning this satisfies NR_2.

Now $M = \sup\left\{\frac{1}{2}\left|\frac{2x^3}{6y^4}\right| : x, y \in I\right\}$, we can simplify the function we are trying to minimise to get $\frac{1}{6}\frac{x^3}{y^4}$. It is obvious that in order to maximise this function we should find the largest possible x and smallest possible y, as both are positive. Hence by taking $x = \sqrt{N}^{-1} + r$ and $y = 1$, then $M = \frac{1}{6}(2\sqrt{N}^{-1} - 1)^3 \leq \frac{1}{6}(2\sqrt{2} - 1)^3$.

Now we consider $M\epsilon_0$:

$$\begin{aligned} M\epsilon_0 &= \frac{1}{6}(2\sqrt{N}^{-1} - 1)^3(\sqrt{N} - 1) \\ &\leq \frac{1}{6}(2\sqrt{2} - 1)^3(\sqrt{2} - 1) \\ &\approx 0.42199376\ldots \\ &< 1 \end{aligned}$$

Therefore as $M\epsilon_0 < 1$ we have satisfied NR_3, and as such we have quadratic convergence of our method to $\sqrt{N}^{-1}$. $\square$

3.5 Comparison of Methods

We have observed several methods that can be used to calculate Square Roots, and so now we will see how the methods compare to each other in practice. The exact root method that we first discussed is the hardest to compare to the other methods as it works in a very different manner to the others. For now we will merely observe that it is an inefficient method that can be will be shown to take longer than the others.

Second we need to compare the different methods discussed for the Newton Square Root method. As they work in the same general method, we really only need to test the computation time of the different methods. To do this we will be testing 1000 values in the range $(0, 1000)$ and will calculate each of these values 100000 times, accurate to within a tolerance of $10^-1)$, for each method to give the most accurate results. The table below gives the calculated results:

	`mpfr_newton_sqrt_v1`	`mpfr_newton_sqrt_v2`	`mpfr_newton_sqrt_v3`
Total time:	`10.507s`	`12.707s`	`8.188s`
Average time:	`0.010s`	`0.012s`	`0.008s`
Minimum time:	`0.003s`	`0.004s`	`0.005s`
Maximum time:	`0.016s`	`0.021s`	`0.016s`

Here we see that our third method, as expected, is the fastest of the proposed methods and so we will use this method going forwards. One unexpected result is that the second method is actually slower than the first, which is likely due to the extra conversions, comparisons and method calls; this slows down the execution more than it is sped up by reduction in number of iterations required.

Now for the comparison of methods we will be comparing modified versions of Algorithms 3.2.4, 3.3.4 and 3.4.1, which will execute for a given number of steps, rather than testing for the approximate error. To do this we need to consider how many iterations each method needs to reach a particular number of decimal places of accuracy.

We have seen the required number of iterations for a tolerance $\tau = 10^{-k} : k \in \mathbb{N}$, for both the bisection and basic newton square root methods, and similar to the basic newton method, we can show that for the inverse newton method we are looking for $n \in \mathbb{N}$ that satisfies the following inequality:

$$n > \log_2 \left(\log_{\frac{1}{6}(\sqrt{2}-1)(2\sqrt{2}-1)^3} \left(\frac{\tau}{\sqrt{2}-1} \right) \right) - 1$$

This gives the following table:

$k : \tau = 10^k$	Bisection Method	Newton Method	Inverse Newton
5	16	4	4
10	33	5	5
100	332	8	9
1,000	3321	12	12
1,000,000	3219280	22	22

To show the above in action we have the table below which shows the convergence of all 3 methods to $\sqrt{0.75} \approx 0.86602540378$, for different numbers of iterations n with the bold digits being those correct:

n	$\text{bisectSquareRoot}(0.75, n)$	$\text{NewtonSquareRoot}(0.75, n)$	$\text{NewtonInvSquareRoot}(0.75, n)$
0	**0.**500000000000000000	**1.**000000000000000000	**0.**750000000000000000
1	**0.**750000000000000000	**0.8**75000000000000000	**0.8**43750000000000000
2	**0.8**75000000000000000	**0.8660**71428571428603	**0.86**5173339843750000
3	**0.8**12500000000000000	**0.86602540**5007363691	**0.86602**4146705512976
4	**0.8**43750000000000000	**0.866025403784438**596	**0.86602540378**1701674
5	**0.8**59375000000000000	**0.866025403784438**596	**0.866025403784438**596
6	**0.86**7187500000000000	**0.866025403784438**596	**0.866025403784438**596

If we compare the methods so that they guarantee an accuracy of 10 decimal places, then we will be able to see their relative efficiency. In particular we will again be testing the three methods using 1000 values in the range $(0, 1000)$, and calculating the square root of each of these values 10000 times for each method; further we will be including the digit by digit method and the built-in C `sqrt` function. The results calculated are present in the following table:

	`root_digits_precise`	`bisect_sqrt`	`newton_sqrt`	`newton_inv_sqrt`	`bui`
Total time:	227.620s	2.520s	1.028s	0.646s	
Average time:	0.227s	0.002s	0.001s	0.000s	
Minimum time:	0.160s	0.002s	0.000s	0.000s	
Maximum time:	0.429s	0.004s	0.004s	0.001s	

Here we see the expected result that the digit by digit method is the least efficient method, taking two orders of magnitude more time than the second least efficient. We also see that while the two different newton methods are similar in time, and that even though they each performed the same number of iterations, the inverse square root method is the faster; this is due to the method having no division operations to perform. The quickest is of course the built-in `sqrt` function from C, this is due to an implementation that uses several low-level features of the C language to achieve the displayed level of performance.

In conclusion we can say that the best method that we have considered is Algorithm **??** which has rapid convergence to the sought square root, while also having fast execution. However if we are in a situation where we require large numbers of digits of accuracy, and yet do not have a suitable floating point types large enough to store these values, then the digit by digit method can be used to get an arbitrary number of digits accuracy.

4 Trigonometric Functions

The trigonometric functions have been studied since antiquity, originally for their relation to triangles, which were incredibly important to early mathematical understanding. Nowadays the trigonometric functions have found applications in a vast array of problems from musical theory to satellite navigation.

Here we will discuss various methods for approximating the trigonometric functions $\sin$, $\cos$ and $\tan$. Further we will explore the inverse trigonometric functions $\sin^{-1}$, $\cos^{-1}$ and $\tan^{-1}$ which also have many practical uses in modern life.

Trigonometric functions can be calculated using either degrees of an angle (e.g. $\sin(60^\circ) = \frac{\sqrt{3}}{2}$) or in radians(e.g. $\sin(\frac{pi}{3}) = \frac{\sqrt{3}}{2}$). For this document we will only discuss the use of radians and consider that $\theta^{\text{rad}} = \theta^\circ \cdot \frac{\pi}{180}$ can be used to convert between the two units if needed.

4.1 Trigonometric Identities

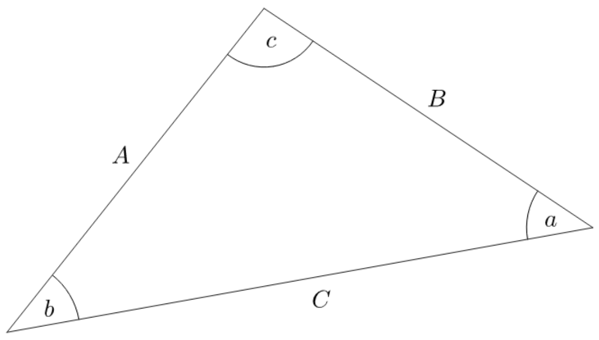
Most readers will be well aware of the standard trigonometric identities: useful equalities that help in the analysis of trigonometric functions; this section will recap such identities that will prove useful in this document. As with section **??** this is not meant to be an exhaustive overview, merely a reminder and as such identities not listed here may be used in the document.

The first identities to consider are the basic ones taught in secondary school such as $\sin^2 x + \cos^2 x = 1$. In particular we are interested in the shifts, reflections and periods of $\sin$ and $\cos$. Some of the relevant functions are included below:

$$\begin{array}{rclrcl}
\sin(-x) & = & -\sin(x) & \cos(-x) & = & \cos(x) \\
\sin(x+\frac{\pi}{2}) & = & \cos(x) & \cos(x+\frac{\pi}{2}) & = & -\sin(x) \\
\sin(x+\pi) & = & -\sin(x) & \cos(x+\pi) & = & -\cos(x) \\
\sin(x+2\pi) & = & \sin(x) & \cos(x+2\pi) & = & \cos(x)
\end{array}$$

Another useful formula to remember is the sine rule, which is detailed in figure **??** as well as the combined angle formulas $\sin(x) = \sin(x)\cos(y) \pm \sin(y)\cos(x)$, $\cos(x \pm y) = \cos(x)\cos(y) \mp \sin(x)\sin(y)$ and $\tan(x) = \frac{\tan(x) \pm \tan(y)}{1 \mp \tan(x)\tan(y)}$.

Figure 4.1.1: The Sine Rule



$$\frac{\sin(a)}{A} = \frac{\sin(b)}{B} = \frac{\sin(c)}{C}$$

A final note in this section is the derivatives of the trigonometric functions, in particular $\frac{d}{dx}\sin(x) = \cos(x)$, $\frac{d}{dx}\cos(x) = -\sin(x)$ and $\frac{d}{dx}\tan(x) = \sec^2(x)$ will be useful later on in the development of methods.

4.2 Calculating π

Several of the methods in this section require that we already know the value of π, for example when we are applying several trig identities. Here we will briefly discuss several methods for calculating the value of π, so that we may use this value in later subsections.

The first method to consider is the method used by ancient mathematicians, such as the Greeks and Chinese. We know that if the radius of the circle is $\frac{1}{2}$, then the circumference of the circle is π, and the value is between the perimeters of the inner and outer polygon perimeters. The internal perimeter is $p_n = n\sin(\frac{\pi}{n})$ and the external perimeter is $P_n = n\tan(\frac{\pi}{n})$.
As we know the values of $\tan(\frac{\pi}{6})$ and $\sin(\frac{\pi}{6}$, then we can calculate P_6 and p_6. It has be shown that $P_{2n} = \frac{2p_nP_n}{p_n+P_n}$ and $p_{2n} = \sqrt{p_nP_{2n}}$, which allows us to create an iterative method to approximate π, by taking the mid-point of the successive polygon perimeters.

Other common historical methods for approximating π are to use infinite series. One such method uses the series expansion of $\tan^{-1}$, which is discussed in detail below, where $\tan^{-1}(1) = \frac{\pi}{4}$. This gives the following approximation using N terms:

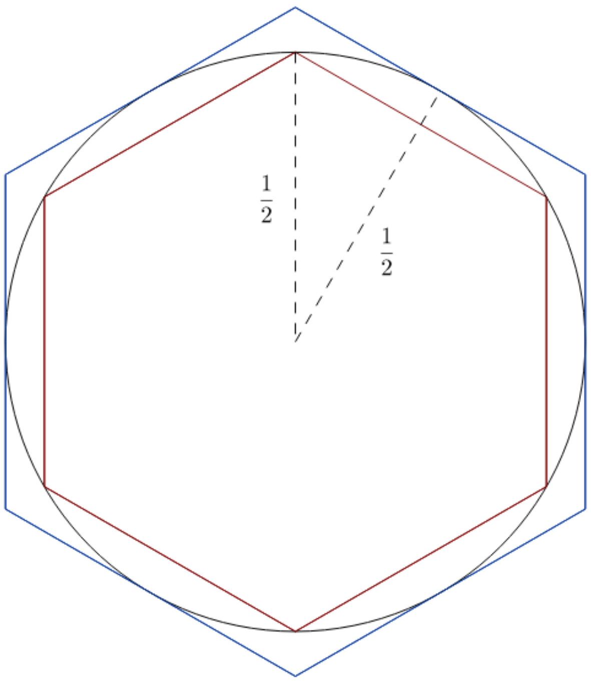
$$\pi = 4\sum_{n=0}^{N}\frac{(-1)^n}{2n+1} = \sum_{n=0}^{N}\frac{8}{(4n+1)(4n+3)} \tag{4.2.1}$$

This sequence converges very slowly, with sub linear convergence, to the correct value. More modern methods have typically revolved around finding more rapidly converging infinite series, examples include Ramanujan's series:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103+26390n)}{(k!)^n 396^{4n}} \tag{4.2.2}$$

or the Chudnovsky algorithm:

Figure 4.2.1: Ancient method of calculating π



$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(13591409 + 545140134n)}{(3n)!(n!)^3 640320^{3n+\frac{3}{2}}} \tag{4.2.3}$$

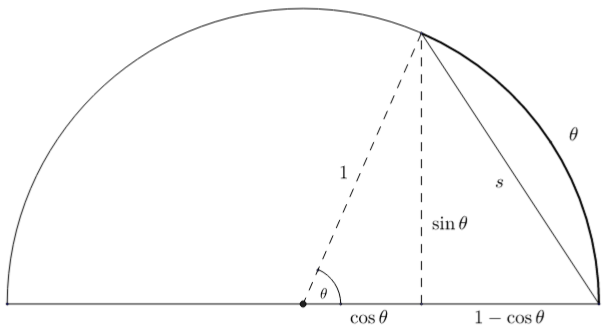
This final series is extremely rapidly convergent to the value of $\frac{1}{\pi}$, for example just the first term gives π accurate to 13 decimal places while we can get π accurate to 1000 decimal places with summing just 71 terms. Compared to Equation **??** which takes the summation of 500 terms to achieve the same 1000 digits of accuracy.

To get large degrees of accuracy for π is extremely computer intensive and using the `mpfr` requires the number of bits of precision and number of terms to be set. This makes calculating π to a large number of decimal places, for example 1000000, computationally infeasible on a regular home computer. Therefore for our purposes we will use the pre calculated value of π to 1000000 decimal places as listed on `http://www.exploratorium.edu/pi/pi_archive/Pi10-6.html`

4.3 Geometric Method

The first method I will be discussing is a method based on geometric properties that are derived on a circle, and we will start by considering values of $\cos$ in the range $[0, \frac{\pi}{2}]$. To do this we will consider the following figure of the unit circle:
Here theta will be given in radians, and we can note that the labelled arc has length θ due the formula for the circumference of a circle. By using the following derivation we can find a formula for θ in terms of s:

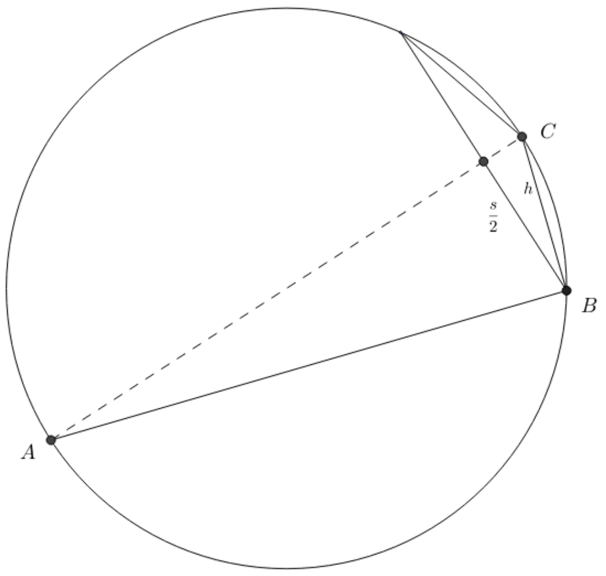
Figure 4.3.1: Diagram showing angles to be dealt with



$$\begin{aligned}
s^2 &= \sin^2\theta + (1-\cos\theta)^2 \\
&= (\sin^2\theta + \cos^2\theta) + 1 - 2\cos\theta \\
&= 2 - 2\cos\theta && \text{By using } \sin^2\theta + \cos^2\theta = 1 \\
\cos\theta &= 1 - \frac{s^2}{2}
\end{aligned}$$

We will now consider a second diagram which will allow us to calculate an approximate value of s.

Figure 4.3.2: Diagram detailing how to calculate s

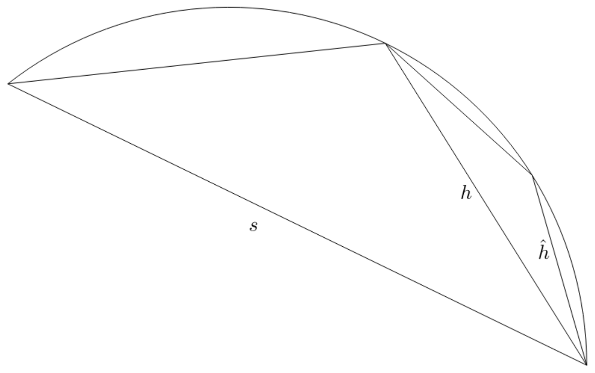


We will first note that by an elementary geometry result we can know that the angle ABC is a right-angle; also we can consider that h is an approximation of $\frac{\theta}{2}$, which will become relevant later. Now because AC is a diameter of our circle then it's length is 2 and thus, by utilising Pythagoras' Theorem, we get that the length of AB is $\sqrt{AC^2 - BC^2} = \sqrt{4-h^2}$.

From here we consider the area of triangle ABC, which can be calculated as $\frac{1}{2} \cdot h \cdot \sqrt{4-h^2}$ and as $\frac{1}{2} \cdot 2 \cdot \frac{s}{2}$; by equating these two, squaring both sides and re-arranging we get that $s^2 = h^2(4-h^2)$. Now we have the basis for a method that will allow us to calculate $\cos\theta$.

To complete our method we will consider introducing a new line that is to h what h is to s as shown in the diagram below:

Figure 4.3.3: Detailing the recursive steps



It is easy to see that if we repeat the steps above we get that $h^2 = \hat{h}^2(4 - \hat{h}^2)$, and it also follows that $\hat{h} \approx \frac{\theta}{4}$. Using this we can take an initial guess of $h_0 := \frac{\theta}{2^k}$, for some $k \in \mathbb{N}$, and then calculate $h_{n+1}^2 = h_n^2(4 - h_n^2)$ where $n \in [0, k] \cap \mathbb{Z}$; finally we calculate $\cos\theta = 1 - \frac{h_k^2}{2}$, giving the following algorithm:

Algorithm 4.3.1: Geometric calculation of $\cos$

```
geometric_cos($\theta \in [0, \frac{\pi}{2}], k \in \mathbb{N}$)
    $h_0 := \frac{\theta}{2^k}$
    $n := 0$
    while $n < K$:
        $h_{n+1}^2 := h_n^2 \cdot (4 - h_n^2)$
        $n \mapsto n + 1$
    return $1 - \frac{h_k^2}{2}$
```

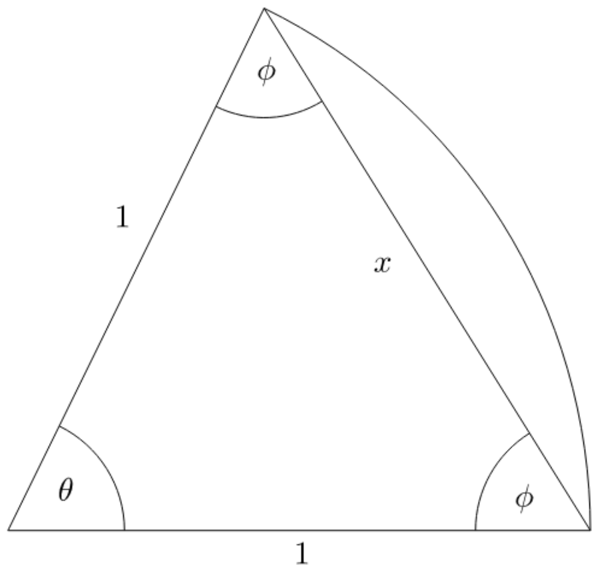
Now we can use the above pseudo-code to calculate any trigonometric function value by using various trigonometric identities. First we suppose $\theta \in \mathbb{R}$, then we can repeatedly apply the identity $\cos\theta = \cos(\theta \pm 2\pi)$ to either add or subtract 2π until we have a value $\theta' \in [0, 2pi)$. Once we have this value we can utilise the following assignment to calculate $\cos\theta$:

$$\cos\theta = \begin{cases} \cos\theta' & : \quad \theta' \in [0, \frac{\pi}{2}] \\ -\cos(\pi - \theta') & : \quad \theta' \in [\frac{\pi}{2}, \pi] \\ -\cos(\theta' - \pi) & : \quad \theta' \in [\pi, \frac{3\pi}{2}] \\ \cos(2\pi - \theta') & : \quad \theta' \in [\frac{3\pi}{2}, 2\pi) \end{cases}$$

Using Algorithm 4.3.1 we can also easily calculate both $\sin\theta$ and $\tan\theta$, by further use of trigonometric identities. In particular we note that $\sin\theta = \cos(\theta - \frac{\pi}{2}$ and $\tan\theta = \frac{\sin\theta}{\cos\theta}$. Hence we can now calculate the trigonometric function value of any angle.

We now wish to analyse the error of our approximation for $\cos$, as the other methods have errors that are derivative of the error for approximating $\cos$. Now Figure 4.3.4 shows an arc of a circle which creates chord x, with this we will be able to calculate the exact length of the chord and thus work on the error of our approximations.

Figure 4.3.4: Diagram to find actual arc approximation



To start we will note that $\phi = \frac{\pi-\theta}{2} = \frac{\pi}{2} - \frac{\theta}{2}$, and then by using the Sine Law we get

$$\frac{x}{\sin\theta} = \frac{1}{\sin\phi} \implies x = \frac{\sin\theta}{\sin\phi}$$

Now we can recall the double angle formula for sin, which gives $\sin\theta = 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}$, and also $\sin\phi = \cos\frac{\theta}{2}$. This allows us to see that

$$x = \frac{2\sin\frac{\theta}{2}\cos\frac{\theta}{2}}{\cos\frac{\theta}{2}} = 2\sin\frac{\theta}{2}$$

Therefore we see that h_n is approximating the chord length associated with angle $\theta 2^{n-k}$, and thus $\epsilon_n = |h_n - 2\sin(\theta 2^{n-k-1})|$. Now as $h_0 = \theta 2^{-k} \approx 2\sin(\theta 2^{-k-1})$ then if follows that $\exists\phi$ such that $h_0 = 2\sin(\phi 2 - k - 1)$, from this we can see that $\phi = 2^{k+1}\sin^{-1}(\theta 2^{-k-1})$. We will uses these facts to prove a couple of propositions.

Proposition 4.3.1. $h_n = 2\sin(\phi 2^{n-k-1}) \forall n \in [0,k] \cap \mathbb{Z}$ *where* $\phi := 2^{k+1}\sin^{-1}(\theta 2^{-k-1})$.

Proof. Proceed by induction on $n \in [0,k] \cap \mathbb{Z}$.

$\mathbf{H}(n)$: $h_n = 2\sin(\phi 2^{n-k-1})$

$\mathbf{H}(0)$:

$$\begin{aligned} 2\sin(\phi 2^{-k-1}) &= 2\sin(\sin^{-1}(\phi 2^{-k-1})) \\ &= \theta 2^{-k} \\ &= h_0 && \text{by definition of } h_0 \end{aligned}$$

$\mathbf{H}(n) \implies \mathbf{H}(n+1)$:

$$\begin{aligned} h_{n+1} &= h_n\sqrt{4 - h_n^2} \\ &= 2\sin(\phi 2^{n-k-1})\sqrt{4 - 4\sin^2(\phi 2^{n-k-1})} && \text{by H}(n) \\ &= 4\sin(\phi 2^{n-k-1})\cos(\phi 2^{n-k-1}) \\ &= 2\sin(\phi 2^{n-k}) && \text{by the use of double angle formulas} \end{aligned}$$

□

Proposition 4.3.2. $h_n > 2\sin(\theta 2^{n-k-1}) \forall n \in [0,k] \cap \mathbb{Z}$

Proof. We start by considering the expansion of the exact value of h_n.

$$\begin{aligned}
h_n &= 2\sin(\phi 2^{n-k-1}) \\
&= 2\sin(2^{n-k-1}(2^{k+1}\sin^{-1}(\theta 2^{-k-1}))) \\
&= 2\sin(2^n \sin^{-1}(\theta 2^{-k-1})) \\
&= 2\sin(\theta 2^{n-k-1} + \tfrac{1}{6}\theta^3 2^{n-3k-3} + \mathcal{O}(2^{-5k})) \quad \text{Detailed in section } \textbf{??}
\end{aligned}$$

Now as we know that $n \leq k$, then it follows that $\theta 2^{n-k-1} \leq \frac{1}{2}\theta$.

Also as $\theta \leq \frac{\pi}{2}$ we know that $\theta 2^{n-k-1} \leq \frac{\pi}{4}$.

We can also show that $\frac{1}{6}\theta^3 2^{n-3k-3} + \mathcal{O}(2^{-5k}) \leq \frac{\pi}{4}$, though the proof is omitted here for brevity; therefore we see that $\phi 2^{n-k-1} \leq \frac{\pi}{2}$, and obviously that $\phi 2^{n-k-1} > \theta 2^{n-k-1}$.

Hence, as $\sin$ is an increasing function in the range $[0, \frac{\pi}{2}]$, we conclude that

$$h_n = 2\sin(\phi 2^{n-k-1}) > 2\sin(\theta 2^{n-k-1})$$

. □

With these two propositions we can now consider the error of our approximation of $\cos$. First we will prove the following proposition regarding the error of the approximation of s:

Proposition 4.3.3. *If* $\epsilon_n := |h_n - 2\sin(\theta 2^{n-k-1})| \forall n \in [0,k] \cap \mathbb{Z}$*, then* $\epsilon_k < 2^k \epsilon_0$).

Proof. $\epsilon_n = h_n - 2\sin(\theta 2^{n-k-1})$ as $h_n > 2\sin(\theta 2^{n-k-1})$ by Proposition 4.3.2.

Now we see that:

$$\begin{aligned}
\epsilon_{n+1} &= h_{n+1} - 2\sin(\theta 2^{n-k}) \\
&= h_n\sqrt{4 - h_n^2} - 4\sin(\theta 2^{n-k-1})\cos(\theta 2^{n-k-1})
\end{aligned}$$

If we consider the equation $\alpha\beta - \gamma\delta = (\alpha - \gamma) + \alpha(\beta - 1) - \gamma(\delta - 1)$ and apply it to our current formula we get:

$$\begin{aligned}
\epsilon_{n+1} &= (h_n - 2\sin(\theta 2^{n-k-1})) + h_n(\sqrt{4 - h_n^2} - 1) - 2\sin(\theta 2^{n-k-1})(2\cos(\theta 2^{n-k-1}) - 1) \\
&= \epsilon_n + h_n(\sqrt{4 - h_n^2} - 1) - 2\sin(\theta 2^{n-k-1})(2\cos(\theta 2^{n-k-1}) - 1) \\
&= 2\epsilon_n + h_n(\sqrt{4 - h_n^2} - 2) - 2\sin(\theta 2^{n-k-1})(2\cos(\theta 2^{n-k-1}) - 2) \\
&= 2\epsilon_n + h_n(\sqrt{4 - h_n^2} - 2) + 2\sin(\theta 2^{n-k-1})(2 - 2\cos(\theta 2^{n-k-1})) \\
&< 2\epsilon_n + h_n(\sqrt{4 - h_n^2} - 2\cos(\theta 2^{n-k-1})) \\
&< 2\epsilon_n + h_n(\sqrt{4 - 4\sin^2(\theta 2^{n-k-1})} - 2\cos(\theta 2^{n-k-1})) \\
&= 2\epsilon_n + h_n(2\cos(\theta 2^{n-k-1}) - 2\cos(\theta 2^{n-k-1})) \\
&= 2\epsilon_n
\end{aligned}$$

The inequalities in the above derivation arise from the fact that $h_n > 2\sin(\theta 2^{n-k-1})$ by Proposition 4.3.2.

Hence as we now know that $\epsilon_{n+1} < 2\epsilon_n$, we then see that $\epsilon_n < 2^n\epsilon_0$. Therefore we prove our statement that

$$\epsilon_k < 2^k\epsilon_0$$

□

Obviously $\epsilon_k = |h_k - s|$, and we can now use this to find the error of our final answer. First we will start by letting $\mathcal{C} := 1 - \frac{1}{2}h_k^2$ and note that analytically $cos\theta = 1 - \frac{1}{2}s^2$. Therefore we will now consider $\epsilon_{\mathcal{C}} = |\mathcal{C} - \cos(\theta)|$:

$$\begin{aligned}
\epsilon_{\mathcal{C}} &= |1 - \frac{h_k^2}{2} - 1 + \frac{s^2}{2}| \\
&= \frac{1}{2}|h_k^2 - s^2| \\
&= \frac{1}{2}|h_k h_k - 2\sin(\frac{\theta}{2})2\sin(\frac{\theta}{2})| \\
&= \frac{1}{2}(h_k h_k - 2\sin(\frac{\theta}{2})2\sin(\frac{\theta}{2})) && \text{as } 2\sin(\frac{\theta}{2}) < h_k \\
&= \frac{1}{2}(2\epsilon_k + h_k(h_k - 2) - 2\sin(\frac{\theta}{2})(2\sin(\frac{\theta}{2}) - 2) \\
&< \frac{1}{2}(2\epsilon_k + h_k(h_k - 2\sin(\frac{\theta}{2}))) \\
&= \frac{1}{2}(2 + h_k)\epsilon_k \\
&= \frac{1}{2}(2 + 2\sin(\frac{\phi}{2}))\epsilon_k \\
&= (1 + \sin(\frac{\phi}{2}))\epsilon_k \\
&\leq 2\epsilon_k
\end{aligned}$$

As $\epsilon_{\mathcal{C}} \leq 2\epsilon_k$, then by Proposition 4.3.3 we see that $\epsilon_{\mathcal{C}} < 2^{k+1}\epsilon_0$. Now to consider ϵ_0 we first observe that $\epsilon_0 = \theta 2^{-k} - 2\sin\theta 2^{-k-1}$, and therefore we can conclude that:

$$\epsilon_{\mathcal{C}} < 2\theta - 2^{k+2}\sin(\theta 2^{-k-1})$$

This looks like an error that may in fact grow exponentially large as $k \to \infty$, due to the multiplication by 2^{k+2}. However if we instead consider the series expansion of $\sin(x)$, shown in Section 4.4 to be $\sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \cdots$, and substitute that into our equation we see that:

$$\begin{aligned}
\epsilon_{\mathcal{C}} &< 2\theta - 2^{k+2}(\theta 2^{-k-1} - \frac{1}{3!}\theta^3 2^{-3k-3} + \frac{1}{5!}\theta^5 2^{-5k-5} - \cdots) \\
&= 2\theta - 2\theta + \frac{1}{3}\theta^3 2^{-2k-1} - \frac{1}{5!}\theta^5 2^{-4k-3} + \cdots \\
&= \frac{1}{3}\theta^3 2^{-2k-1} - \frac{1}{5!}\theta^5 2^{-4k-1} + \cdots
\end{aligned}$$

Now obviously the last line tends towards zero as k tends to infinity, due to it being a formula of order $\mathcal{O}(2^{-2k-1})$. Therefore we know that $\forall\tau \in \mathbb{R}^+ \exists\mathcal{K} \in \mathbb{N} : \epsilon_{\mathcal{C},k} < \tau \forall k \in [\mathcal{K}, \infty) \cap \mathbb{Z}$. In particular, if we then wish to calculate $\cos\theta$ accurate to N decimal places then we are looking to find $k \in \mathbb{N}$ such that:

$$2\theta - 2^{k+2}\sin(\theta 2^{-k-1}) < 10^{-N} \implies 2^{k+2}\sin(\theta 2^{-k-1}) > 2\theta - 10^{-N}$$

For an example of the above in action we will be taking $\theta = 0.5$. The table below shows the minimum $k \in \mathbb{N}$ to guarantee N digits of accuracy in the result:

N	k
5	6
10	14
50	80
100	163
1000	1658

As can be seen the value of k required to achieve N digits of accuracy increases roughly linearly when $\theta = 0.5$. Testing for other values of θ reveals them to have similar required values for k, at least within the same order of each other.

Another consideration for Algorithm 4.3.1 is that we could "run it in reverse" to attain an algorithm for the inverse cosine function. To start take line 7 which is $\mathcal{C} = 1 - \frac{1}{2}h_k^2$, which can be re-arranged to give $h_k^2 = 2 - 2\mathcal{C}$, where we know $\mathcal{C}$ as our initial value.

Line 5 is a little more difficult, but by re-arranging we see that $h_n^4 - 4h_n^2 + h_{n+1}^2 = 0$, which can be solved via the quadratic formula to give $h_n^2 = 2 \pm \sqrt{4 - h_{n+1}^2}$. Now we can make the observation that if $x \in \mathbb{R}_0^+$, then $\cos^{-1}(-x) = \pi - \cos^{-1}(x)$ and so we can restrict our algorithm to only consider $x \in [0, 1]$. With this we know that $\theta \in [0, \frac{\pi}{2}]$, and thus $h_k \leq \sqrt{2}$. Therefore as $h_{n+1} > h_n \forall n \in [0, k-1] \cap \mathbb{Z}$ we see that $h_n^2 \leq 2 \forall n \in [0, k] \cap \mathbb{Z}$. This allows us to ascertain that to reverse Line 5 we perform $h_n^2 = 2 - \sqrt{4 - h_{n+1}^2}$.

Finally line 2 is reversed by returning the value $2^k h_0$; therefore we get the following algorithm for $\cos^{-1}(x)$ where $x \in [0, 1]$:

Algorithm 4.3.2: Geometric calculation of $\cos^{-1}$

```
1  geometric_aCos($x \in [0,1], k \in \mathbb{N}$)
2      $h_k := 2 - 2x$
3      $n := k - 1$
4      while $n \geq 0$:
5          $h_n^2 := 2 - \sqrt{4 - h_{n+1}^2}$
6          $n \mapsto n - 1$
7      return $2^k h_0$
```

Similar to the regular trigonometric functions we can use trigonometric identities to calculate the inverse trigonometric functions from $\cos^{-1}$. To start we recall that $\cos^{-1}(-x) = -\cos(x)$ where $x \in [0, 1]$, then we can use the identities that $\sin^{-1}(x) = \frac{\pi}{2} - \cos^{-1}(x)$ and $\tan^{-1}(x) = \sin^{-1}(\frac{x}{\sqrt{x^2+1}})$.

If we suppose that all operations in the method are accurately computed then Algorithm 4.3.2 is a computation with high accuracy. This is because there is no initial guess, such as in Algorithm 4.3.1, and so the only introduction of error is assuming that $2^k h_0 \approx \theta$. However as we discuss in detail in Section **??**, calculating square roots is not a simple task and thus will introduce error to the method in general; therefore the accuracy of the method is roughly as accurate as our method of calculating square roots.

4.4 Taylor Series

If we consider our definition of a McClaurin Series from Section **??**, we can use this to approximate our Trigonometric Functions. Consider first $\cos\theta$, for which we know that $\frac{d}{d\theta}\cos\theta = -\sin\theta$; it then follows that $\frac{d^2}{d\theta^2}\cos\theta = -\cos\theta$, $\frac{d^3}{d\theta^3}\cos\theta = \sin\theta$ and $\frac{d^4}{d\theta^4}\cos\theta = \cos\theta$.

If we let $f(x) = \cos x$ and use the known values $\cos(0) = 1$ and $\sin(0) = 0$, then we see that:

$$f^{(n)}(0) = \begin{cases} 1 & : \ 4 \mid n \\ 0 & : \ 4 \mid n-1 \\ -1 & : \ 4 \mid n-2 \\ 0 & : \ 4 \mid n-3 \end{cases}$$

By simplifying this by omitting the 0 coefficient terms we get the following series:

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \cdots \tag{4.4.1}$$

By using similar working we can get that the series associated with $\sin x$):

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{4.4.2}$$

Before we go any further we need to consider when Equations 4.4.1 and 4.4.2 converge to their respective functions. To do this we will use the ratio test for series as defined in **??**, using Equation 4.4.1 we see that

$$\begin{aligned} L_{\mathcal{C}} &= \lim_{n\to\infty} \left| \frac{a_{n+1}}{a_n} \right| \\ &= \lim_{n\to\infty} \left| \frac{\frac{(-1)^{n+1}}{(2n+2)!} x^{2n+2}}{\frac{(-1)^n}{(2n)!} x^{2n}} \right| \\ &= \frac{(2n)!}{(2n+2)!} |x|^2 \\ &= \frac{1}{(2n+2)(2n+1)} |x|^2 \end{aligned}$$

Now it is easy to see that, $L_{\mathcal{C}} = 0$ for all values of x as the fractional component decreases as n increases and $|x|^2$ is a constant. Therefore we can conclude that Equation 4.4.1 converges to $\cos(x)$ for all values of x. We can use a very similar deduction to show that Equation 4.4.2 converges to $\sin(x)$ for all values of x.

The above means that $\cos$ and $\sin$ can be approximated using Taylor Polynomials, in particular for a given $N \in \mathbb{N}$:

$$\cos x \approx \textstyle\sum_{n=0}^{N} \frac{(-1)^n}{(2n)!} x^{2n} \quad \text{and} \quad \sin x \approx \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

This allows us to create the following two methods for computing $\cos x$ and $\sin x$:

Algorithm 4.4.1: Taylor computation of $\cos$ and $\sin$

```
taylor_cos($x \in \mathbb{R}, N \in \mathbb{N}$)
    $\mathcal{C} := 0$
    $n := 0$
    while $n < N$:
        $\mathcal{C} \mapsto \mathcal{C} + (-1)^n \cdot \frac{1}{(2n)!} x^{2n}$
        $n \mapsto n + 1$
    return $\mathcal{C}$

taylor_sin($x \in \mathbb{R}, N \in \mathbb{N}$)
    $\mathcal{S} := 0$
    $n := 0$
    while $n < N$:
        $\mathcal{S} \mapsto \mathcal{S} + (-1)^n \cdot \frac{1}{(2n+1)!} x^{2n+1}$
        $n \mapsto n + 1$
    return $\mathcal{S}$
```

As these two methods are obviously very similar and the fact that $\sin(x) = \cos(x - \frac{\pi}{2})$, we will continue by examining only the Taylor method for approximating $\cos$. We will assume that any calculations for $\sin$ are transformed into a problem of finding a $\cos$ value.

It should be noted that this $\cos$ algorithm is particularly inefficient to calculate on a computer implementation; this is primarily due to the way in which the update of $\mathcal{C}$ is calculated each loop.

In each loop we are calculating x^{2n}, which has a naive complexity of $\mathcal{O}(2n)$. However what we are actually calculating $x^{2(n-1)} \cdot x^2$ and thus if we store the values of $x^{2(n-1)}$ and x^2, the complexity of this step drops to $\mathcal{O}(1)$. Similarly we are also calculating $\frac{1}{(2n)!}$ in each loop which, by the same logic, is $\frac{1}{2(n-1)!} \cdot \frac{1}{(2n)(2n-1)}$, and we can use the same storage and update method as for x^{2n}.

As another step towards optimizing the algorithm we can start with an initial value of $\mathcal{C} = 1$, and then perform two updates of $\mathcal{C}$ each loop until we reach or surpass N. This saves calculating $(-1)^n$ each loop, by explicitly performing two different calculations. Implementing all of the above gives us the following two updated methods:

Algorithm 4.4.2: Taylor computation of $\cos$ optimised

```
taylor_cos($x \in \mathbb{R}, N \in \mathbb{N}$)
    $\mathcal{C} := 1$
    $x_2 := x^2$
    $a := 1$
    $b := 1$
    $n := 1$
    while $n < N$:
        $a \mapsto a \cdot \frac{1}{(2n-1)(2n)}$
        $b \mapsto b \cdot x_2$
        $\mathcal{C} \mapsto \mathcal{C} - a \cdot b$
        $a \mapsto a \cdot \frac{1}{(2n+1)(2n+2)}$
```

```
            $b \mapsto b \cdot x_2$
            $\mathcal{C} \mapsto \mathcal{C} + a \cdot b$
            $n \mapsto n + 2$
        return $\mathcal{C}$
```

As the next term of the polynomial is known definitively then we can see that it is very easy to calculate the error of our approximation. We see that

$$\begin{aligned}
\epsilon_N &= |\cos(x) - \text{taylor\_cos(x, N)}| \\
&= \mathcal{O}(|x|^{N'+1}) && \text{where } N' \text{ is the smallest} \\
& && \text{odd integer such that } N' \geq N \\
&\leq \frac{1}{(2(N'+1))!}|x|^{N'+1} \\
&\leq \frac{1}{(2(N+1))!}|x|^{N+1}
\end{aligned}$$

If we place bounds on the value of $\cos$ calculated as in Section 4.3, then we know that $|x| \leq \frac{\pi}{2}$, and thus we get the following bound for the error of our approximation:

$$\epsilon_N \leq \frac{\pi^{N'+1}}{2^{N'+1}(2(N'+1))!}$$

Thus if we find $N \in \mathbb{N}$ such that $\frac{\pi^N+1}{2^{N+1}(2(N+1)!)} < \tau \in \mathbb{R}+$ then we know that $\epsilon_N < \tau$. If we consider $\tau = 10^k$, then we can find $N \in \mathbb{N}$ such that our approximation is accurate to k decimal places. Below is a table which details some values of k and the corresponding minimum N to guarantee k decimal places of accuracy:

k	N
5	4
10	7
50	21
100	36
1000	233

Now for $\tan x$ we can either calculate both $\sin x$ and $\cos x$ using taylor_cos(x,N) and divide the resulting value, or we can calculate $\tan x$ directly using a Taylor expansion.

In calculating the McClaurin series for $\tan x$ we start by letting $\tan x = \sum_{n=0}^{\infty} a_n x^n$, and then noting that as $\tan x$ is an odd series then it's McClaurin series only contains non-zero coefficients for odd powers of x; therefore we get that $\tan x = \sum_{n=0}^{\infty} a_{2n+1}x^{2n+1} = a_1 x + a_3 x^3 + a_5 x^5 + \cdots$.

Next we consider that $\frac{d}{dx}\tan x = 1 + \tan^2 x$, and knowing the McClaurin series form of $\tan x$ we get the following:

$$\begin{aligned}
\sum_{n=0}^{\infty}(2n+1)a_{2n+1}x^{2n} &= 1 + (\sum_{n=0}^{\infty} a_{2n+1}x^{2n+1})^2 \\
&= 1 + a_1^2 x^2 + (2a_1a_3)x^4 + (2a_1a_5 + a_3^2)x^6 + \cdots
\end{aligned}$$

Considering the coefficients of powers on the right hand side of the above equation we see that $2a_1a_3 = a_1a_3 + a_{3a1} = a_1a_{4-1} + a_3a_{4-3}$ and $2a_1a_5 + a_3^2 = a_1a_5 + a_3a_3 + a_5a_1 = a_1a_{6-1} + a_3a_{6-3} + a_5a_{6-5}$. This indicates that our general form for the co-efficient of $2n$ on the right hand side is $\sum_{k=1}^{n} a_{2k-1}a_{2n-2k+1}$, and thus returning to our equation we get

$$a_1 + \sum_{n=1}^{\infty}(2n+1)a_{2n+1}x^{2n} = 1 + \sum_{n=1}^{\infty}(\sum_{k=1}^{n} a_{2k-1}a_{2n-2k+1})x^{2n}$$

Using this we conclude that $a_1 = 1$ and $a_{2n+1} = \frac{1}{2n+1}\sum_{k=1}^{n} a_{2k-1}a_{2n-2k+1} \forall n \in \mathbb{N}$. We can note immediately that the calculation of any previous coefficients will provide no help in calculating later coefficients and so the entire sum must be calculated each loop, while also storing each co-efficient already calculated.

This means that the complexity to calculate co-efficient a_{2n+1} is $\mathcal{O}(n)$ and will be the n^{th} such calculation, making the complexity of calculating n coefficients to be $\mathcal{O}(n^2)$. Comparing this to the taylor_cos method we see that to calculate up to n coefficients of both $\cos$ and $\sin$ has complexity $\mathcal{O}(n)$. Therefore it is more efficient to calculate $\tan$ by calculating both $\cos$ and $\sin$ using Algorithm **??**, and performing division than directly using Taylor Polynomial approximation.

We would also like to be able to calculate the inverse trigonometric functions using this method, which means we need to find our McClaurin series of the inverse trigonometric functions. The simplest of these is $\tan^{-1}$, where we start by recalling that $\frac{d}{dx}\tan^{-1}x = \frac{1}{1+x^2}$ and then by integrating both sides we get:

$$\begin{aligned}
\tan^{-1}x &= \int \frac{1}{1+x^2}dx \\
&= \int (1-(-x^2))^-1 dx \\
&= \int \sum_{n=0}^{\infty}(-x^2)^n dx \qquad \text{by Equation } \textbf{??} \\
&= \int \sum_{n=0}^{\infty}(-1)^n x^{2n} dx \\
&= c + \sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}x^{2n+1}
\end{aligned}$$

As $\tan^{-1}(0) = 0$ then we see that $c = 0$ and thus gives us the following formula for $\tan^{-1}$:

$$\tan^{-1}x = \sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}x^{2n+1}$$

Now due to the restrictions from Equation **??** the above is only valid for $x \in [-1, 1]$, but we know that the domain of $\tan^{-1}$ is $x \in \mathbb{R}$. To fix this we will first recognise that $\tan^{-1}(-x) = -\tan^{-1}(x)$, so we can restrict our problem to $x \in \mathbb{R}_0^+$. Now if we take the double angle formula for $\tan$:

$$\tan(\alpha+\beta) = \frac{\tan(\alpha)+\tan(\beta)}{1-\tan(\alpha)\tan(\beta)}$$

By substituting $\alpha = \tan^{-1}(x)$ and $\beta = \tan^{-1}(x)$ into the above then we get

$$\tan^{-1}(x) + \tan^{-1}(y) = \tan^{-1}\left(\frac{x+y}{1-xy}\right)$$

Using this, suppose we are looking for $\tan^{-1}(z)$ where $z \in (1, \infty)$ and let $y = 1$, then $\tan^{-1}(y) = \frac{\pi}{4}$. We can then re-arrange the equation $z = \frac{x+1}{1-x}$ to get $x = \frac{z-1}{z+1}$; finally as $z > 1$, then $0 < x < 1$. This allows us to calculate:

$$\tan^{-1}(z) = \frac{\pi}{4} + \tan^{-1}\left(\frac{z-1}{z+1}\right)$$

In the above the calculated value is in the range $[0, 1]$ and so it is valid to use a Taylor polynomial using our McClaurin series above. This gives the following method

Algorithm 4.4.3: Taylor Method for $\tan^{-1}$

```
taylor_aTan($x \in [0,1], N \in \mathbb{N}$)
    $\mathcal{T} := 0$
    $x_2 := x^2$
    $y := x$
    $n := 0$
    while $n < N$:
        $\mathcal{T} \mapsto \mathcal{T} + \frac{1}{2n+1}y$
        $y \mapsto y \cdot x_2$
        $\mathcal{T} \mapsto \mathcal{T} - \frac{1}{2n+2}y$
        $y \mapsto y \cdot x_2$
        $n \mapsto n + 2$
    return $\mathcal{T}$
```

Similar to Algorithm **??** the error of Algorithm 4.4.3 is easy to calculate. We see that

$$\begin{aligned}\epsilon_N &= |\tan^{-1}(x) - \text{taylor\_aTan(x, N)}| \\ &\leq \frac{1}{2N+3}|x|^{2N+3} \\ &\leq \frac{1}{2N+3} && \text{as } x \leq 1\end{aligned}$$

The next function we will consider is $\sin^{-1}$, which starts it's derivation in much the same way as $\tan^{-1}$. First we start by recalling that $\frac{d}{dx}\sin^{-1}(x) = (1-x^2)^{-\frac{1}{2}}$, then by taking integrals of both sides we get the following derivation:

$$\begin{aligned}
\sin^{-1}(x) &= \int (1-x^2)^{-\frac{1}{2}} dx \\
&= \int \sum_{n=0}^{\infty} \binom{-\frac{1}{2}}{n} (-x^2)^n \\
&= c + \sum_{n=0}^{\infty} (-1)^n \left(\prod_{k=1}^{n} \frac{-\frac{1}{2} - k + 1}{k} \right) \frac{x^{2n+1}}{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{(-1)^n}{n!(2n+1)} \left(\prod_{k=1}^{n} \tfrac{1}{2} - k \right) x^{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{(-1)^{2n}}{n!(2n+1)} \left(\prod_{k=1}^{n} \frac{2k-1}{2} \right) x^{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{1}{n!(2n+1)2^n} \left(\prod_{k=1}^{n} 2k-1 \right) x^{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{1}{n!(2n+1)2^n} (1 \times 3 \times 5 \times \cdots \times (2n-1)) x^{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{1}{n!(2n+1)2^n} \times \frac{1 \times 2 \times 3 \times \cdots \times (2n)}{2 \times 4 \times \cdots \times (2n)} x^{2n+1} \\
&= c + \sum_{n=0}^{\infty} \frac{(2n)!}{(n!)^2(2n+1)4^n} x^{2n+1}
\end{aligned}$$

As $\sin^{-1}(0) = 0$ then we see that $c = 0$. Because the above is valid for $x \in (-1,1)$, and we know the values of $\sin^{-1}(-1)$ and $\sin^{-1}(1)$, then we can have the following method for evaluating $\sin^{-1}$:

Algorithm 4.4.4: Taylor Method for $\sin^{-1}$

```
taylor_aSin($x \in [-1,1], N \in \mathbb{N}$)
    if $x = 1$:
        return $\frac{\pi}{2}$
    if $x = -1$:
        return $-\frac{\pi}{2}$
    $\mathcal{S} := x$
    $x_2 := x^2$
    $y := x$
    $a := 1$
    $b := 1$
    $c := 1$
    $n := 1$
    while $n < N$:
        $a \mapsto 2n \cdot (2n-1) \cdot a$
        $b \mapsto n^2 \cdot b$
        $c \mapsto 4 \cdot c$
        $y \mapsto x_2 \cdot y$
        $\mathcal{S} \mapsto \mathcal{S} + \frac{a}{b \cdot c \cdot (2n+1)} \cdot y$
        $n \mapsto n + 1$
        return $\mathcal{S}$
```

The error for this method is similar to the $\tan^{-1}$ method, in that $\epsilon_N \leq \frac{(2(N+1))!}{((N+1)!)(2N+1)4^{N+1}}$. Finally we note that $\cos^{-1}(x) = \frac{\pi}{2} - \sin^{-1}(x)$, and thus can be calculated from a value calculated with Algorithm 4.4.4.

4.5 CORDIC

CORDIC is an algorithm that stands for **CO**rdinate **R**otation **DI**gital Computer and can be used to calculate many functions, including Trigonometric Values. The CORDIC algorithm works by utilising Matrix Rotations of unit vectors. This algorithm is less accurate than some other methods but has the advantage of being able to be implemented for fixed point real numbers in efficient ways using only addition and bit shifting.

CORDIC works by taking an initial value of $\mathbf{x}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ which can be rotated through an anti-clockwise angle of γ by the matrix

$$\begin{pmatrix} \cos\gamma & -\sin\gamma \\ \sin\gamma & \cos\gamma \end{pmatrix} = \frac{1}{\sqrt{1+\tan\gamma^2}} \begin{pmatrix} 1 & -\tan\gamma \\ \tan\gamma & 1 \end{pmatrix}$$

By taking taking smaller and smaller values of γ we can create an iterative process to find $\mathbf{x}_n$ which converges, for a given $\beta \in (-\frac{\pi}{2}, \frac{\pi}{2})$, to

$$\begin{pmatrix} \cos\beta \\ \sin\beta \end{pmatrix}$$

To do this we repeatedly add and subtract our values for γ from β to bring it as close to 0 as possible. For our purposes we wish to have a sequence $(\gamma_k : k \in [0, n] \cap \mathbb{Z})$ which will allow us to construct all angles in the range $(-\frac{\pi}{2}, \frac{\pi}{2})$ to within a known level of accuracy.

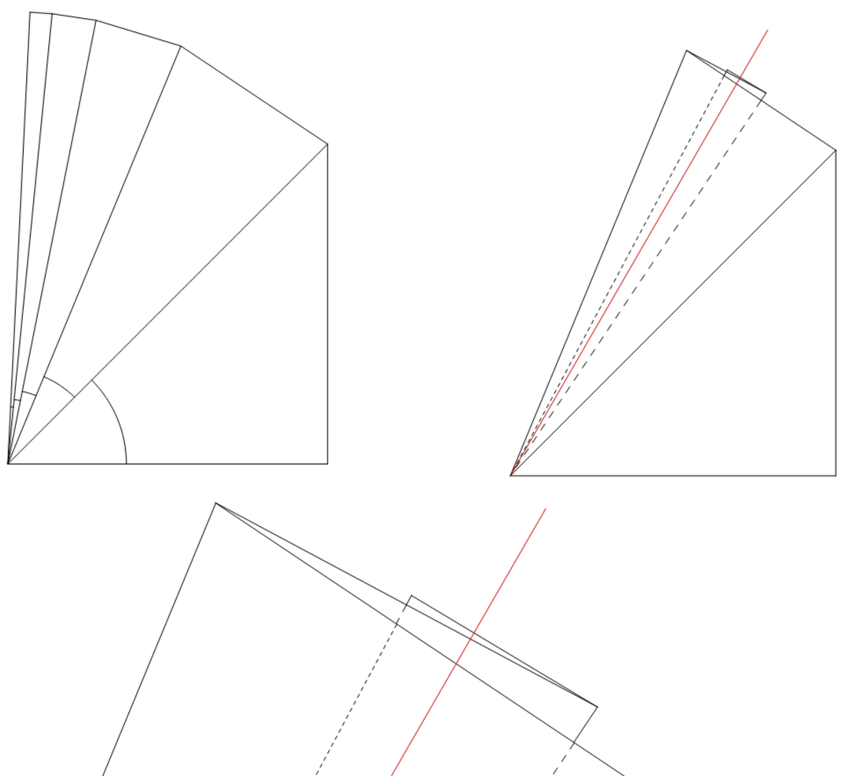
The way that this works can be thought of like a paper fan where each section is smaller than the last and to approximate the desired angle we repeatedly fold the angle back and forth. An visualisation of this is in figure **??**, which shows three views of the CORDIC fan. The top left view is the unfolded fan, the top right is the fan folded to approximate the angle shown by the red line and the view at the bottom is a close up of the previous view.
While there are many possible choices for γ_k we wish to consider $(\gamma_k : k \in [0, n] \cap \mathbb{Z})$ such that $\tan\gamma_k = 2^{-k} \forall k \in [0, n] \cap \mathbb{Z}$. We can note that the powers of 2 have a useful property, in that if $m > n \in \mathbb{N}$ we see that $\sum_{k=n}^{m-1} 2^k = 2^m - 2^n$. We wish to show that our choice for γ_k have a similar property which will be useful in showing that they are a good choice for our CORDIC algorithm.

Proposition 4.5.1. *If $m \in \mathbb{Z}_0^+$ and $n \in \mathbb{Z}^+$ such that $m > n$ and $\gamma_k = \tan^{-1}(2^-k) \forall k \in \mathbb{Z}_0^+$, then $\gamma_m < \gamma_n + \sum_{k=m+1}^{n} \gamma_k$.*

Proof. We know that $2^{-m} = 2^{-n} + \sum_{k=m+1}^{n} 2^-k$, and thus by applying $\tan^{-1}$ to both sides we get:

$$\tan^{-1} 2^{-m} = \gamma_m = \tan^{-1}(2^{-m-1} + 2^{-m-2} + \cdots + 2^{-n} + 2^{-n})$$

Figure 4.5.1: The CORDIC fan



Let $a := 2^{-m-1}+2^{-m-2}+\cdots+2^{-n}+2^{-n}$ and $b := 2^{-m-2}+\cdots+2^{-n}+2^{-n}$. Obviously $a < b$ and further we know that $\tan^{-1}$ is continuous on $[a, b]$ and differentiable on (a, b). Therefore we can apply the Mean Value Theorem from calculus to find that

$$\exists c \in (a, b) : \frac{1}{c^2+1} = \frac{\tan^{-1}(b) - \tan^{-1}(a)}{b-a}$$

By re-arranging we see that

$$\begin{aligned}\tan^{-1}(b) &= \frac{2^{-m-1}}{c^2+1} + \tan^{-1}(a)\\ &< \frac{2^{-m-1}}{2^{-2m-2}+1} + \tan^{-1}(a)\end{aligned}$$

It can be shown, by considering the series expansion of $\tan^{-1}(2^{-m-1})$, that $\frac{2^{-m-1}}{2^{-2m-2}+1} < \tan^{-1}(2^{-m-1}) \forall m \in \mathbb{Z}_0^+$; therefore we get that:

$$\tan^{-1}(b) < \tan^{-1}(2^{-m-1}) + \tan^{-1}(a)$$

Following this an using the assumed value of γ_{m+1}, we see that:

$$\gamma_m < \gamma_{m+1} + \tan^{-1}(2^{-m-2} + \cdots + 2^{-n} + 2^{-n})$$

By repeating the above process we eventually see that:

$$\gamma_m < \sum_{k=m+1}^{n-1} \gamma_k + \tan^{-1}(2^{-n} + 2^{-n})$$

In a similar manner we can repeat the above process with $a := \tan^{-1}(2^{-n})$ and $b := \tan^{-1}(2^{-n} + 2^{-n})$. This will show that:

$$\gamma_m < \gamma_n + \sum_{k=m+1}^{n} \gamma_n$$

□

Using the previous proposition we can then show that our γ_k have the property that every angle in $(-\frac{\pi}{2}, \frac{\pi}{2})$ can be approximated by either adding or subtracting successive γ_k to within a tolerance of γ_n.

Proposition 4.5.2. *If* $\gamma_k = \tan^{-1}(2^-k) \forall k \in \mathbb{Z}$, *then for any* $n \in \mathbb{N}$

$$\exists\, (c_k \in \{-1, 1\} : k \in [0, n] \cap \mathbb{Z}) \;:\; |\beta - \sum_{k=0}^{n} c_k \gamma_k| \leq \gamma_n \quad \forall\, \beta \in (-\frac{\pi}{2}, \frac{\pi}{2})$$

Proof. We let $\beta \in (-\frac{\pi}{2}, \frac{\pi}{2})$ and then will proceed by induction on $n \in \mathbb{N}$.

$\mathbf{H}(n)$**:** $\exists (c_k \in -1, 1 : k \in [0, n] \cap \mathbb{Z}) : |\beta - \sum_{k=0}^{n} c_k \gamma_k| \leq \gamma_n$

$\mathbf{H}(0)$**:** We have 4 cases to consider:

Case $\beta \in [0, \frac{\pi}{4})$**:** In this case $-\frac{\pi}{4} \leq \beta - \gamma_0 < 0$
Therefore $|\beta - \gamma_0| \leq \gamma_0$.

Case $\beta \in [\frac{\pi}{4}, \frac{\pi}{2})$**:** In this case $0 \leq \beta - \gamma_0 < \frac{\pi}{4}$
Therefore $|\beta - \gamma_0| \leq \gamma_0$.

Case $\beta \in (-\frac{\pi}{4}, 0)$**:** In this case $0 < \beta + \gamma_0 < \frac{\pi}{4}$
Therefore $|\beta - \gamma_0| < \gamma_0$.

Case $\beta \in (-\frac{\pi}{2}, -\frac{\pi}{4}$:] In this case $-\frac{\pi}{4} < \beta - \gamma_0 \leq 0$
Therefore $|\beta - \gamma_0| < \gamma_0$.

Therefore we see that $\mathrm{H}(0)$ holds true.

$\mathbf{H}(n) \implies \mathbf{H}(n+1)$**:**
By $\mathrm{H}(n)$ $\exists (c_k \in -1, 1 : k \in [0, n] \cap \mathbb{Z}) : |\beta - \sum_{k=0}^{n} c_k \gamma_k| \leq \gamma_n$; so let $\beta_n := \beta - \sum_{k=0}^{n} c_k \gamma_k$.
By Proposition 4.5.1 we know that $\gamma_n < 2\gamma_{n+1}$, and so we can proceed by case analysis:

Case $\beta_n \in [0, \gamma_{n+1})$**:**
$-\gamma_{n+1} \leq \beta_n - \gamma_{n+1} < 0 \implies |\beta - \sum_{k=0}^{n+1} c_k \gamma_k| \leq \gamma_{n+1}$ where $c_{n+1} = -1$.

Case $\beta_n \in [\gamma_{n+1}, \gamma_n)$**:**
$0 \leq \beta_n - \gamma_{n+1} < \gamma_{n+1} \implies |\beta - \sum_{k=0}^{n+1} c_k \gamma_k| \leq \gamma_{n+1}$ where $c_{n+1} = -1$.

Case $\beta_n \in [-\gamma_{n+1}, 0)$**:**
$0 \leq \beta_n + \gamma_{n+1} < \gamma_{n+1} \implies |\beta - \sum_{k=0}^{n+1} c_k \gamma_k| \leq \gamma_{n+1}$ where $c_{n+1} = 1$.

Case $\beta_n \in (-\gamma_n, -\gamma_{n+1})$**:**
$-\gamma_{n+1} < \beta_n + \gamma_{n+1} < 0 \implies |\beta - \sum_{k=0}^{n+1} c_k \gamma_k| \leq \gamma_{n+1}$ where $c_{n+1} = 1$.

Therefore as we have found a suitable c_n in all cases then we have shown that $\mathrm{H}(n) \implies \mathrm{H}(n+1)$. $\square$

With this proposition we see that our choice for γ_k is a good choice to use for the CORDIC algorithm as it covers the entire range of $(-\frac{\pi}{2}, \frac{\pi}{2})$.

Now, as stated before, the basis of our algorithm is to calculate $\begin{pmatrix} \cos\beta \\ \sin\beta \end{pmatrix}$ by using rotations of a unit vector. By putting our values for γ_k into our rotation matrix we get the following:

$$\begin{pmatrix} \cos\gamma_k & -\sin\gamma_k \\ \sin\gamma_k & \cos\gamma_k \end{pmatrix} = \frac{1}{\sqrt{1+2^{-2k}}} \begin{pmatrix} 1 & -2^{-k} \\ 2^{-k} & 1 \end{pmatrix}$$

Then if we take a current estimate of $\begin{pmatrix} \cos\beta \\ \sin\beta \end{pmatrix}$ at step k to be $\begin{pmatrix} x_n \\ y_n \end{pmatrix}$, we see that

$$\begin{pmatrix} \cos\gamma_k & -\sin\gamma_k \\ \sin\gamma_k & \cos\gamma_k \end{pmatrix} \begin{pmatrix} x_k \\ y_k \end{pmatrix} = \frac{1}{\sqrt{1+2^{-2k}}} \begin{pmatrix} x_k - 2^{-k} y_k \\ y_k + 2^{-k} x_k \end{pmatrix}$$

This gives a very simple formula for the update of x_k and y_k, which can be used as the basis of the CORDIC Algorithm.

As seen in our proof of Proposition 4.5.2, we can approximate our desire angle at step n by keeping a track of $\beta_n := \beta - \sum_{k=0}^{n-1} c_k \gamma_k$. At step n we then have $\beta_{n+1} = \beta_n - \gamma_n$ if $\beta_{n+1} \geq 0$, and $\beta_{n+1} = \beta_n + \gamma_n$ otherwise. This leads us to the general implementation of CORDIC for Trigonometric Functions:

Algorithm 4.5.1: General Cordic

```
CORDIC($\beta \in (-\frac{\pi}{2}, \frac{\pi}{2}), n \in \mathbb{N}$):
    $x := 1$
    $y := 0$
    $k := 0$
    while $k < n$:
        if $\beta \geq 0$:
            $t := x$
            $x \mapsto \frac{1}{\sqrt{1+2^{-2k}}}(x - 2^{-k}y)$
            $y \mapsto \frac{1}{\sqrt{1+2^{-2k}}}(y + 2^{-k}t)$
            $\beta \mapsto \beta - \tan^{-1}(2^{-k})$
        else:
            $t := x$
            $x \mapsto \frac{1}{\sqrt{1+2^{-2k}}}(x + 2^{-k}y)$
            $y \mapsto \frac{1}{\sqrt{1+2^{-2k}}}(y - 2^{-k}t)$
            $\beta \mapsto \beta + \tan^{-1}(2^{-k})$
        $k \mapsto k + 1$
    return $(x, y)^T$
```

There are few improvements we can make on the general algorithm, however if we start to consider implementations of the algorithm we can find several ways to make our algorithm more efficient.

First we consider the representation of our values in the program, and while in many of the previous algorithms a floating point `double` value, as described in Section **??**, we will see here that we wish to use a fixed point representation. If we have a fixed point representation of our values, then we are using an N bit integer to represent the value in question, with a fixed number of bits set aside for the integer part and the remainder for the fractional part. In this case the process of addition, subtraction as well as multiplication and division by powers of 2 is the same as that for integers.

In particular as our values never exceed the range of $(-2, 2)$, then we can use $N-2$ bits of our N bit integer to be the fractional part; this gives us a maximum precision of 2^{2-N}. Further as we are only performing multiplication and division by two, this operation can be performed by bit shifting the values, which is much quicker than actual integer multiplication.

Second we can pre calculate all of the values needed for the algorithm to trade storage space for a reduction in computational complexity. The values which we need to pre-calculate are $\gamma_k = \tan^{-1}(2^{-k})$ and $\frac{1}{\sqrt{1+2^{-2k}}}$ for $k \in [0, n) \cap \mathbb{Z}$. The first thing to note about this is that instead of calculating the multiplication $\frac{1}{\sqrt{1+2^{-2k}}}$ at each stage we can actually take this value out of the loops and pre-calculate $\prod_{k=0}^{n} \frac{1}{\sqrt{1+2^{-2k}}}$ for $k \in [0, n) \cap \mathbb{Z}$. Using these pre calculated products we can then replace $x := 1$ with $x := \prod_{k=0}^{n} \frac{1}{\sqrt{1+2^{-2k}}}$ in the initialisation stage.

Now to consider an actual implementation, suppose we are using the 16 bit integer `int16_t` to represent our values; which will have the leading two bits represent the integer part and the remaining 14 bits represent the fractional part. In this case the level of precision is $2^{-14} = 0.00006103515625$ and further we can show that as $\gamma_{14} = \tan^{-1}(2^{-14}) \approx 2^{-14}$; therefore the largest we will choose $n := 14$ to ensure the maximum possible accuracy, without performing excessive calculations

This means we can simplify our algorithm further by calculating only $\prod_{k=0}^{14} \frac{1}{\sqrt{1+2^{-2k}}}$ and $\tan^{-1}(2^{-k}) \forall k \in [0, 14] \cap \mathbb{Z}$. One further note is that these values then need to be converted to approximations in our 16 bit fixed point representation. The first value is:

$$\begin{aligned}
\prod_{k=0}^{14} \frac{1}{\sqrt{1+2^{-2k}}} &= 0.607252936517010234128971242079738890 82\ldots \\
&\approx 00.10011011011101_2 \\
&= 26\mathrm{dd}_{16}
\end{aligned}$$

Below is a table of all the angles in the relevant formats

γ_k	Exact Form	Binary	Hexadecimal
γ_0	$0.7853981633\ldots$	00.11001001000011_2	3243_{16}
γ_1	$0.4636476090\ldots$	00.01110110101100_2	1dac_{16}
γ_2	$0.2449786631\ldots$	00.00111110101101_2	0fad_{16}
γ_3	$0.1243549945\ldots$	00.00011111110101_2	$07\text{f}5_{16}$
γ_4	$0.0624188099\ldots$	00.00001111111110_2	03fe_{16}
γ_5	$0.0312398334\ldots$	00.00000111111111_2	01ff_{16}
γ_6	$0.0156237286\ldots$	00.00000100000000_2	0100_{16}
γ_7	$0.0078123410\ldots$	00.00000010000000_2	0080_{16}
γ_8	$0.0039062301\ldots$	00.00000001000000_2	0040_{16}
γ_9	$0.0019531225\ldots$	00.00000000100000_2	0020_{16}
γ_{10}	$0.0009765621\ldots$	00.00000000010000_2	0010_{16}
γ_{11}	$0.0004882812\ldots$	00.00000000001000_2	0008_{16}
γ_{12}	$0.0002441406\ldots$	00.00000000000100_2	0004_{16}
γ_{13}	$0.0001220703\ldots$	00.00000000000010_2	0002_{16}
γ_{14}	$0.0000610351\ldots$	00.00000000000001_2	0001_{16}

This allows us to then write the following method in C to calculate both $\cos\beta$ and $\sin\beta$, provided $\beta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ is given in 16 bit fixed point representation:

```c
int16_t *cordic_16(int16_t beta)
{
        const int16_t GAMMA = {0x3243, 0x1dac, 0x0fad, 0x07f5, 0x03fe,
                               0x01ff, 0x0100, 0x0080, 0x0040, 0x0020,
                               0x0010, 0x0008, 0x0004, 0x0002, 0x0001};

        int16_t x = 0x26dd, y = 0x0000, t, result;

        for(int k = 0; k <= 14; ++k)
        {
                t = x;
                if(beta >= 0)
                {
                        beta -= GAMMA[k];
                        x = x - (y >> k);
                        y = y + (t >> k);
                }
                else
                {
                        beta += GAMMA[k];
                        x = x + (y >> k);
                        y = y - (t >> k);
                }
        }

        //This line is required by C to allow the value to be returned
        result = malloc(2 * sizeof(int16_t));

        result[0] = x;
        result[1] = y;
        return result;
}
```

As can easily be seen in the algorithm the number of calculations each iteration is constant, and the number of iterations is fixed at 15. This means that the algorithm is an $\mathcal{O}(1)$ algorithm, and guarantees an answer accurate to 4 decimal places as $2^{-14} < 10^{-4}$. Further as the only calculations are integer addition, subtraction and bit shifting this method executes extremely quickly.

Similar methods exist for other fixed length formats such as using `int32_t` or `int64_t`. To examine in more detail how the method converges we will consider an implementation using `int64_t`, which will be approximating $\cos(0.5)$. The code used is included in the Appendix **??** and can perform the calculations with $n \leq 63$. Below are some of the functions approximations for different values of n:

n	Output with bold accurate digits
1	**0.**70710678118654757273731
2	**0.**94868329805051376801827
3	**0.8**4366148773210747346951
4	**0.**90373783889353875853345
5	**0.87**527458786899225984257
6	**0.8**8995346811933362385360
...	...
19	**0.87758**301847694786257392
20	**0.877582**10404530012649360
21	**0.877582561**26152311971111
22	**0.877582**78986933524468128
...	...
53	**0.8775825618903727**5873943
54	**0.877582561890372**64771712
55	**0.8775825618903727**5873943
56	**0.8775825618903727**5873943
...	...
63	**0.8775825618903727**5873943

This table shows us several interesting features of the algorithm, the first being that while there are points at which a certain number of decimal places are guaranteed; before that point the number of decimal places of accuracy can vary, such as in the first few iterations. As we know that the error after n iterations is at most $\gamma_n = \tan^{-1}(2^{-n})$, then we can guarantee that we have at least d decimal places of accuracy if we use as least $\log_2(\cot(10^{-d}))$ iterations.

Second there are some values of n which have uncharacteristically close approximations of the actual value, such as the case when 21 iterations are used. This arises due to the algorithm finding a good approximation for β, but then successive numbers of iterations move away from this value, thus once more decreasing the number of decimal digits of accuracy.

Finally at the end of the able we see that from 55 iterations onwards, the results do not get any more accurate. It turns out this is due to the program converting the `int64_t` fixed point values into `double` values, which typically have an precision of around 2^{-55}. If we instead modify the program to use a more precise floating point representation we see that the 53 to 56 section of the table becomes:

n	Output with bold accurate digits
53	**0.8775825618903727**3965747
54	**0.877582561890372**68653156
55	**0.87758256189037271**298609
56	**0.8775825618903727**2621336

This is much more in line with what we would expect to see from the known error of the algorithm.

Now another use of CORDIC is to effectively run it in reverse, which will allow us to calculate the Inverse Trigonometric functions. To do this we will start by considering the method for calculating $\tan^{-1}$, and then use trigonometric identities to calculate both $\cos^{-1}$ and $\sin^{-1}$.

To accomplish this we will be fixing some initial values for $\sin\theta$ and $\cos\theta$, and then running the CORDIC algorithm to move the approximation of $\sin\theta$ towards zero. In doing this we will effectively run our algorithm in reverse, and if we keep track of the angles that we rotate through we can find $\tan^{-1}$.

We know that $\tan\theta = \frac{\sin\theta}{\cos\theta}$, which means that if we have a current approximation $\begin{pmatrix} x_k \\ y_k \end{pmatrix}$ then $\frac{y_k}{x_k} \approx \tan\theta$. Using this, if we have an input of $\tan\theta = z$ then we can take our initial values to be $x_0 := \frac{1}{2}$ and $y_0 := \frac{z}{2}$. This has the desired property that $\frac{y_0}{x_0} = z$, and if we have y_n tending to 0 then the angle we approximate in the process will be θ.

If we again consider a 16 bit fixed point implementation for our algorithm we can implement it as follows:

```c
int16_t *cordic_atan_16(int16_t z)
{
        const int16_t GAMMA = {0x3243, 0x1dac, 0x0fad, 0x07f5, 0x03fe,
                               0x01ff, 0x0100, 0x0080, 0x0040, 0x0020,
                               0x0010, 0x0008, 0x0004, 0x0002, 0x0001};

        int16_t x = 0x2000, y = z >> 1, t, theta;

        for(int k = 0; k <= 14; ++k)
        {
                t = x;
                if(y < 0)
                {
                        theta -= GAMMA[k];
                        x = x - (y >> k);
                        y = y + (t >> k);
                }
                else
                {
                        theta += GAMMA[k];
                        x = x + (y >> k);
                        y = y - (t >> k);
                }
        }

        return theta;
}
```

Similar to our considerations when dealing with the Taylor method of calculating $\tan^{-1}$, we need to ensure that the input value is not too large, and so can perform the same transformations to the value to ensure we are always calculating a value in the range $[0, 1)$. Using this we can then use the identities $\sin^{-1}(z) = \tan^{-1}(\frac{z}{\sqrt{1-z^2}})$ and $\cos^{-1}(z) = \tan^{-1}(\frac{\sqrt{1-z^2}}{z})$.

Obviously there are basic exceptional values that need to be checked for, in particular $\cos^{-1}(0) = \frac{\pi}{2}$, and $\sin^{-1}(\pm 1) = \pm\frac{\pi}{2}$. If these values are checked before hand then we are never dividing by 0, $z \in [-1, 1] \cap \mathbb{Z}$, and thus we have a complete algorithm, that calculates the inverse Trigonometric Functions.

This method, like the CORDIC method for the regular Trigonometric Functions, has an approximation that is accurate to within γ_n. Thus for our 16 bit implementation, the output will be accurate to within an error of $2^{-14} = 0.00006103515625$, in particular guaranteeing at least 4 decimal places of accuracy. A final note is that the Inverse Trigonometric Functions, again much like the regular CORDIC algorithm, is an $\mathcal{O}(1)$ algorithm with simple calculations, making the algorithm extremely efficient.

4.6 Comparison of Methods

We have observed three different methods for calculating the Trigonometric Functions, as well as their inverses and so should compare their efficiency and accuracy properties.

First we will compare how quickly each algorithm approaches the correct value for different inputs of n, and using $\theta = 0.5$. The comparison will use `double` values for computation, so that all three methods may be equally compared. The table below compares the convergence of $\cos\theta$, with the bold digits being the correct digits found:

n	geometric_Cos(0.5, n)	taylor_Cos(0.5, n)	CORDIC(0.5, n)
1	**0.87**695312500000000	**1.**000000000000000000	**0.**707106781186547572
2	**0.877**426177263259887	**0.877**604166666666629	**0.**948683298050513768
3	**0.8775**43526076081437	**0.877**604166666666629	**0.8**43661487732107473
4	**0.8775**72806699400187	**0.87758256**2158978118	**0.**903737838893538758
5	**0.87758**0123327654892	**0.87758256**2158978118	**0.87**5274587868992259
6	**0.87758**1952264380182	**0.877582561890037**3424	**0.8**89953468119333623
7	**0.877582**409484792491	**0.877582561890037**3424	**0.8**82719918613777410
8	**0.8775825**23789035007	**0.87758256189037272**58	**0.87**9022003513595939
9	**0.8775825**52365041901	**0.87758256189037272**58	**0.877**152884812089639
10	**0.8775825**59509040183	**0.87758256189037272**58	**0.87**8089122532394572

This table demonstrates that taylor_Cos has the fastest convergence, and also demonstrates the staggered increase in accuracy as each step of the algorithm calculates two updates to $\cos\theta$, and thus the output only gets more accurate every other value of n. The geometric_Cos method has the second best convergence, while the CORDIC algorithm lags behind, having inconsistent convergence as measured in correct digits.

Next we will note that all algorithms can guarantee 10 digits of accuracy in a fixed number of steps. In particular we can guarantee 10 digits of accuracy for geometric_Cos when $n \geq 16$, taylor_Cos when $n \geq 8$ and CORDIC when $n \geq 34$. Using the lower bounds of each of these values for n we can directly compare the speed of the methods.

To compare the methods we will be testing 1000 random values in the range $[0, \frac{\pi}{2})$ for which we will calculate the cosine of with each method 100000 times. This will then also be compared to the standard C implementation of the $\cos$ function, available in `math.h`. The results of my personal testing follow, where the given times are for individual values, not individual method execution times:

	`geometric_cos`	`taylor_cos`	`cordic_cos`	`builtin_cos`
Total time:	16.029s	7.937s	21.471s	0.243s
Average time:	0.016s	0.007s	0.021s	0.000s
Minimum time:	0.015s	0.007s	0.020s	0.000s
Maximum time:	0.022s	0.013s	0.030s	0.000s

These values show that the fastest algorithm that we have discussed is Algorithm **??** (taylor_Cos), while the slowest is the CORDIC algorithm. However all of our Algorithms are much less efficient than the built-in `cos` function of C. It turns out this discrepancy is due to inefficient implementation as the `cos` function also uses a Taylor approximation, but is implemented in a much lower-level method that optimises the execution of the code.

TODO: Ref the C code
TODO: https://sourceware.org/git/?p=glibc.git;a=blob;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEADl281

Next we will compare our methods for the Inverse Trigonometric Functions, starting with how they converge to the correct value, as detailed in the following table:

n	geometric_aCos(0.5, n)	taylor_aCos(0.5, n)	CORDIC(0.5, n)
1	**2.3**51425307918200591	**2.**270796326794896735	**2.3**56194490192344837
2	**2.34**7503635391542609	**2.**327962993461563101	**1.**892546881191538687
3	**2.346**521397812842746	**2.**340568243461563113	**2.**137525544318402914
4	**2.346**275724597314926	**2.34**4244774711563117	**2.**261880538865164602
5	**2.346**214299177873829	**2.34**5470795757570225	**2.3**24299348861121661
6	**2.34619**8942378459939	**2.34**5913166442261221	**2.3**55539182291389810
7	**2.34619**5103149716576	**2.346**081295659538934	**2.3**39915453670913247
8	**2.34619**4143336564508	**2.3461**47594614218956	**2.34**7727794731014228
9	**2.346193**903386887935	**2.3461**74467628018511	**2.34**3821564599047224
10	**2.3461938**43452078375	**2.3461**85594784405026	**2.34**5774687115525836

Here we see for the inverse trigonometric functions the convergence speed has been altered with the Geometric method now having the fastest convergence, the Taylor Method converges much slower and the CORDIC method is more stable. One interesting behaviour that emerges for larger values of n in the geometric_aCos is demonstrated in the following table:

n	geometric_aCos(0.5, n)
13	**2.346193822**083380897
14	**2.3461938**12716280469
15	**2.346193**737779483257
...	...
22	**2.346**097524754926944
23	**2.34**1202123910687049
24	**2.3**51023238547698124

This behaviour arises due to the use of `double` to calculate values of very small magnitude, this causes the value to become effectively 0 and thus lead to the inaccuracies seen. If we use a higher precision representation for the calculations we get the following table instead:

n	geometric_aCos(0.5, n)
13	**2.346193823**718087586
14	**2.3461938234**83759158
15	**2.3461938234**25177051
...	...
22	**2.3461938234056**50874
23	**2.346193823405649**980
24	**2.346193823405649**757

With this we see that Algorithm **??** continues in the same pattern as before and is actually correct. So we may again look to time our functions to test their efficiency as compared to each other. To do this we will again use 1000 random values, this time in the range $(-1, 1)$, each of which we will calculate $\cos^{-1}$ using each method 100000 times. We note that the algorithms can guarantee 10 decimal places of accuracy for different values of n, in particular geometric_aCos when $n \geq 18$, taylor_aCos when $n \geq 30$ and CORDIC when $n \geq 50$.

	`geometric_cos`	`taylor_cos`	`cordic_cos`	`builtin_cos`
Total time:	27.273s	14.358s	29.142s	2.143s
Average time:	0.027s	0.014s	0.029s	0.002s
Minimum time:	0.026s	0.014s	0.028s	0.001s
Maximum time:	0.033s	0.018s	0.032s	0.006s

Again this table shows that the Taylor method is the quickest of those analysed and the CORDIC method is the slowest, however they also both are much slower than the built in methods. One thing to note is that the inverse trigonometric functions are simply less efficient to calculate, as can be seen in the execution time of the built-in method, which appears to be two orders of magnitude greater than the corresponding trigonometric method.

We conclude that for most implementations the Taylor method is the most appropriate method to use to ensure a high accuracy quickly. However the CORDIC algorithm is of use when more advanced features such as floating point type values, or hardware multipliers are not present; further it is possible to create hardware implementations of the CORDIC algorithm which can even further speed up the calculations.

5 Logarithms and Exponentials

Exponentiation is the operation of calculating x^y where x and y are members of some field, for the purposes of this document we will be considering $x, y \in \mathbb{R}$. This operation is widely used by many different branches of mathematics and industry, for example many real world phenomena can be modelled by exponentials; we would therefore like to be able to calculate x^y quickly and efficiently.

The first thing we consider is that x^y when $x \in$ and $y \in \mathbb{R} \setminus \mathbb{Z}$ is not well-defined on $\mathbb{R}$, and requires consideration of the function on the complex plane. Due to this we will not be considering negative numbers to non-integer bases; in particular, unless stated otherwise, we will be assuming that $x \in \mathbb{R}_0^+$.

Now we also know that $x^{-y} = \frac{1}{x^y}$ when $y \in \mathbb{R}$, and as such we will also be restricting this section to the assumption that $y \in \mathbb{R}_0^+$. Further we consider the following facts:

$$x^0 = 1 \forall x \in \mathbb{R}_0^+$$
$$0^y = 0 \forall y \in \mathbb{R}^+$$

If we take out these known trivial cases then we can restrict this section to considering only $(x, y) \in \mathbb{R}^{+2}$.

Now if we have $y \in \mathbb{R}^+$ then it follows that $\exists (a, b) \in \mathbb{Z}_0^+ \times [0, 1)$ such that $y = a + b$. This allows us to use the identity that $x^{m+n} = x^m x^n$ to consider the following two cases separately:

$$x^a : a \in \mathbb{Z}_0^+ \tag{5.0.1}$$
$$x^b : b \in [0, 1) \tag{5.0.2}$$

5.1 Calculating x^a

As we know that $a \in \mathbb{Z}_0^+$, then we know that $x^a = \underbrace{x \times \cdots \times x}_{a}$; i.e. the problem is equivalent to finding x multiplied with itself a times. As we are only dealing with $a \in \mathbb{Z}_0^+$, then we will be considering $x \in \mathbb{R}$ as we can calculate exponentials of negative numbers.

The naive way to go about calculating x^a is to simply perform the multiplication of x by itself a times. The algorithm for that can be seen below:

Algorithm 5.1.1: Naive integer exponentiation

```
naive_int_exp($x \in \mathbb{R}, a \in \mathbb{Z}_0^+$):
    $n := 0$
    $z := 1$
    while $n < a$:
        $z \mapsto x \cdot z$
    return $z$
```

This algorithm is very simple and has complexity of $\mathcal{O}(a)$, which is a reasonable complexity, but still has the chance to grow large as a grows. Instead we can consider a more informed approach, in particular we know that either $2 \mid a$ or $2 \nmid a$, which then gives us the following:

$$x^a = \begin{cases} (x^2)^{\frac{a}{2}} & : \ 2 \mid a \\ x \cdot (x^2)^{\frac{a-1}{2}} & : \ 2 \nmid a \end{cases}$$

We can use this fact to build a recursive method of calculating x^a, where we repeatedly call the method from within itself. To ensure the method ends correctly we need to identify a base case for the recursion, i.e. where the process stops and returns the correct value. We can see that eventually the above will reach the point where $a = 0$, in which case we know that $x^0 = 1$; this will be the base case of our recursion.

We want to ensure that the algorithm will terminate, which we can do by seeing that it terminates when $a = 0$ and then considering $a \in \mathbb{Z}^+$. Now if $2 \mid a$ then $\frac{a}{2} \in \mathbb{Z}^+$ and also $\frac{a}{2} < a$, similarly if $2 \nmid a$ then $\frac{a-1}{2} \in \mathbb{Z}_0^+$ because $a \geq 1$ and also $\frac{a-1}{2} < a$. Thus we see that the sequence produced by $a \in \mathbb{Z}^+$ is a strictly decreasing sequence that is bounded below by 0 and thus we must eventually reach 0, meaning the algorithm terminates.

Instead of a recursive algorithm that calls itself the algorithm below is an iterative version which performs the same function:

Algorithm 5.1.2: Exponentiation by squaring

```
exp_by_squaring($x \in \mathbb{R}, a \in \mathbb{Z}_0^+$):
    $n := a$
    $z := 1$
    $\hat{x} := x$
    while $n > 0$:
        if $2 \nmid n$:
            $z \mapsto \hat{x} \times z$
            $n \mapsto n - 1$
        $\hat{x} \mapsto \hat{x}^2$
        $n \mapsto \frac{n}{2}$
    return $z$
```

This algorithm is much more efficient than Algorithm 5.1.1 due to the number of times the inner loop is executed. The inner loop drives a towards 0 by dividing by 2 each step, this means that as $a = \mathcal{O}(2^{\log_2(a)})$, then this goal is achieved in only $\log_2(a)$ loops. Therefore the complexity of this algorithm is $\mathcal{O}(\log_2(a))$, which is an improvement upon the previous algorithm's complexity of $\mathcal{O}(a)$.

To see this difference in efficiency in action the following table shows the times taken for each method when comparing 1000 different pairs of values $(x, a) \in [0, 10] \times ([0, 100] \cap \mathbb{Z})$. With these values we calculated x^a using both methods 100000 times to get the following results:

	`naive_int_exp`	`squaring_int_exp`
Total time:	16.800s	2.593s
Average time:	0.016s	0.002s
Minimum time:	0.000s	0.000s
Maximum time:	0.037s	0.004s

With this we will move on to further subsections as there are few improvements that can be made on an $\mathcal{O}(\log_2(a))$ algorithm, particularly in this instance.

5.2 Calculating x^b

If we have $b \in (0,1)$, then we obviously can't use the our previous subsection for calculating x^y. The most common way of calculating such exponentiation is by considering that $x = e^{\ln(x)}$ and thus $x^b = (e^{\ln(x)})^b = e^{b\ln(x)}$; however this now raises the problem of how to calculate both e^α and $\ln(\beta)$. The following will deal with how to calculate these values and thus use them in conjunction to calculate x^b.

5.3 Naive Method

The mathematical constant e has been known since the early 1600s and was originally calculated by Jacob Bernoulli, and was studied bye Leonhard Euler, where it appeared in Euler's Mechanica in 1736. While several possible equivalent definitions of e exist the most common such definition is that $e := \lim_{n\to\infty}(1+\frac{1}{n})^n$.

If we now consider the definition of e and also consider e^x, then we can show that $e^\alpha = \lim_{n\to\infty}(1+\frac{x}{n})^n$. This gives us our first basic method of how to calculate e^x:

Algorithm 5.3.1: Baisc Method for calculating e^α

```
basic_exp($x \in \mathbb{R}, n \in \mathbb{N}$)
    return $(1+\frac{x}{n})^n$
```

If we consider $(1+\frac{x}{n})^n$ as a function of a continuous n then we can find the following derivation:

$$\begin{aligned}
\frac{d}{dn}\left[(1+\frac{x}{n})^n\right] &= (1+\frac{x}{n})^n \frac{d}{dn}\left[n\ln(1+\frac{x}{n})\right] \\
&= (1+\frac{x}{n})^n(\frac{d}{dn}[n]\ln(1+\frac{x}{n}) + n\frac{d}{dn}\left[\ln(1+\frac{x}{n})\right]) \\
&= (1+\frac{x}{n})^n(\ln(1+\frac{x}{n}) + \frac{n}{1+\frac{x}{n}}\frac{d}{dn}(1+\frac{x}{n})) \\
&= (1+\frac{x}{n})^n(\ln(1+\frac{x}{n}) - \frac{x}{n+x}) \\
&= \frac{(1+\frac{x}{n})^n}{x+n}((x+n)\ln(1+\frac{x}{n}) - x)
\end{aligned}$$

By the last line of this we can see that because $(x,n) \in \mathbb{R}^{+2}$ then $\ln(1+\frac{x}{n} > 0$ and thus we conclude that $(x+n)\ln(1+\frac{x}{n}) - x > 0$. Therefore we see that $\frac{d}{dn}\left[(1+\frac{x}{n})^n\right] > 0$ for all $(x,n) \in \mathbb{R}^{+2}$, and in particular this means that $(1+\frac{x}{n})^n < (1+\frac{x}{n+1})^{n+1}\forall n \in \mathbb{N}$.

One consequence of this is that $(1+\frac{x}{n})^n < e^x\forall n \in \mathbb{N}$, therefore we can define the error of algorithm 5.3.1 as $\epsilon_N := |e^x - (1+\frac{x}{n})^n| = e^x - (1+\frac{x}{n})^n$. Now as $\lim_{n\to\infty}(1+\frac{x}{n})^n = e^x$ then we see that $\lim_{n\to\infty}\epsilon_n = 0$, and thus our algorithm is correct and valid for approximating e^x.

Next we see that this method, while simple, approximates e^x very poorly. In particular the table below shows the approximation of $e^{0.75}$ for different values of n, where the bold digits are the correctly approximated digits.

n	Approximation of $e^{0.75}$
1	1.800000000000000044
10	**2.**158924997272786787
100	**2.2**18468215957572747
1000	**2.22**4829248807374831
10000	**2.225**469716120127850
100000	**2.2255**33806810873500
1000000	**2.225540**216319864358
10000000	**2.225540**857275162929
100000000	**2.22554092**1370736781
1000000000	**2.22554092**7780294606

With this table we see that the method very poorly approximates e^x, requiring a very large n to get just a few digits of accuracy. While this does not require more calculations from the method, requiring this large a value of n can lead to inaccuracies in the implementation of the algorithm using `double` data types in C.

In general there are better methods of approximating e^x and also $\ln(x)$, which while requiring more calculations are much more accurate than the most basic method presented here.

5.4 Taylor Series Method

If we take the elementary result from calculus that $\frac{d}{dx}e^x = e^x$, then we can calculate the McClaurin series of e^x. By the definition of a McClaurin series we know that the series expansion of e^x about 0 is

$$\sum_{k=0}^{\infty} \frac{d^{(k)}}{dx^k}[e^x](0)\frac{x^k}{k!}$$

As $\frac{d^{(k)}}{dx^k}[e^x] = e^x \forall k \in \mathbb{Z}_0^+$ and $e^0 = 1$ then we see that the series becomes

$$\sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Using this we see that $e^x \approx \sum_{k=0}^{n} \frac{x^k}{k!}$, which gives the following method for approximating e^x:

Algorithm 5.4.1: Taylor Method for calculating e^x

```
taylor_exp($x \in \mathbb{R}, n \in \mathbb{Z}_0^+$)
    $t = 1$
    $z = 1$
    $k = 1$
    while $k < n$:
        $t \mapsto \frac{t \cdot x}{n}$
        $z \mapsto z + t$
        $k \mapsto k + 1$
    return $z$
```

This allows us to calculate e^x more efficiently as we can note that the error of the approximation is easy to approximate. We know that $\epsilon_n := |e^x - \sum_{k=0}^{n} \frac{x^k}{k!}| \leq \frac{|x|^{n+1}}{(n+1)!}$ for all $n \in \mathbb{Z}_0^+$.

While we can't guarantee the size of x in general we will consider $x \in (0, 1)$ for the purposes of analysing this function.

As $x \in (0, 1)$ then it follows that $x < 1$ and thus we can see that $\epsilon_n < \frac{1}{n!} \forall n \in \mathbb{Z}_0^+$. Using this we can see that if we wish to use our method such that the error is at most $\tau_d := 10^{-d}$, then we need to find $\in \mathbb{Z}_0^+ : \frac{1}{n!} < \tau_d$. The table below shows some values for (n, d) pairs such that n is the smallest positive integer such that $\frac{1}{n!} < \tau_d$:

$d \in \mathbb{N}$	$\arg\min \left\{n \in \mathbb{N} : n! > 10^d\right\}$
1	4
10	14
100	70
1000	450

Thus we see that that we can guarantee 100 digits of accuracy with an input of $n \geq 70$ and 1000 digits of accuracy with $n \geq 450$, this is much less than our previous method where an input of $n = 1000$ only gave 2 decimal places of accuracy.

The inverse of the function $z = e^x$ is the logarithm function $\ln(z) = x$, which we can again consider for Taylor Series expansion. First we will show the result from calculus that $\frac{d}{dx}[\ln(x)] = \frac{1}{x}$:

Proposition 5.4.1.

$$\frac{d}{dx}[\ln(x)] = \frac{1}{x}$$

Proof. We will prove this from the first principles using the definition that $\frac{d}{dx}[f(x)] = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$

$$\begin{aligned}\frac{d}{dx}[\ln(x)] &= \lim_{h\to 0} \frac{\ln(x+h) - \ln(x)}{h} \\ &= \lim_{h\to 0} \frac{\ln(1 + \frac{h}{x})}{h} \\ &= \lim_{h\to 0} \ln\left((1 + \frac{h}{x})^{\frac{1}{h}}\right)\end{aligned}$$

If we let $u := \frac{h}{x}$, then we get that $ux = h$ and $\frac{1}{h} = \frac{1}{ux}$. Also $\lim_{h\to 0} u = 0$, and so we get the following:

$$\begin{aligned}\frac{d}{dx}[\ln(x)] &= \lim_{u\to 0} \ln((1+u)^{\frac{1}{ux}}) \\ &= \frac{1}{x} \lim_{u\to 0} \ln((1+u)^{1/u})\end{aligned}$$

If we now let $n := \frac{1}{u}$ and consider that $\lim_{u\to 0} n = \infty$, then our derivative becomes:

$$\begin{aligned}
\frac{d}{dx}\ln(x) &= \frac{1}{x}\lim_{n\to\infty}\ln((1+\frac{1}{n})^n) \\
&= \frac{1}{x}\ln(\lim_{n\to\infty}(1+\frac{1}{n})^n) \\
&= \frac{1}{x}\ln(e) && \text{by the definition of } e \\
&= \frac{1}{x}
\end{aligned}$$

□

Now we know that $\frac{d^k}{dx^k}[\frac{1}{x}] = (-1)^k k! x^{-k-1}$, and thus we can build up a Taylor Series expansion. In this case, rather than centring the series about $x = 0$ for a McClaurin series we can instead centre the series around $x = 1$ which gives the following series expansion for $\ln(x)$:

$$\begin{aligned}
\sum_{k=0}^{\infty}\frac{\frac{d^k}{dx^k}[\ln(x)](1)}{k!}(x-1)^k &= \ln(1) + \sum_{k=1}^{\infty}\frac{\frac{d^{k-1}}{dx^{k-1}}[x^{-1}](1)}{k!}(x-1)^k \\
&= \sum_{k=1}^{\infty}\frac{[(-1)^{k-1}(k-1)!x^{-k}](1)}{k!}(x-1)^k \\
&= \sum_{k=1}^{\infty}\frac{(-1)^{k-1}}{k}(x-1)^k \\
&= -\sum_{k=1}^{\infty}\frac{(1-x)^k}{k}
\end{aligned}$$

We know that $\ln(x) = -\sum_{k=1}^{\infty}\frac{(1-x)^k}{k}$ when the series $\sum_{k=1}^{\infty}\frac{(1-x)^k}{k}$ converges. We thus need to know when the sum converges.

Proposition 5.4.2. *The series $\sum_{k=1}^{\infty}\frac{(1-x)^k}{k}$ converges when $x \in (0,2)$.*

Proof. Let $a_k := \frac{(1-x)^k}{k}$. We will proceed by using the ratio test to show when the series converges absolutely. The test states that the series converges when $\lim_{k\to}\left|\frac{a_{k+1}}{a_k}\right| < 1$.

Now we can consider the following derivation:

$$\begin{aligned}
\lim_{k\to\infty}\left|\frac{a_{k+1}}{a_k}\right| &= \lim_{k\to\infty}\left|\frac{\frac{1}{k+1}(1-x)^{k+1}}{\frac{1}{k}(1-x)^k}\right| \\
&= \lim_{k\to\infty}\left|\frac{k}{k+1}(1-x)\right| \\
&= |1-x|\lim_{k\to\infty}\left|\frac{k}{k+1}\right| \\
&= |1-x|
\end{aligned}$$

Therefore our series converges when:

$$\begin{aligned}|1-x|<1 &\iff -1<1-x<1\\ &\iff -1<x-1<1\\ &\iff 0<x<2\end{aligned}$$

Hence $\sum_{k=1}^{\infty}\frac{(1-x)^k}{k}$ converges when $x\in(0,2)$. $\square$

Now as we can't know if $x\in(0,2)$ then we can consider that $\forall x\in\mathbb{R}^+\exists(a,b)\in[\frac{1}{2},1)\times\mathbb{Z}:$ $x=a\cdot 2^b$; thus we see that $\ln(x)=\ln(a\cdot 2^b)=b\ln(2)+\ln(a)$. As previously noted in Section **??** this operation, while theoretically complex, is simple to calculate for most computers by how the represent floating point values.

With this we can then use the following method to approximate $\ln(x)$ by the Taylor polynomial $-\sum_{k=1}^{n}\frac{(1-x)^k}{k}$:

Algorithm 5.4.2: Taylor Method for calculating $\ln(x)$

```
taylor_nat_log($x\in\mathbb{R}^+, n\in\mathbb{N}$):
    Find $(a,b)\in[\frac{1}{2},1)\times\mathbb{Z}$ such that $x=a\cdot 2^b$
    $y := 1-a$
    $t := y$
    $z := y$
    $k := 1$
    while $k<n$:
        $t\mapsto t\cdot y$
        $z\mapsto z+\frac{t}{k}$
        $k\mapsto k+1$
    return $b\ln(2)-z$
```

The first thing to consider for the above method is how to calculate $\ln(2)$. It is not possible to directly calculate $\ln(2)$ using the above algorithm as $2=\frac{1}{2}\cdot 2^2$, however $\frac{1}{2}=\frac{1}{2}\cdot 2^0$ and so we do not need to know $\ln(2)$ to calculate $\ln(\frac{1}{2})$. We can see that $\ln(2)=-\ln(\frac{1}{2})$, and so we can calculate our constant value $\ln(2)$ to be used in the algorithm by using the algorithm itself.

Now similar to previous Taylor approximations the final error of our approximation using the above method is $\epsilon_n := |\ln(x)-\text{taylor\_log}(x,n)|$. As the next term of the approximation would be $\frac{(1-x)^n}{n}$, then we know that $\epsilon_n\leq\left|\frac{(1-a)^n}{n}\right|$; further we know that $a\in[\frac{1}{2},1)$ and thus $\epsilon_n<\frac{1}{2^n n}$.

Using this approximation we can see that if we wish to guarantee d decimal places of accuracy then it suffices to find $n\in\mathbb{N}$ such that $\frac{1}{2^n n}<10^{-d}\implies 2^n n>10^d$. As $n\in\mathbb{N}$ then $2^n<2^n n$ and so we merely need to find $n\in\mathbb{N}$ such that $2^n>10^d$ to guarantee d decimal places of accuracy. Some example values are included in the table below:

$d\in\mathbb{N}$	$\arg\min\left\{n\in\mathbb{N}:2^n>10^d\right\}$
1	4
10	34
100	333
1000	3322

As we now have Taylor methods for approximating both e^x and $\ln(x)$, then we can use the two to derive a Taylor method of calculating x^y and $\log_x(y)$. To start we will consider $x^y = e^{y\ln(x)}$ and $x = a \cdot 2^b$, giving the solution as $x^y = e^{y(b\ln(2)+\ln(a))}$. Similarly we not that $\log_x(y) = \frac{\ln(y)}{\ln(x)}$, and if we consider that $x = a \cdot 2^b$ and $y = c \cdot 2^d$, then we see that $\log_x(y) = \frac{d\ln(2)+\ln(c)}{b\ln(2)+\ln(a)}$. Below are the Taylor methods for approximating these functions:

Algorithm 5.4.3: Taylor Method for calculating x^y and $\log_x(y)$

```
taylor_log($x \in \mathbb{R}^+, y \in \mathbb{R}^+, n \in \mathbb{N}$):
    $a :=$ taylor_nat_log($y, n$)
    $b :=$ taylor_nat_log($x, n$)
    return $\frac{a}{b}$

taylor_pow($x \in \mathbb{R}^+, y \in \mathbb{R}, n \in \mathbb{N}$):
    $a :=$ taylor_nat_log($x, n$)
    $a \mapsto y \cdot a$
    return taylor_exp($a, n$)
```

To test the convergence of the Taylor methods above we are going to test calculations of $7.3^{4.8}$, $7.3^{-4.8}$, $0.21^{4.8}$, $7.3^{0.21}$, $\log_{7.3}(4.8)$, $\log_{0.21}(4.8)$ and $\log_{7.3}(0.21)$. These values are calculated for several different values of n with the bold digits representing the correct values in the tables below:

n	$7.3^{4.8}$	$7.3^{-4.8}$	$0.21^{4.8}$	$7.3^{0.21}$
1	**1.**0000000000	**1.0000**000000	**1.00**00000000	**1.**0000000000
2	**10.**561319400	**-**8.561319400	**-**6.422212933	**1.**4183077237
3	**5**6.076838311	**36.**990949511	**2**1.518877680	**1.5**046585363
4	**2**00.85920964	**-1**07.8118783	**-**48.47602784	**1.51**67171202
5	**54**6.24576990	**2**37.58122696	**8**2.710783892	**1.51**79778747
6	**1**205.3726532	**-**421.5471761	**-**113.8668463	**1.5180**831956
7	**2**253.5829747	**6**26.66342673	**1**31.57101558	**1.518090**5223
8	**3**682.4131809	**-8**02.1668232	**-1**31.0877429	**1.5180909**589
9	**53**86.6141612	**90**2.03416303	**1**14.86315726	**1.51809098**16
10	**7**193.4074522	**-9**04.7591286	**-**89.85299062	**1.5180909827**
...	...	...	...	...
20	**1390**1.238666	**-11.**00988984	**-0.**092958315	**1.5180909827**
...	...	...	...	...
40	**13929.955484**	**0.00007178**62	**0.0005580236**	**1.5180909827**
...	...	...	...	...
80	**13929.955484**	**0.0000717877**	**0.0005580236**	**1.5180909827**

As we can see in the table the taylor_pow does not converge perfectly, and may even diverge from the correct value for small values of n; however we see that the methods do converge for large values of n. This behaviour is due to the values being outside the restrictions used in the analysis of the functions.

n	$\log_{7.3}(4.8)$	$\log_{0.21}(4.8)$	$\log_{7.3}(0.21)$
1	**0.**8431178860	**-1.**086107266	**-0.**776274970
2	**0.**8431178860	**-1.**086107266	**-0.**776274970
3	**0.**8045021618	**-1.**025878600	**-0.7**84207957
4	**0.7**938608884	**-1.**011309817	**-0.7**84982875
5	**0.7**906472231	**-1.0**07102721	**-0.785**071082
6	**0.789**6173849	**-1.00**5776909	**-0.7850**82036
7	**0.789**2739993	**-1.00**5337682	**-0.78508**3473
8	**0.789**1562591	**-1.005**187460	**-0.785083**668
9	**0.789**1150494	**-1.005**134935	**-0.7850836**95
10	**0.789**1003970	**-1.005**116266	**-0.78508369**9
...	...	...	...
50	**0.7890920869**	**-1.005105681**	**-0.785083699**

This shows that $\mathrm{taylor\_log}$ converges better than $\mathrm{taylor\_exp}$, however part of this is due to the values tested having magnitudes close to 1. Answers with a larger or smaller magnitudes tend to converge slower, which can be seen in the table for $\mathrm{taylor\_exp}$ where the value that had best convergence had an answer of about 1.5 and all other values tested had answers that were several orders of magnitude different from 1.

5.5 Hyperbolic Series Method

There are more efficient series which can be used to find $\ln$, which converge quicker than the Taylor approximation; one such method is to consider the Hyperbolic Trigonometric function $\tanh$. We start by considering the definition that $\tanh(x) :=^{x} -e^{-x}e^{x} + e^{-x}$, and then find a formula for $\tanh^{-1}(x)$:

$$\begin{aligned}
z = \frac{e^x - e^{-x}}{e^x + e^{-x}} &\implies z = \frac{e^{2x} - 1}{e^{2x} + 1} \\
&\implies ze^{2x} + z = e^{2x} - 1 \\
&\implies e^{2x}(1 - z) = 1 + z \\
&\implies e^{2x} = \frac{1+z}{1-z} \\
&\implies e^{x} = \left(\frac{1+z}{1-z}\right)^{\frac{1}{2}} \\
&\implies x = \frac{1}{2}\ln\left(\frac{1+z}{1-z}\right)
\end{aligned}$$

Using this we can see that $2\tanh^{-1}\left(\frac{x-1}{x+1}\right) = \ln(x)$, and we can use the Taylor Expansion of $\tanh^{-1}$ to approximate $\ln$.

Now to attain the Taylor series for $\tanh^{-1}(x)$ we can use the same method as when we calculated the Taylor series for $\ln$. The exact calculations are omitted, but the end result is that we get that:

$$\tanh^{-1}(x) = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{2n+1} \forall x \in \mathbb{R}^+$$

And thus by using this series we get the result that:

$$\ln(x) = 2\sum_{n=0}^{\infty} \frac{1}{2n+1}\left(\frac{x-1}{x+1}\right)^{2n+1} \quad \forall x \in \mathbb{R}^+$$

The implementation of this is similar to previous implementations of series approximations of a function and is detailed below:

Algorithm 5.5.1: Hyperbolic seies method for $\ln$

```
hyperbolic_nat_log($x \in \mathbb{R}^+, n \in \mathbb{Z}_0^+$):
    $a := \frac{x-1}{x+1}$
    $b := y^2$
    $c := a$
    $k := 0$
    while $k \leq n$:
        $a \mapsto a \cdot b$
        $c \mapsto c + \frac{a}{2k+1}$
        $k \mapsto k + 1$
    return $2 \cdot c$
```

Using this we see that if we have $\epsilon_n := |\ln(x) - \text{hyperbolic\_nat\_log}(x, n)|$, then we know that $\epsilon_n \leq \frac{1}{2n+3}\left|\frac{x-1}{x+1}\right|^{2n+3}$. If we consider restricting our calculations to $x \in [\frac{1}{2}, 1)$ by using the same calculations as shown for Algorithm **??**, then we can see that $|x-1| \leq \frac{1}{2}$ and $|x+1| \geq \frac{3}{2}$; therefore $\epsilon_n \leq \frac{1}{3^{2n+3}(2n+3)}$.

By considering the final simplification that $\epsilon_n < \frac{1}{3^{2n+3}}$, then if we wish to have $\epsilon_n < \tau \in \mathbb{R}^+$ it suffices to find $n \in \mathbb{N}$ such that $\frac{1}{3^{2n+3}} < \tau$. In particular we consider when $\tau = 10^{-d}$ which will guarantee d decimal places of accuracy, below is a table showing the smallest $n \in \mathbb{N}$ that guarantees d decimal places of accuracy:

$d \in \mathbb{N}$	$\arg\min\left\{n \in \mathbb{N} : 3^{2n+3} > 10^d\right\}$
1	1
10	8
100	104
1000	1047

As can be seen in the table, fewer iterations are needed to approximate $\ln(x)$ to the same degree of accuracy using hyperbolic series as when using the Taylor series. Further the calculations performed each iteration are very similar in complexity, both being $\mathcal{O}(1)$, and thus we can expect that Algorithm 5.5.1 will execute faster than 5.4.2.

5.6 Continued fractions

Another method for evaluating e^x is the use of continued fractions, which are a way of approximating real functions by a rational number with a recursive structure. Such fractions have been studied for many years and can be used to rationally approximate functions. Some examples of continued fractions for real numbers are:

$$e = 2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{4 + \ddots}}}}} \qquad \pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{1 + \ddots}}}}}$$

In general a continued fraction for a number $x \in \mathbb{R}$ has the form:

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \ddots}}} \tag{5.6.1}$$

As writing of continued fractions in the above manner takes up a lot of room and has a degree of ambiguity we will use the following notation:

$$\mathbf{K}_{n=1}^{\infty}\frac{a_n}{b_n} := \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cfrac{a_4}{b_4 + \ddots}}}} \tag{5.6.2}$$

Therefore we can re-write Equation 5.6.1 as $b_0 + \mathbf{K}_{n=1}^{\infty}\frac{a_n}{b_n}$.

One of the most useful formulas regarding continued fractions was formulated by Leonhard Euler, and deals with the sum $a_0 + a_0a_1 + a_0a_1a_2 + \cdots + (a_0 \cdots a_n) = \sum_{i=0}^{n}(\prod_{j=0}^{i} a_j)$. The formula derived by Euler is known as Euler's Continued Fraction Formula and is as follows:

$$\sum_{i=0}^{n}\left(\prod_{j=0}^{i} a_j\right) = \mathbf{K}_{i=0}^{n}\frac{\alpha_i}{\beta_i} \text{ where } \alpha_i := \begin{cases} a_0 & : \ i = 0 \\ -a_i & : \ i \in [1,n] \cap \mathbb{Z} \end{cases}$$
$$\beta_i := \begin{cases} 1 & : \ i = 0 \\ 1 + a_i & : \ i \in [1,n] \cap \mathbb{Z} \end{cases} \tag{5.6.3}$$

Many Taylor series have a structure that is compatible with equation 5.6.3 and so can be approximated by a continued fraction in this way. In particular we are looking at $e^x = \sum_{k=0}^{n} \frac{x^n}{n!}$ where we note that we can re-write the series as $e^x = 1 + \sum_{i=1}^{n}(\prod_{j=1}^{i} \frac{x}{j})$ and therefore by using Euler's Continued Fraction Formula we see that:

$$e^x = 1 + \cfrac{x}{1 - \cfrac{\frac{1}{2}x}{1 + \frac{1}{2}x - \cfrac{\frac{1}{3}x}{\ddots - \cfrac{\frac{1}{n-1}x}{1 + \frac{1}{n-1}x - \frac{\frac{1}{n}x}{1+\frac{1}{n}x}}}}} \tag{5.6.4}$$

$$= 1 + \mathbf{K}_{i=1}^{n} \frac{\alpha_i}{\beta_i} \qquad \text{where } \alpha_i := \begin{cases} x & : \ i = 1 \\ -\frac{1}{i}x & : \ i \in [2,n] \cap \mathbb{Z} \end{cases}$$

$$\beta_i := \begin{cases} 1 & : \ i = 1 \\ 1 + \frac{1}{i}x & : \ i \in [2,n] \cap \mathbb{Z} \end{cases}$$

We would like to simplify the above equation to remove the fractional coefficients. If we consider multiplying α_1 by some constant c_1, then to have an equivalent fraction we would have to multiply it's denominator by c_1; in practice this means multiplying β_1 and α_2 by c_1. We can suppose that we could continue in a similar manner for constants $c_2, c_3, \ldots$ multiplying $\alpha_2, \alpha_3, \ldots$.

Proposition 5.6.1. *If we have a continued fraction $b_0 + \mathbf{K}_{i=1}^{n} \frac{a_i}{b_i}$ and constants $(c_i : i \in [1,n] \cap \mathbb{Z})$, then:*

$$b_0 + \mathbf{K}_{i=1}^{n} \frac{a_i}{b_i} = b_0 + \mathbf{K}_{i=1}^{n} \frac{c_{i-1}c_i a_i}{c_i b_i}$$

where $c_0 := 1$, for any $n \in \mathbb{N}$.

Proof. We will proceed by induction on $n \in \mathbb{N}$.

$\mathbf{H}(n)$**:** $b_0 + \mathbf{K}_{i=1}^{n} \frac{a_i}{b_i} = b_0 + \mathbf{K}_{i=1}^{n} \frac{c_{i-1}c_i a_i}{c_i b_i}$

$\mathbf{H}(1)$**:**

$$\begin{aligned} b_0 + \frac{c_0 c_1 a_1}{c_1 b_1} &= b_0 + \frac{c_1 a_1}{c_1 b_1} && \text{as } c_0 = 1 \\ &= b_0 + \frac{a_1}{b_1} && \text{as required} \end{aligned}$$

$\mathbf{H}(n) \implies \mathbf{H}(n+1)$**:**

$$\begin{aligned} b_0 + \mathbf{K}_{i=1}^{n+1} \frac{c_{i-1}c_i a_i}{c_i b_i} &= b_0 + \left(\mathbf{K}_{i=1}^{n} \frac{c_{i-1}c_i a_i}{c_i b_i} \right)_{+} \frac{c_n c_{n+1} a_{n+1}}{c_{n+1} b_{n+1}} \\ &= b_0 + \left(\mathbf{K}_{i=1}^{n} \frac{c_{i-1}c_i a_i}{c_i b_i} \right)_{+} c_n \frac{a_{n+1}}{b_{n+1}} \end{aligned}$$

Now let us define b_i' as:

$$\begin{cases} b_n + \frac{a_{n+1}}{b_{n+1}} & : \ i = n \\ b_i & : \ i \neq n \end{cases}$$

Therefore we can continue and see that:

$$\begin{aligned}
b_0 + \mathbf{K}_{i=1}^{n+1}\frac{c_{i-1}c_i a_i}{c_i b_i} &= b_0 + \mathbf{K}_{i=1}^{n}\frac{c_{i-1}c_i a_i}{c_i b_i'} \\
&= b_0 + \mathbf{K}_{i=1}^{n}\frac{a_i}{b_i'} && \text{by H}(n) \\
&= b_0 + \mathbf{K}_{i=1}^{n+1}\frac{a_i}{b_i}
\end{aligned}$$

□

Using this proposition we can see that if we have the sequence $(c_1, c_2, \ldots, c_n)$ defined as $c_i = i$ and apply it to our sequence for e^x we get that:

$$e^x = 1 + \cfrac{x}{1 - \cfrac{x}{2 + x - \cfrac{2x}{\ddots - \cfrac{(n-1)x}{n - 1 + x - \frac{x}{n+x}}}}} \tag{5.6.5}$$

$$= 1 + \mathbf{K}_{i=1}^{n}\frac{\alpha_i}{\beta_i} \qquad \text{where } \alpha_i := \begin{cases} x & : \ i = 1 \\ -(i-1)x & : \ i \in [2, n] \cap \mathbb{Z} \end{cases}$$

$$\beta_i := \begin{cases} 1 & : \ i = 1 \\ x + i & : \ i \in [2, n] \cap \mathbb{Z} \end{cases}$$

This is a much simpler continued fraction, but evaluating it would still be expensive due to the repeated division operations; to get around this we can consider what are known as the convergents of a continued fraction. It is obvious that if we use a continued fraction to approximate some value z by the continued fraction $b_0 + \mathbf{K}_{i=1}^{n}\frac{a_i}{b_i}$, then there are some $A_n, B_n \in \mathbb{N}$ such that $z = \frac{A_n}{B_n}$.

To start we will define $A_{-1} := 1$ and $B_{-1} := 0$, and consider the case when $n = 0$; for this case $z = b_0$, which means that $A_0 = b_0$ and $B_0 = 1$. For the case when $n = 1$ we have $z = b_0 + \frac{a_1}{b_1}$, which when rearranged is $z = \frac{b_0 b_1 + a_1}{b_1}$. This means that $A_1 = b_0 b_1 + a_1 = b_1 A_0 + a_1 A_{-1}$ and $B_1 = b_1 = b_1 B_0 + a_1 B_{-1}$, and for the case when $n = 2$ we get the similar result that $A_2 = b_0 b_1 b_2 + a_2 b_0 + a_1 b_2 = b_2 A_1 + a_2 A_0$ and $B_2 = b_1 b_2 + a_2 = b_2 B_1 + a_2 B_0$.

It is actually true that this relationship continues for all $n \in \mathbb{N}$, and thus we get what are known as the Fundamental Recurrence Formulas for continued fractions:

$$\begin{aligned}
A_{-1} &= 1 & B_{-1} &= 0 \\
A_0 &= b_0 & B_0 &= 1 \\
A_{n+1} &= b_{n+1}A_n + a_{n+1}A_{n-1} & B_{n+1} &= b_{n+1}B_n + a_{n+1}B_{n-1} \quad \forall n \in \mathbb{Z}_0^+
\end{aligned}$$

Using this and our simplified continued fraction for e^x we can use the following method to approximate e^x by using a continued fraction up to a_n, b_n where $n \geq 2$:

Algorithm 5.6.1: Continued fraction for e^x

```
cont_frac_exp($x \in \mathbb{R}, n \in \mathbb{N}$):
    $A_1 := x + 1$
    $B_1 := 1$
    $A_2 := x^2 + 2x + 2$
    $B_2 := 2$
    $a := -x$
    $b := 2 + x$
    $k := 2$
    while $k < n$:
        $a \mapsto a - x$
        $b \mapsto b + 1$
        $A_{k+1} := bA_k + aA_{k-1}$
        $B_{k+1} := bB_k + ab_{k-1}$
        $k \mapsto k + 1$
    return $\frac{A_k}{B_K}$
```

One observation of the above algorithm, when implemented on a computer, is that if we pre-generate b_i and a_i for $i \in [2, n] \cap \mathbb{Z}$ then the calculations of A_i and B_i are independent. This means that, if supported by the computer, both A_i and B_i could be computed in parallel. This may allow an implementation of the algorithm to be more efficient than one that computes the function in sequence.

Now while continued fractions are useful for approximating functions, it is difficult to evaluate the error of their output analytically. One result is that if $a_n = 1 \forall n \in \mathbb{N}$ when approximating some value z, then $|z - \frac{A_n}{B_n}| \leq \frac{1}{|B_{n+1}B_n|}$. If we transform the continued fraction of e^x into this form by using Proposition **??**, then we get that:

$$e^x = 1 + \cfrac{1}{(-\frac{1}{x}) + \cfrac{1}{(-\frac{2+x}{2x}) + \cfrac{1}{(-\frac{3+x}{3x}) + \ddots}}}$$

By using a computer to implement the calculations for a test value of $x = 1$, we see that $\frac{1}{B_5 B_6} = 0.009131261889664\ldots$ and $\frac{1}{B_{10}B_{11}} = 0.000041307209877\ldots$; thus we can guarantee two decimal place of accuracy with $\mathrm{cont\_frac\_exp}(1, 5)$ and 4 with $\mathrm{cont\_frac\_exp}(1, 10)$. However if we instead have $x = 14$ then $\frac{1}{B_{10}B_{11}} = 0.314711263190806\ldots$ and convergence is similarly poor for negative values.

Further computations show that convergence of $x \in (0, 1)$ is better than the convergence when $x = 1$, and thus we can use the identities and conversions to ensure good convergence. In particular if $x \in$ then we can calculate the reciprocal of $\mathrm{cont\_frac\_exp}(-x, n)$ and if $x \in (1, \infty)$ we use the identity that $x = a \cdot 2^b$; with this we see that $e^x = (e^a)^{2^b}$ and $2^b \in \mathbb{Z}^+$.

As $a \in (0, 1)$ and $2^b \in \mathbb{Z}^+$ then we can calculate $z = 2^a$ algorithm **??**. Then we can calculate z^{2^b} using algorithm **??**, to find our approximation of e^x. Performing the calculation in this way allows us to use the our continued fraction method to guarantee fast convergence and the $\mathcal{O}(1)$ integer exponential algorithm to guarantee the correct approximation without increasing the algorithmic complexity of the calculations by more than a constant factor.

With this restriction in place we know that algorithm 5.6.1 converges at least as quickly as it does for $x = 1$, and thus we can use it's convergence to guarantee the convergence of our method. Below is a table that shows the minimum n needed to achieve the associated d decimal places of accuracy:

d	Minimum n to guarantee d decimal places of accuracy
1	2
10	22
100	235
1000	2386

An alternative continued fraction for e^x that arises from the generalized hyper geometric series is:

$$e^x = \cfrac{1}{1 - \cfrac{x}{1 + \cfrac{x}{2 - \cfrac{x}{3 + \cfrac{2x}{4 - \cfrac{2x}{5 + \cfrac{3x}{6 - \ddots}}}}}}} \tag{5.6.6}$$

$$= \mathbf{K}_{i=1}^{n} \frac{\alpha_i}{\beta_i} \qquad \text{where } \alpha_i := \begin{cases} 1 & : \ i = 1 \\ -x & : \ i = 2 \\ (-1)^{i-1}\lfloor\frac{i-1}{2}\rfloor x & : \ i \in [3,\infty) \cap \mathbb{Z} \end{cases}$$

$$\beta_i := \begin{cases} 1 & : \ i = 1 \\ i-1 & : \ i \in [2,\infty) \cap \mathbb{Z} \end{cases}$$

Due to the $(-1)^{i-1}\lfloor\frac{i-1}{2}\rfloor$ factor in the definition of α_i it is more efficient to perform two updates each step rather than one. Below is the implementation of this method:

Algorithm 5.6.2: Continued fraction for e^x version 2

```
cont_frac_exp_v2($x \in \mathbb{R}, n \in \mathbb{N}$):
    $A_1 := 1$
    $B_1 := 1$
    $A_2 := 1$
    $B_2 := 1 - x$
    $a := 1$
    $b := 1$
    $k := 2$
    while $k < n$:
        $a \mapsto xa$
        $b \mapsto b + 1$
        $A_{k+1} := bA_k + aA_{k-1}$
        $B_{k+1} := bB_k + ab_{k-1}$
        $k \mapsto k + 1$
        $b \mapsto b + 1$
```

```
                $A_{k+1} := bA_k - aA_{k-1}$
                $B_{k+1} := bB_k - ab_{k-1}$
                $k \mapsto k+1$
        return $\frac{A_k}{B_K}$
```

The fraction needed to analyse this method is again found by using proposition **??** and is:

$$\cfrac{1}{1+\cfrac{1}{-\frac{1}{x}+\cfrac{1}{-2+\cfrac{1}{\frac{3}{x}+\cfrac{1}{2+\cfrac{1}{-\frac{5}{x}+\cfrac{1}{-2+\ddots}}}}}}}$$

By implementing this with a computer we get similar results to above, particularly that there is rapid convergence for $x \in (0,1)$. Further the convergence of values in $x \in (0,1)$ is more rapid than $x = 1$ and so we can use the convergence of $x = 1$ as an upper bound of our method. Below is the table showing the smallest $n \in \mathbb{N}$ needed to ensure d decimal places of accuracy for some $d \in \mathbb{N}$:

d	Minimum n to guarantee d decimal places of accuracy
1	4
10	12
100	61
1000	405

As can be seen the convergence of 5.6.6 appears to be significantly faster than that of 5.6.5 and one might be satisfied by this, however an even better solution exists.

As the fraction 5.6.6 can be shown to converge for all values of x to e^x then we can consider the even and odd convergents. The even convergents of the sequence are $\frac{A_0}{B_0}, \frac{A_2}{B_2}, \frac{A_4}{B_4}, \ldots$, while the odd convergents are $\frac{A_1}{B_1}, \frac{A_3}{B_3}, \frac{A_5}{B_5}, \ldots$. As $\lim_{n\to\infty} \frac{A_n}{B_n} = e^x$, then $\lim_{n\to\infty} \frac{A_{2n}}{B_{2n}} = \lim_{n\to\infty} \frac{A_{2n+1}}{B_{2n+1}} = e^x$; the following proposition gives an explicit form for the odd and even convergents.

Proposition 5.6.2. *If $z = \mathbf{K}_{i=1}^{\infty} \frac{a_i}{1}$, then the limit of the odd convergent of z is:*

$$x_{odd} = a_1 - \cfrac{a_1a_2}{1+a_2+a_3-\cfrac{a_3a_4}{1+a_4+a_5-\cfrac{a_5a_6}{1+a_6+a_7-\ddots}}}$$

while the limit of the even convergent is:

$$x_{even} = \cfrac{a_1}{1+a_2-\cfrac{a_2a_3}{1+a_3+a_4-\cfrac{a_4a_5}{1+a_5+a_6-\ddots}}}$$

Proof. Omitted □

If we apply proposition **??** to 5.6.6, to achieve the form $\mathbf{K}_{i=1}^{\infty}\frac{a_i}{1}$ then we end up with the following fraction:

$$e^x = \cfrac{1}{1+\cfrac{-x}{1+\cfrac{\frac{1}{2}x}{1+\cfrac{-\frac{1}{6}x}{1+\cfrac{\frac{1}{6}x}{1+\cfrac{-\frac{1}{10}x}{1+\cfrac{\frac{1}{10}x}{1+\cfrac{-\frac{1}{14}x}{1+\ddots}}}}}}}} \tag{5.6.7}$$

Now if we apply proposition 5.6.2 to the above fraction we see that:

$$e^x = 1 + \cfrac{x}{1-x+\frac{1}{2}x+\cfrac{\frac{1}{12}x^2}{1-\frac{1}{6}x+\frac{1}{6}x+\cfrac{\frac{1}{60}x^2}{1-\frac{1}{10}x+\frac{1}{10}x+\cfrac{\frac{1}{140}x^2}{\ddots}}}} \tag{5.6.8}$$

Finally by simplifying and applying proposition **??** one more time we reach the following continued fraction for e^x:

$$\begin{aligned} e^x &= 1 + \cfrac{2x}{2-x+\cfrac{x^2}{6+\cfrac{x^2}{10+\cfrac{x^2}{14+\ddots}}}} \\ &= 1 + \mathbf{K}_{i=1}^{\infty}\frac{\alpha_i}{\beta_i} \qquad \text{where } \alpha_i := \begin{cases} 2x & : \ i = 1 \\ x^2 & : \ i \in [2,\infty)\cap\mathbb{Z} \end{cases} \\ &\qquad\qquad\qquad\quad \beta_i := \begin{cases} 2-x & : \ i = 1 \\ 4i-2 & : \ i \in [2,\infty)\cap\mathbb{Z} \end{cases} \end{aligned} \tag{5.6.9}$$

If we implement this method by using the Fundamental Recurrence Formula then we get the following:

Algorithm 5.6.3: Continued fraction for e^x version 3

```
cont_frac_exp_v3($x \in \mathbb{R}, n \in \mathbb{N}$):
    $A_0 := 1$
    $B_0 := 1$
    $A_1 := 2 + x$
    $B_1 := 2 - x$
    $a := x^2$
```

```
        $b := 2$
        $k := 1$
        while $1 < n$:
            $b \mapsto b + 4$
            $A_{k+1} := bA_k + aA_{k-1}$
            $B_{k+1} := bB_k + ab_{k-1}$
            $k \mapsto k + 1$
        return $\frac{A_k}{B_K}$
```

As with the previous two continued fraction methods of approximating e^x we can apply proposition **??** to 5.6.9 to find the following equivalent continued fraction:

$$1 + \cfrac{1}{\frac{1}{x} - \frac{1}{2} + \cfrac{1}{\frac{12}{x} + \cfrac{1}{\frac{5}{x} + \cfrac{1}{\frac{28}{x} + \ddots}}}}$$

Again a computer was used to evaluate B_k of the above fraction, which gave the expected results of quick convergence for $x \in (0, 1)$ and more rapid convergence for $x \in (0, 1)$ than $x = 1$. Using this the table below was generated to show the minimum $n \in \mathbb{N}$ that guarantees d digits of accuracy:

d	Minimum n to guarantee d decimal places of accuracy
1	2
10	6
100	30
1000	202

This has the fastest theoretical convergence of the three methods, and thus is expected to perform the best.

5.7 Comparison of Methods

We have introduced several methods for calculating both logarithms and exponentials in this chapter, and considered their theoretical convergence; we now look at a direct comparison of the methods as implemented in C.

The first consideration is which values to use while comparing methods. While all the methods converge for all values, or can be made to by using transformations of the inputs and outputs, most methods converge best for small values. Therefore values being tested will typically be in the range of $[0.5, 1)$.

The first methods to be compared here are the versions of the continued fraction method discussed previously. Below we have the outputs of different versions of the program for different values of n, with the bold digits being the correctly approximated digits.

n	`cont_frac_exp_v1`	`cont_frac_exp_v2`	`cont_frac_exp_v3`
1	**1.**9449999999999	3.3333333333333	**2.0**769230769230
2	**2.0**021666666666	3.3333333333333	**2.013**2689987937
3	**2.01**21708333333	**2.0**769230769230	**2.01375**43842848
4	**2.013**5714166666	**2.0**054200542005	**2.01375270**42253
5	**2.0137**348180555	**2.013**2689987937	**2.01375270747**44
6	**2.01375**11581944	**2.0137**906192914	**2.0137527074704**
7	**2.013752**5879565	**2.01375**43842848	**2.0137527074704**
8	**2.013752**6991603	**2.013752**6161232	**2.0137527074704**
9	**2.01375270**69445	**2.01375270**42253	**2.0137527074704**
10	**2.0137527074**399	**2.013752707**6056	**2.0137527074704**

As can be seen here the first two methods have similar convergence, however despite having a very poor theoretical convergence the first method converges better than the second version. However it is obvious that the third method has the fastest convergence, and thus should be the one to use in further comparisons.

Now we can compare the speed of the Taylor and continued fraction methods of calculating exponential values. For this we will use 1000 values in the range $[\frac{1}{2}, 1)$ and calculate each 100000 times to compare the speed of the method. We will be using values of n which guarantee 10 decimal places of accuracy, in particular $n = 14$ for `taylor_exp` and $n = 6$ for `cont_frac_exp_v3`.

The results of the tests run on my computer are included in the table below alongside those for the built in `exp` function in `math.h`:

	Total time:	Average time:	Minimum time:	Maximum time:
`taylor_exp`	12.430s	0.012s	0.012s	0.019s
`cont_frac_exp`	4.741s	0.004s	0.004s	0.007s
`builtin_exp`	2.608s	0.002s	0.002s	0.004s

This shows that the continued fractions method of evaluating exponential functions is almost three times as efficient as the standard Taylor series method. However both fall short of the built in method, despite the hyperbolic series method being a close second. This is likely due to a lower level implementation of the exponential function with various highly efficient programming practices implemented to optimize the code execution speed.

However one consideration is that if we instead test values in the range $[-5, 50]$, then while both `taylor_exp` and `cont_frac_exp` have similar results the total time for `cont_frac_exp` becomes 9.347s. This discrepancy is due to the additional calculations needed by `cont_frac_exp` so that it evaluates only values in the range $[\frac{1}{2}, 1)$ for a quicker convergence.

The two methods discussed to evaluate $\ln$ have their convergence for different values of n shown below, where they are approximating the value 0.7, with the bold digits representing the correctly approximated digits:

n	`taylor_nat_log`	`hyperbolic_nat_log`
1	**-0.3**00000000000	**-0.3566**04925707
2	**-0.3**00000000000	**-0.35667**3383305
3	**-0.3**45000000000	**-0.3566749**06089
4	**-0.35**4000000000	**-0.35667494**2973
5	**-0.356**025000000	**-0.3566749439**13
6	**-0.356**511000000	**-0.356674943938**
7	**-0.356**632500000	**-0.356674943938**
8	**-0.3566**63742857	**-0.356674943938**
9	**-0.35667**1944107	**-0.356674943938**
10	**-0.356674**131107	**-0.356674943938**

We can see here that the hyperbolic method converges a lot faster than the Taylor method; one particular note is that the hyperbolic series accurately approximates the first 12 decimal places of $\ln(0.7)$ accurately in just 6 iterations while the Taylor method only achieves 6 decimal places after 10 iterations.

To further test the two methods the table below shows the timings of calculating 1000 values in the range $[0.02, 50]$, each of which will be calculated to 10 decimal places 100000 times by each method. To guarantee 10 decimal places of accuracy with `taylor_log` we can use $n = 34$ and $n = 8$ for `hyperbolic_log`, below is the table that displays the results alongside the results for the built in `log` function:

	Total time:	Average time:	Minimum time:	Maximum time:
`taylor_log`	22.247s	0.022s	0.021s	0.026s
`hyperbolic_log`	7.742s	0.007s	0.007s	0.009s
`builtin_log`	3.438s	0.003s	0.003s	0.005s

Here we can see that the hyperbolic method of approximating $\ln(x)$ is the better of the two methods discussed, being around three times faster in execution. While the built in function is, as to be expected, the fastest executing function, `hyperbolic_log` is not far behind, implying that `builtin_log` may use an optimized version of `hyperbolic_log`.

Finally we get to comparing the general exponential, x^y, and logarithm, $log_x(y)$, functions. First we will test the convergence of the two variations of the $log_x(y)$ function for different values of n, using $(x, y) = (1.5, 15)$:

n	`taylor_log`	`hyperbolic_log`
1	**6.**1155499597569	**6.678**4758082659
2	**6.**1155499597569	**6.6788**677803210
3	**6.**5747854684469	**6.678873**4950163
4	**6.6**587865280763	**6.67887358**57263
5	**6.67**48050386470	**6.6788735872**409
6	**6.678**0194099644	**6.678873587267**1
7	**6.678**6895339230	**6.6788735872675**
8	**6.6788**331533503	**6.6788735872675**
9	**6.6788**645711387	**6.6788735872675**
10	**6.67887**15529216	**6.6788735872675**

This table clearly demonstrates that `hyperbolic_log` converges faster to the correct value than `taylor_log` as expected. The table below shows the convergence of `taylor_pow` and `improved_pow` for the input of $(x, y) = (1.115, 15)$:

n	`taylor_pow`	`improved_pow`
1	**1.**0000000000000	**5.**7430173458025
2	**4.**7597077083991	**5.11**63939774264
3	**5.**9158698156503	**5.118**5134154921
4	**5.**6528248111124	**5.1182**823832710
5	**5.**3825631874287	**5.11826**88605223
6	**5.**2383576844918	**5.118267**9322812
7	**5.1**703487304639	**5.1182678**673534
8	**5.1**401274582517	**5.11826786**27291
9	**5.1**272612067887	**5.118267862**3951
10	**5.1**219353833296	**5.118267862**3708
...	...	...
20	**5.11826**84126550	**5.1182678623688**
...	...	...
30	**5.118267862**4756	**5.1182678623688**
...	...	...
40	**5.118267862368**9	**5.1182678623688**

Both of these methods for the general exponential function have slow convergences, though the improved method does converge faster. This implies that there may be a more efficient method for approximating x^y.

Next we will consider the actual speed of both the logarithm and exponential functions presented. One note is that C does not have a general $\log$ function for an arbitrary base in `math.h`, and so to implement this we will use `log(y)/log(x)` for the built in logarithm. All of the functions will have values of $(x, y) \in (0, 2] \times [0, 3)$ and will use a value of n sufficient to calculate their answer accurate to 10 decimal places. Below are the calculations for 1000 random values calculated 10000 times for each method:

	Total time:	Average time:	Minimum time:	Maximum time:
`taylor_log`	4.750s	0.004s	0.004s	0.008s
`hyperbolic_log`	1.589s	0.001s	0.001s	0.002s
`builtin_log`	0.690s	0.000s	0.000s	0.000s
`taylor_pow`	6.956s	0.006s	0.006s	0.007s
`improved_pow`	2.456s	0.002s	0.002s	0.003s
`pow`	0.787s	0.000s	0.000s	0.001s

Again we see that the methods that we showed to be theoretically superior, do in fact have superior execution speeds; however our methods still fail to match the execution speed of those built into C.

Overall we can conclude that if one were to want to implement calculating logarithms of a number then the hyperbolic series method is the best choice discussed, while the best choice for evaluating exponentials is the continued fraction method.

The special case of the exponentiation by squaring is worth considering in the case where a computer only supports integers. This is because the algorithm will still work for integer only values, while most of the others will not, and has a computational complexity of $\mathcal{O}(1)$.

6 Conclusion

In this document we set out to consider different methods of calculating common functions that one may find on a calculator, as such we succeeded and now have a deeper understanding of these functions. We have also gained an insight into how many calculators or computers may operate in calculating these functions.

In studying the root functions we have seen that while there are various methods available the most efficient method is the inverse newton square root method. This method converges quadratically to the required root and has a faster computation time than the standard Newton method due to the lack of division operations. We saw that both of these methods outstripped the linear convergence of the bisection method, which while simple and efficient in the computational complexity sense, takes many more steps to achieve comparable accuracy and so is less efficient in computational time.

The digit by digit method of calculating square roots is interesting but ultimately of little practical value for modern computers due to it's poor efficiency, though it has very interesting accuracy properties. It's integer square root counterpart on the other hand is particularly interesting due to its $\mathcal{O}(1)$ computational complexity and reliance on simple integer operations, and has possible practical applications if square roots are only needed accurate to only their integer part.

The root functions were successfully implemented in C and when implemented with MPFR were able to give answers accurate to arbitrary precision. In particular we were able to accurately compute $\sqrt{2}$ accurate to 1000000 decimal places in a reasonably short span of time.

The trigonometric functions are an interesting topic to study and in doing so we found several very different methods for approximating their values. The geometric method studied is conceptually simple, but turned out to be complex to analyse, the end result giving a method that had a low computational complexity per iteration but required many iterations to achieve accuracy comparable to other methods.

The Taylor method for trigonometric functions turned out to be the most efficient method, once the range of inputs was restricted. Further this method was easy to analyse the accuracy of due to the nature of the Taylor series, making it simple to guarantee a given degree of accuracy.

The CORDIC algorithm was the least efficient of the methods analysed, but as mentioned earlier, it still has it's place. In particular CORDIC is still useful for simple systems that do not have the capability of handling floating point values, or for which the floating point operations take a long time to compute. Further the CORDIC algorithm has the capability to be directly implemented in hardware which would guarantee it's use as being the most efficient method.

Finally in the analysis of the Logarithmic and Exponential functions we saw some more methods, ranging from the trivial and naive, to the detailed and reasoned. As expected the more reasoned methods that took advantage of aspects of the functions being approximated had better results than those that did not.

The Taylor method for approximating both logarithms and exponentials were good starting points, as the methods were conceptually simple with low computational complexity for each iteration. Similar to what was witnessed in the analysis of the trigonometric functions, it was very simple to calculate the number of iterations required for a given accuracy, which is a desirable property to have.

Unlike the trigonometric section there was no one method that could be used to the efficiency of both the exponential function and the logarithm function. However the two methods considered both gave significant increases in efficiency over the Taylor method. The analysis of the two resulting methods showed that they both converged at a faster rate than the standard Taylor method, and neither of the methods was significantly more computationally complex at each iteration.

A final note is that while our analysis has shown when one algorithm is better than another, and even achieved good computational times, they still fall short of the built in versions from the standard C libraries. This is due to either the libraries using some even more efficient methods than those discussed here, the libraries utilising low level programming techniques to speed up computation, or a combination of the two.

7 Preliminary References

http://math.exeter.edu/rparris/peanut/cordic.pdf
Inside your Calculator by Gerald R Rising
Wolfram Alpha